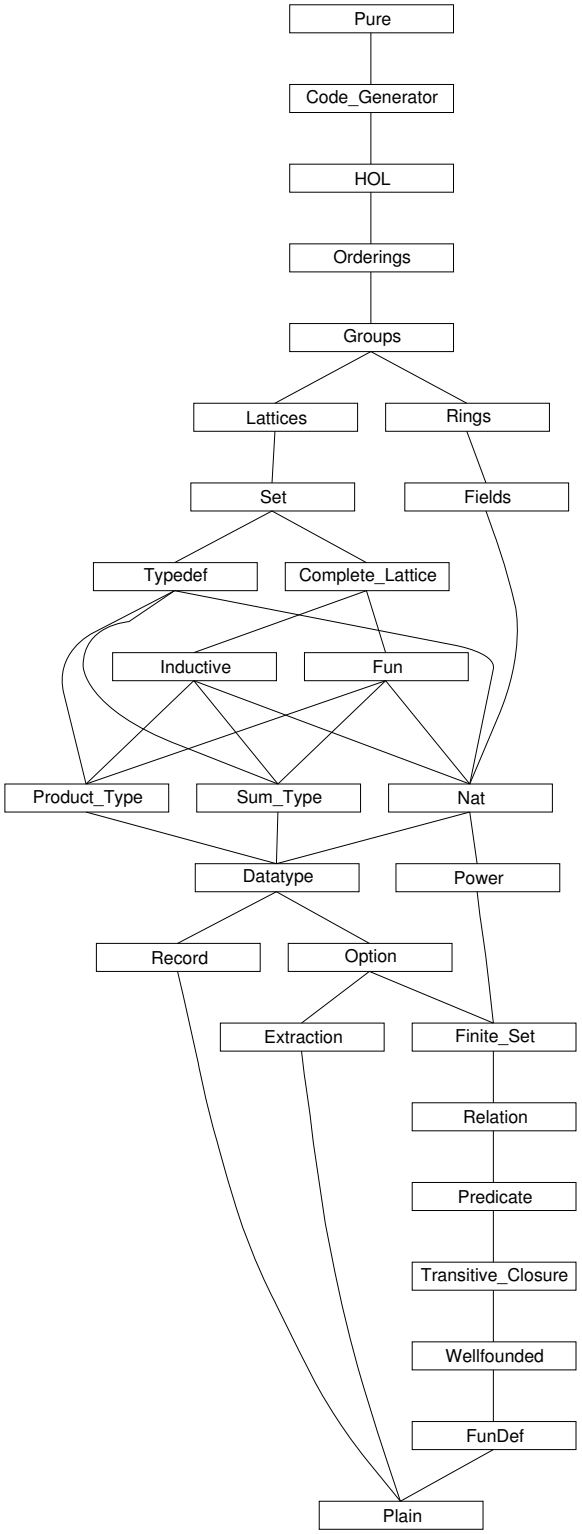
Isabelle/HOL — Higher-Order Logic

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

⟨*ML*⟩

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

⟨*ML*⟩

2.1 Primitive logic

2.1.1 Core syntax

classes *type*
default-sort *type*
⟨*ML*⟩

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*)

⟨*ML*⟩

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

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

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

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

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

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

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

axiomatization
$undefined :: 'a$

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

2.2 Fundamental rules

2.2.1 Equality

lemma *sym*: $s = t ==> t = s$
⟨*proof*⟩

lemma *ssubst*: $t = s ==> P\ s ==> P\ t$
⟨*proof*⟩

lemma *trans*: $[|\ r{=}s;\ s{=}t\ |] ==> r{=}t$
⟨*proof*⟩

lemma *meta-eq-to-obj-eq*:
assumes *meq*: $A == B$
shows $A = B$
⟨*proof*⟩

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

lemma *box-equals*: $[|\ a{=}b;\ \ a{=}c;\ \ b{=}d\ |] ==> c{=}d$
⟨*proof*⟩

For calculational reasoning:

lemma *forw-subst*: $a = b ==> P\ b ==> P\ a$
⟨*proof*⟩

lemma *back-subst*: $P\ a ==> a = b ==> P\ b$
⟨*proof*⟩

2.2.2 Congruence rules for application

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

lemma *fun-cong*: $(f::'a=>'b) = g ==> f(x)=g(x)$
⟨*proof*⟩

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

lemma *arg-cong*: $x=y ==> f(x)=f(y)$
⟨*proof*⟩

lemma *arg-cong2*: $[\![a = b; c = d]\!] \Longrightarrow f\ a\ c = f\ b\ d$
⟨*proof*⟩

lemma *cong*: $[| f = g; (x::'a) = y |] ==> f\ x = g\ y$
⟨*proof*⟩

⟨*ML*⟩

2.2.3 Equality of booleans – iff

lemma *iffI*: **assumes** $P ==> Q$ **and** $Q ==> P$ **shows** $P=Q$
⟨*proof*⟩

lemma *iffD2*: $[| P=Q; Q |] ==> P$
⟨*proof*⟩

lemma *rev-iffD2*: $[| Q; P=Q |] ==> P$
⟨*proof*⟩

lemma *iffD1*: $Q = P \Longrightarrow Q \Longrightarrow P$
⟨*proof*⟩

lemma *rev-iffD1*: $Q \Longrightarrow Q = P \Longrightarrow P$
⟨*proof*⟩

lemma *iffE*:
assumes *major*: $P=Q$
and *minor*: $[| P --> Q; Q --> P |] ==> R$
shows R
⟨*proof*⟩

2.2.4 True

lemma *TrueI*: *True*
⟨*proof*⟩

lemma *eqTrueI*: $P ==> P = True$
⟨*proof*⟩

lemma *eqTrueE*: $P = True \Longrightarrow P$
⟨*proof*⟩

2.2.5 Universal quantifier

lemma *allI*: **assumes** $!!x::'a.\ P(x)$ **shows** $ALL\ x.\ P(x)$
⟨*proof*⟩

lemma *spec*: $ALL\ x::'a.\ P(x) \Longrightarrow P(x)$
⟨*proof*⟩

lemma *allE*:
assumes *major*: $ALL\ x.\ P(x)$
and *minor*: $P(x) \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *all-dupE*:
assumes *major*: $ALL\ x.\ P(x)$
and *minor*: $[|\ P(x);\ ALL\ x.\ P(x)\ |] \Longrightarrow R$
shows R
⟨*proof*⟩

2.2.6 False

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

lemma *FalseE*: $False \Longrightarrow P$
⟨*proof*⟩

lemma *False-neq-True*: $False = True \Longrightarrow P$
⟨*proof*⟩

2.2.7 Negation

lemma *notI*:
assumes $P \Longrightarrow False$
shows $\sim P$
⟨*proof*⟩

lemma *False-not-True*: $False \sim= True$
⟨*proof*⟩

lemma *True-not-False*: $True \sim= False$
⟨*proof*⟩

lemma *notE*: $[|\ \sim P;\ \ P\ |] \Longrightarrow R$
⟨*proof*⟩

lemma *notI2*: $(P \Longrightarrow \neg\ Pa) \Longrightarrow (P \Longrightarrow Pa) \Longrightarrow \neg\ P$
⟨*proof*⟩

2.2.8 Implication

lemma *impE*:
assumes $P-->Q\ P\ Q$ ==> R
shows R
⟨*proof*⟩

lemma *rev-mp*: [| P; P --> Q |] ==> Q
⟨*proof*⟩

lemma *contrapos-nn*:
assumes *major*: ~Q
and *minor*: P==>Q
shows ~P
⟨*proof*⟩

lemma *contrapos-pn*:
assumes *major*: Q
and *minor*: P ==> ~Q
shows ~P
⟨*proof*⟩

lemma *not-sym*: t ~= s ==> s ~= t
⟨*proof*⟩

lemma *eq-neq-eq-imp-neq*: [| $x = a$; a ~= b; $b = y$ |] ==> x ~= y
⟨*proof*⟩

lemma *rev-contrapos*:
assumes *pq*: P ==> Q
and *nq*: ~Q
shows ~P
⟨*proof*⟩

2.2.9 Existential quantifier

lemma *exI*: $P\ x$ ==> $EX\ x::'a.\ P\ x$
⟨*proof*⟩

lemma *exE*:
assumes *major*: $EX\ x::'a.\ P(x)$
and *minor*: !!$x.\ P(x)$ ==> Q
shows Q
⟨*proof*⟩

2.2.10 Conjunction

lemma *conjI*: [| *P*; *Q* |] ==> *P*&*Q*
⟨*proof*⟩

lemma *conjunct1*: [| *P* & *Q* |] ==> *P*
⟨*proof*⟩

lemma *conjunct2*: [| *P* & *Q* |] ==> *Q*
⟨*proof*⟩

lemma *conjE*:
assumes *major*: *P*&*Q*
and *minor*: [| *P*; *Q* |] ==> *R*
shows *R*
⟨*proof*⟩

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

2.2.11 Disjunction

lemma *disjI1*: *P* ==> *P*|*Q*
⟨*proof*⟩

lemma *disjI2*: *Q* ==> *P*|*Q*
⟨*proof*⟩

lemma *disjE*:
assumes *major*: *P*|*Q*
and *minorP*: *P* ==> *R*
and *minorQ*: *Q* ==> *R*
shows *R*
⟨*proof*⟩

2.2.12 Classical logic

lemma *classical*:
assumes *prem*: ~*P* ==> *P*
shows *P*
⟨*proof*⟩

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

lemma *rev-notE*:
assumes *premp*: *P*
and *premnot*: ~*R* ==> ~*P*
shows *R*

⟨*proof*⟩

lemma *notnotD*: ~~*P* ==> *P*
⟨*proof*⟩

lemma *contrapos-pp*:
assumes *p1*: *Q*
and *p2*: ~*P* ==> ~*Q*
shows *P*
⟨*proof*⟩

2.2.13 Unique existence

lemma *ex1I*:
assumes *P a* !!*x*. *P*(*x*) ==> *x*=*a*
shows *EX*! *x*. *P*(*x*)
⟨*proof*⟩

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

lemma *ex-ex1I*:
assumes *ex-prem*: *EX x*. *P*(*x*)
and *eq*: !!*x y*. [| *P*(*x*); *P*(*y*) |] ==> *x*=*y*
shows *EX*! *x*. *P*(*x*)
⟨*proof*⟩

lemma *ex1E*:
assumes *major*: *EX*! *x*. *P*(*x*)
and *minor*: !!*x*. [| *P*(*x*); *ALL y*. *P*(*y*) --> *y*=*x* |] ==> *R*
shows *R*
⟨*proof*⟩

lemma *ex1-implies-ex*: *EX*! *x*. *P x* ==> *EX x*. *P x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *theI*:
assumes *P a* **and** !!*x*. *P x* ==> *x*=*a*
shows *P* (*THE x*. *P x*)
⟨*proof*⟩

lemma *theI'*: *EX*! *x*. *P x* ==> *P* (*THE x*. *P x*)
⟨*proof*⟩

lemma *theI2*:
assumes $P\ a$ $!!x.\ P\ x\ ==>\ x{=}a$ $!!x.\ P\ x\ ==>\ Q\ x$
shows $Q\ (THE\ x.\ P\ x)$
⟨*proof*⟩

lemma *the1I2*: **assumes** $EX!\ x.\ P\ x$ $\bigwedge x.\ P\ x \Longrightarrow Q\ x$ **shows** $Q\ (THE\ x.\ P\ x)$
⟨*proof*⟩

lemma *the1-equality* [*elim?*]: $[|\ EX!x.\ P\ x;\ P\ a\ |]\ ==>\ (THE\ x.\ P\ x) = a$
⟨*proof*⟩

lemma *the-sym-eq-trivial*: $(THE\ y.\ x{=}y) = x$
⟨*proof*⟩

2.2.15 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes $^{\sim}Q\ ==>\ P$ **shows** $P|Q$
⟨*proof*⟩

lemma *excluded-middle*: $^{\sim}P\ |\ P$
⟨*proof*⟩

case distinction as a natural deduction rule. Note that $\neg\ P$ is the second case, not the first

lemma *case-split* [*case-names True False*]:
assumes *prem1*: $P\ ==>\ Q$
and *prem2*: $^{\sim}P\ ==>\ Q$
shows Q
⟨*proof*⟩

lemma *impCE*:
assumes *major*: $P{-}{-}{>}Q$
and *minor*: $^{\sim}P\ ==>\ R$ $Q\ ==>\ R$
shows R
⟨*proof*⟩

lemma *impCE'*:
assumes *major*: $P{-}{-}{>}Q$
and *minor*: $Q\ ==>\ R$ $^{\sim}P\ ==>\ R$
shows R
⟨*proof*⟩

lemma *iffCE*:
assumes *major*: $P=Q$
and *minor*: $[|\ P;\ Q\ |] ==> R\ \ [|\ ~P;\ ~Q\ |] ==> R$
shows R
⟨*proof*⟩

lemma *exCI*:
assumes $ALL\ x.\ ~P(x) ==> P(a)$
shows $EX\ x.\ P(x)$
⟨*proof*⟩

2.2.16 Intuitionistic Reasoning

lemma *impE'*:
assumes *1*: $P\ --> Q$
and *2*: $Q ==> R$
and *3*: $P\ --> Q ==> P$
shows R
⟨*proof*⟩

lemma *allE'*:
assumes *1*: $ALL\ x.\ P\ x$
and *2*: $P\ x ==> ALL\ x.\ P\ x ==> Q$
shows Q
⟨*proof*⟩

lemma *notE'*:
assumes *1*: $~\ P$
and *2*: $~\ P ==> P$
shows R
⟨*proof*⟩

lemma *TrueE*: $True ==> P ==> P$ ⟨*proof*⟩
lemma *notFalseE*: $~\ False ==> P ==> P$ ⟨*proof*⟩

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*

⟨*ML*⟩

2.2.17 Atomizing meta-level connectives

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

lemma *atomize-all* [*atomize*]: $(!!x.\ P\ x) == \mathit{Trueprop}\ (\mathit{ALL}\ x.\ P\ x)$
⟨*proof*⟩

lemma *atomize-imp* [*atomize*]: $(A ==> B) == \mathit{Trueprop}\ (A --> B)$
⟨*proof*⟩

lemma *atomize-not*: $(A ==> \mathit{False}) == \mathit{Trueprop}\ (\sim A)$
⟨*proof*⟩

lemma *atomize-eq* [*atomize*]: $(x == y) == \mathit{Trueprop}\ (x = y)$
⟨*proof*⟩

lemma *atomize-conj* [*atomize*]: $(A\ \&\&\&\ B) == \mathit{Trueprop}\ (A\ \&\ B)$
⟨*proof*⟩

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

2.2.18 Atomizing elimination rules

⟨*ML*⟩

lemma *atomize-exL*[*atomize-elim*]: $(!!x.\ P\ x ==> Q) == ((\mathit{EX}\ x.\ P\ x) ==> Q)$
⟨*proof*⟩

lemma *atomize-conjL*[*atomize-elim*]: $(A ==> B ==> C) == (A\ \&\ B ==> C)$
⟨*proof*⟩

lemma *atomize-disjL*[*atomize-elim*]: $((A ==> C) ==> (B ==> C) ==> C) == ((A \mid B ==> C) ==> C)$
⟨*proof*⟩

lemma *atomize-elimL*[*atomize-elim*]: $(!!B.\ (A ==> B) ==> B) == \mathit{Trueprop}\ A$
⟨*proof*⟩

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*⟩

2.3.2 Classical Reasoner setup

lemma *imp-elim*: $P --> Q ==> (\sim R ==> P) ==> (Q ==> R) ==> R$
⟨*proof*⟩

lemma *swap*: ~ *P* ==> (~ *R* ==> *P*) ==> *R*
⟨*proof*⟩

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

⟨*ML*⟩

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*⟩

lemma *contrapos-np*: ~ *Q* ==> (~ *P* ==> *Q*) ==> *P*
⟨*proof*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

2.3.3 Simplifier

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

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
$!!P.\ (EX\ x.\ x{=}t\ \&\ P(x)) = P(t)$
$!!P.\ (EX\ x.\ t{=}x\ \&\ P(x)) = P(t)$
$!!P.\ (ALL\ x.\ x{=}t \longrightarrow P(x)) = P(t)$
$!!P.\ (ALL\ x.\ t{=}x \longrightarrow P(x)) = P(t)$
⟨*proof*⟩

lemma *disj-absorb*: $(A \mid A) = A$
⟨*proof*⟩

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

lemma *conj-absorb*: $(A\ \&\ A) = A$
⟨*proof*⟩

lemma *conj-left-absorb*: $(A \ \& \ (A \ \& \ B)) = (A \ \& \ B)$
⟨*proof*⟩

lemma *eq-ac*:
shows *eq-commute*: $(a{=}b) = (b{=}a)$
and *eq-left-commute*: $(P{=}(Q{=}R)) = (Q{=}(P{=}R))$
and *eq-assoc*: $((P{=}Q){=}R) = (P{=}(Q{=}R))$ ⟨*proof*⟩
lemma *neq-commute*: $(a^{\sim}{=}b) = (b^{\sim}{=}a)$ ⟨*proof*⟩

lemma *conj-comms*:
shows *conj-commute*: $(P\&Q) = (Q\&P)$
and *conj-left-commute*: $(P\&(Q\&R)) = (Q\&(P\&R))$ ⟨*proof*⟩
lemma *conj-assoc*: $((P\&Q)\&R) = (P\&(Q\&R))$ ⟨*proof*⟩

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

lemma *disj-comms*:
shows *disj-commute*: $(P|Q) = (Q|P)$
and *disj-left-commute*: $(P|(Q|R)) = (Q|(P|R))$ ⟨*proof*⟩
lemma *disj-assoc*: $((P|Q)|R) = (P|(Q|R))$ ⟨*proof*⟩

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

lemma *conj-disj-distribL*: $(P\&(Q|R)) = (P\&Q \mid P\&R)$ ⟨*proof*⟩
lemma *conj-disj-distribR*: $((P|Q)\&R) = (P\&R \mid Q\&R)$ ⟨*proof*⟩

lemma *disj-conj-distribL*: $(P|(Q\&R)) = ((P|Q) \ \& \ (P|R))$ ⟨*proof*⟩
lemma *disj-conj-distribR*: $((P\&Q)|R) = ((P|R) \ \& \ (Q|R))$ ⟨*proof*⟩

lemma *imp-conjR*: $(P \dashrightarrow (Q\&R)) = ((P\dashrightarrow Q) \ \& \ (P\dashrightarrow R))$ ⟨*proof*⟩
lemma *imp-conjL*: $((P\&Q) \dashrightarrow R) \ = (P \dashrightarrow (Q \dashrightarrow R))$ ⟨*proof*⟩
lemma *imp-disjL*: $((P|Q) \dashrightarrow R) = ((P\dashrightarrow R)\&(Q\dashrightarrow R))$ ⟨*proof*⟩

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

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

lemma *imp-disj1*: $((P\dashrightarrow Q)|R) = (P\dashrightarrow Q|R)$ ⟨*proof*⟩
lemma *imp-disj2*: $(Q|(P\dashrightarrow R)) = (P\dashrightarrow Q|R)$ ⟨*proof*⟩

lemma *imp-cong*: $(P = P') \Longrightarrow (P' \Longrightarrow (Q = Q')) \Longrightarrow ((P \dashrightarrow Q) = (P' \dashrightarrow Q'))$
⟨*proof*⟩

lemma *de-Morgan-disj*: $(^{\sim}(P \mid Q)) = (^{\sim}P \ \& \ ^{\sim}Q)$ ⟨*proof*⟩
lemma *de-Morgan-conj*: $(^{\sim}(P \ \& \ Q)) = (^{\sim}P \mid ^{\sim}Q)$ ⟨*proof*⟩
lemma *not-imp*: $(^{\sim}(P \dashrightarrow Q)) = (P \ \& \ ^{\sim}Q)$ ⟨*proof*⟩
lemma *not-iff*: $(P^{\sim}{=}Q) = (P = (^{\sim}Q))$ ⟨*proof*⟩
lemma *disj-not1*: $(^{\sim}P \mid Q) = (P \dashrightarrow Q)$ ⟨*proof*⟩

lemma *disj-not2*: $(P \mid \sim Q) = (Q \dashrightarrow P)$ — changes orientation :-(
⟨*proof*⟩
lemma *imp-conv-disj*: $(P \dashrightarrow Q) = ((\sim P) \mid Q)$ ⟨*proof*⟩

lemma *iff-conv-conj-imp*: $(P = Q) = ((P \dashrightarrow Q) \;\&\; (Q \dashrightarrow P))$ ⟨*proof*⟩

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.
⟨*proof*⟩

lemma *not-all*: $(\sim (!\ x.\ P(x))) = (?\ x.\sim P(x))$ ⟨*proof*⟩
lemma *imp-all*: $((!\ x.\ P\ x) \dashrightarrow Q) = (?\ x.\ P\ x \dashrightarrow Q)$ ⟨*proof*⟩
lemma *not-ex*: $(\sim (?\ x.\ P(x))) = (!\ x.\sim P(x))$ ⟨*proof*⟩
lemma *imp-ex*: $((?\ x.\ P\ x) \dashrightarrow Q) = (!\ x.\ P\ x \dashrightarrow Q)$ ⟨*proof*⟩
lemma *all-not-ex*: $(ALL\ x.\ P\ x) = (\sim (EX\ x.\ \sim\ P\ x\ ))$ ⟨*proof*⟩

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

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

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

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

lemma *rev-conj-cong*:
$(Q = Q') \Longrightarrow (Q' \Longrightarrow (P = P')) \Longrightarrow ((P \;\&\; Q) = (P' \;\&\; Q'))$
⟨*proof*⟩

The | congruence rule: not included by default!

lemma *disj-cong*:
$(P = P') \Longrightarrow (\sim P' \Longrightarrow (Q = Q')) \Longrightarrow ((P \mid Q) = (P' \mid Q'))$
⟨*proof*⟩

if-then-else rules

lemma *if-True* [*code*]: $(if\ True\ then\ x\ else\ y) = x$
⟨*proof*⟩

lemma *if-False* [*code*]: $(if\ False\ then\ x\ else\ y) = y$
⟨*proof*⟩

lemma *if-P*: $P \Longrightarrow (if\ P\ then\ x\ else\ y) = x$
⟨*proof*⟩

lemma *if-not-P*: $\sim P \Longrightarrow (\textit{if } P \textit{ then } x \textit{ else } y) = y$
⟨*proof*⟩

lemma *split-if*: $P\ (\textit{if } Q \textit{ then } x \textit{ else } y) = ((Q \longrightarrow P(x)) \ \&\ (\sim Q \longrightarrow P(y)))$
⟨*proof*⟩

lemma *split-if-asm*: $P\ (\textit{if } Q \textit{ then } x \textit{ else } y) = (\sim((Q \ \&\ \sim P\ x) \mid (\sim Q \ \&\ \sim P\ y)))$
⟨*proof*⟩

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

lemma *if-cancel*: $(\textit{if } c \textit{ then } x \textit{ else } x) = x$
⟨*proof*⟩

lemma *if-eq-cancel*: $(\textit{if } x = y \textit{ then } y \textit{ else } x) = x$
⟨*proof*⟩

lemma *if-bool-eq-conj*: $(\textit{if } P \textit{ then } Q \textit{ else } R) = ((P \longrightarrow Q) \ \&\ (\sim P \longrightarrow R))$
— This form is useful for expanding *if*s on the RIGHT of the ⟹ symbol.
⟨*proof*⟩

lemma *if-bool-eq-disj*: $(\textit{if } P \textit{ then } Q \textit{ else } R) = ((P \& Q) \mid (\sim P \& R))$
— And this form is useful for expanding *if*s on the LEFT.
⟨*proof*⟩

lemma *Eq-TrueI*: $P \Longrightarrow P \equiv \textit{True}$ ⟨*proof*⟩
lemma *Eq-FalseI*: $\sim P \Longrightarrow P \equiv \textit{False}$ ⟨*proof*⟩

let rules for simproc

lemma *Let-folded*: $f\ x \equiv g\ x \Longrightarrow \textit{Let } x\ f \equiv \textit{Let } x\ g$
⟨*proof*⟩

lemma *Let-unfold*: $f\ x \equiv g \Longrightarrow \textit{Let } x\ f \equiv g$
⟨*proof*⟩

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*: $(\textit{PROP } P \Longrightarrow \textit{PROP } Q)$
shows $\textit{PROP } P \ =\textit{simp}\Rightarrow\ \textit{PROP } Q$
⟨*proof*⟩

lemma *simp-impliesE*:
assumes *PQ*: $\textit{PROP } P \ =\textit{simp}\Rightarrow\ \textit{PROP } Q$
and *P*: *PROP P*

and *QR*: *PROP Q* $\Longrightarrow$ *PROP R*
shows *PROP R*
⟨*proof*⟩

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*⟩

lemma *uncurry*:
assumes $P \longrightarrow Q \longrightarrow R$
shows $P \wedge Q \longrightarrow R$
⟨*proof*⟩

lemma *iff-allI*:
assumes $\bigwedge x.\ P\ x = Q\ x$
shows $(\forall x.\ P\ x) = (\forall x.\ Q\ x)$
⟨*proof*⟩

lemma *iff-exI*:
assumes $\bigwedge x.\ P\ x = Q\ x$
shows $(\exists x.\ P\ x) = (\exists x.\ Q\ x)$
⟨*proof*⟩

lemma *all-comm*:
$(\forall x\ y.\ P\ x\ y) = (\forall y\ x.\ P\ x\ y)$
⟨*proof*⟩

lemma *ex-comm*:
$(\exists x\ y.\ P\ x\ y) = (\exists y\ x.\ P\ x\ y)$
⟨*proof*⟩

⟨*ML*⟩

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

⟨*ML*⟩

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

lemma *ex-simps*:
!!*P Q*. (*EX x*. *P x* & *Q*) = ((*EX x*. *P x*) & *Q*)
!!*P Q*. (*EX x*. *P* & *Q x*) = (*P* & (*EX x*. *Q x*))
!!*P Q*. (*EX x*. *P x* | *Q*) = ((*EX x*. *P x*) | *Q*)
!!*P Q*. (*EX x*. *P* | *Q x*) = (*P* | (*EX x*. *Q x*))
!!*P Q*. (*EX x*. *P x* −−> *Q*) = ((*ALL x*. *P x*) −−> *Q*)
!!*P Q*. (*EX x*. *P* −−> *Q x*) = (*P* −−> (*EX x*. *Q x*))
— Miniscoping: pushing in existential quantifiers.

⟨*proof*⟩

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

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*⟩

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*)

⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *eq-cong2*:
assumes $u = u'$
shows $(t \equiv u) \equiv (t \equiv u')$
⟨*proof*⟩

lemma *if-distrib*:
f (*if c then x else y*) = (*if c then f x else f y*)
⟨*proof*⟩

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 $(x = z) = (y = z)$
⟨*proof*⟩

2.3.4 Generic cases and induction

Rule projections:

⟨*ML*⟩

definition *induct-forall* **where**
induct-forall P == $\forall x.$ *P x*

definition *induct-implies* **where**
induct-implies A B == $A \longrightarrow B$

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*))
⟨*proof*⟩

lemma *induct-implies-eq*: (*A* ==> *B*) == *Trueprop* (*induct-implies A B*)
⟨*proof*⟩

lemma *induct-equal-eq*: (*x* == *y*) == *Trueprop* (*induct-equal x y*)
⟨*proof*⟩

lemma *induct-conj-eq*: (*A* &&& *B*) == *Trueprop* (*induct-conj A B*)
⟨*proof*⟩

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-forall* (λ*x*. *induct-conj* (*A x*) (*B x*)) =
induct-conj (*induct-forall A*) (*induct-forall B*)
⟨*proof*⟩

lemma *induct-implies-conj*: *induct-implies C* (*induct-conj A B*) =
induct-conj (*induct-implies C A*) (*induct-implies C B*)
⟨*proof*⟩

lemma *induct-conj-curry*: (*induct-conj A B* ==> *PROP C*) == (*A* ==> *B* ==> *PROP C*)
⟨*proof*⟩

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

lemma *induct-trueI*: *induct-true*
⟨*proof*⟩

Method setup.

⟨*ML*⟩

Pre-simplification of induction and cases rules

lemma [*induct-simp*]: (!!x. *induct-equal x t* ==> *PROP P x*) == *PROP P t*
⟨*proof*⟩

lemma [*induct-simp*]: (!!x. *induct-equal t x* ==> *PROP P x*) == *PROP P t*
⟨*proof*⟩

lemma [*induct-simp*]: (*induct-false* ==> *P*) == *Trueprop induct-true*
⟨*proof*⟩

lemma [*induct-simp*]: (*induct-true* ==> *PROP P*) == *PROP P*
⟨*proof*⟩

lemma [*induct-simp*]: (*PROP P* ==> *induct-true*) == *Trueprop induct-true*
⟨*proof*⟩

lemma [*induct-simp*]: (!!x. *induct-true*) == *Trueprop induct-true*
⟨*proof*⟩

lemma [*induct-simp*]: *induct-implies induct-true P* == *P*
⟨*proof*⟩

lemma [*induct-simp*]: (x = x) = *True*
⟨*proof*⟩

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

⟨*ML*⟩

2.3.5 Coherent logic

⟨*ML*⟩

2.3.6 Reorienting equalities

⟨*ML*⟩

2.4 Other simple lemmas and lemma duplicates

lemma *ex1-eq* [*iff*]: *EX*! x. x = t *EX*! x. t = x
⟨*proof*⟩

lemma *choice-eq*: (*ALL x. EX*! y. *P x y*) = (*EX*! f. *ALL x. P x* (*f x*))
⟨*proof*⟩

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*))

$(\neg\ \neg(P)) = P$
⟨*proof*⟩

2.5 Basic ML bindings

⟨*ML*⟩

2.6 Code generator setup

2.6.1 SML code generator setup

⟨*ML*⟩

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* -))

⟨*ML*⟩

2.6.2 Generic code generator preprocessor setup

⟨*ML*⟩

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* =)
⟨*proof*⟩

lemma *eq-refl*: *eq x x* ⟷ *True*
⟨*proof*⟩

lemma *equals-eq*: (*op* =) ≡ *eq*
⟨*proof*⟩

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

end

declare *equals-eq* [*code*]

⟨*ML*⟩

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*) ⟨*proof*⟩

lemma [*code*]:
shows *False* ∧ *P* ⟷ *False*
and *True* ∧ *P* ⟷ *P*
and *P* ∧ *False* ⟷ *False*
and *P* ∧ *True* ⟷ *P* ⟨*proof*⟩

lemma [*code*]:
shows *False* ∨ *P* ⟷ *P*
and *True* ∨ *P* ⟷ *True*
and *P* ∨ *False* ⟷ *P*
and *P* ∨ *True* ⟷ *True* ⟨*proof*⟩

lemma [*code*]:
shows (*False* ⟶ *P*) ⟷ *True*
and (*True* ⟶ *P*) ⟷ *P*
and (*P* ⟶ *False*) ⟷ ¬ *P*

and $(P \longrightarrow True) \longleftrightarrow True$ ⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

lemma *eq-itself-code* [*code*]:
eq-class.eq TYPE($'a$) *TYPE*($'a$) ⟷ *True*
⟨*proof*⟩

Equality

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

⟨*ML*⟩

lemma *equals-alias-cert*: *OFCLASS*($'a$, *eq-class*) ≡ ((*op* = :: $'a$ ⇒ $'a$ ⇒ *bool*) ≡ *eq*) (**is** *?ofclass* ≡ *?eq*)
⟨*proof*⟩

⟨*ML*⟩

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

Cases

lemma *Let-case-cert*:
assumes *CASE* ≡ (λ*x*. *Let x f*)
shows *CASE x* ≡ *f x*
⟨*proof*⟩

lemma *If-case-cert*:
assumes *CASE* ≡ (λ*b*. *If b f g*)
shows (*CASE True* ≡ *f*) &&& (*CASE False* ≡ *g*)
⟨*proof*⟩

⟨*ML*⟩

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*

⟨*ML*⟩

2.7 Counterexample Search Units

2.7.1 Quickcheck

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

2.7.2 Nitpick setup

⟨*ML*⟩

2.8 Preprocessing for the predicate compiler

⟨*ML*⟩

2.9 Legacy tactics and ML bindings

⟨*ML*⟩

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* $\leq$) **and**
less-eq ((-/ $\leq$ -) [*51*, *51*] *50*)

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

notation (*input*)
greater-eq (**infix** $\geq$ *50*)

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

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.
⟨*proof*⟩

lemma *less-irrefl* [*iff*]: $\neg\ x < x$
⟨*proof*⟩

lemma *less-imp-le*: $x < y \Longrightarrow x \leq y$
⟨*proof*⟩

Asymmetry.

lemma *less-not-sym*: $x < y \Longrightarrow \neg (y < x)$
⟨*proof*⟩

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

Transitivity.

lemma *less-trans*: $x < y \Longrightarrow y < z \Longrightarrow x < z$
⟨*proof*⟩

lemma *le-less-trans*: $x \leq y \Longrightarrow y < z \Longrightarrow x < z$
⟨*proof*⟩

lemma *less-le-trans*: $x < y \Longrightarrow y \leq z \Longrightarrow x < z$
⟨*proof*⟩

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

lemma *less-imp-not-less*: $x < y \Longrightarrow (\neg\ y < x) \longleftrightarrow \mathit{True}$
⟨*proof*⟩

lemma *less-imp-triv*: $x < y \Longrightarrow (y < x \longrightarrow P) \longleftrightarrow \mathit{True}$
⟨*proof*⟩

Transitivity rules for calculational reasoning

lemma *less-asym'*: $a < b \Longrightarrow b < a \Longrightarrow P$
⟨*proof*⟩

Dual order

lemma *dual-preorder*:
class.preorder $(op \geq)$ $(op >)$
⟨*proof*⟩

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*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$
⟨*proof*⟩

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x = y$
— NOT suitable for iff, since it can cause PROOF FAILED.
⟨*proof*⟩

lemma *le-imp-less-or-eq*: $x \leq y \Longrightarrow x < y \vee x = y$
⟨*proof*⟩

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

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

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

Transitivity rules for calculational reasoning

lemma *neq-le-trans*: $a \neq b \Longrightarrow a \leq b \Longrightarrow a < b$

⟨*proof*⟩

lemma *le-neq-trans*: $a \leq b \Longrightarrow a \neq b \Longrightarrow a < b$
⟨*proof*⟩

Asymmetry.

lemma *eq-iff*: $x = y \longleftrightarrow x \leq y \wedge y \leq x$
⟨*proof*⟩

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *less-imp-neq*: $x < y \Longrightarrow x \neq y$
⟨*proof*⟩

Least value operator

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

lemma *Least-equality*:
  assumes $P\ x$
    and $\bigwedge y.\ P\ y \Longrightarrow x \leq y$
  shows *Least* $P = x$
⟨*proof*⟩

lemma *LeastI2-order*:
  assumes $P\ x$
    and $\bigwedge y.\ P\ y \Longrightarrow x \leq y$
    and $\bigwedge x.\ P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow x \leq y \Longrightarrow Q\ x$
  shows Q (*Least* P)
⟨*proof*⟩

Dual order

lemma *dual-order*:
  class.order $(op \geq)$ $(op >)$
⟨*proof*⟩

end

3.4 Linear (total) orders

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

lemma *less-linear*: $x < y \vee x = y \vee y < x$
⟨*proof*⟩

lemma *le-less-linear*: $x \leq y \vee y < x$
⟨*proof*⟩

lemma *le-cases* [*case-names le ge*]:
$(x \leq y \Longrightarrow P) \Longrightarrow (y \leq x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *linorder-cases* [*case-names less equal greater*]:
$(x < y \Longrightarrow P) \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow (y < x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *not-less*: $\neg\ x < y \longleftrightarrow y \leq x$
⟨*proof*⟩

lemma *not-less-iff-gr-or-eq*:
$\neg(x < y) \longleftrightarrow (x > y \mid x = y)$
⟨*proof*⟩

lemma *not-le*: $\neg\ x \leq y \longleftrightarrow y < x$
⟨*proof*⟩

lemma *neq-iff*: $x \neq y \longleftrightarrow x < y \vee y < x$
⟨*proof*⟩

lemma *neqE*: $x \neq y \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *antisym-conv1*: $\neg\ x < y \Longrightarrow x \leq y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *antisym-conv2*: $x \leq y \Longrightarrow \neg\ x < y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *antisym-conv3*: $\neg\ y < x \Longrightarrow \neg\ x < y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *leI*: $\neg\ x < y \Longrightarrow y \leq x$
⟨*proof*⟩

lemma *leD*: $y \leq x \Longrightarrow \neg\ x < y$
⟨*proof*⟩

lemma *not-leE*: $\neg\ y \leq x \Longrightarrow x < y$
⟨*proof*⟩

Dual order

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

⟨*proof*⟩

min/max

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

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

lemma *min-le-iff-disj*:
  min x y ≤ *z* ⟷ *x* ≤ *z* ∨ *y* ≤ *z*
⟨*proof*⟩

lemma *le-max-iff-disj*:
  z ≤ *max x y* ⟷ *z* ≤ *x* ∨ *z* ≤ *y*
⟨*proof*⟩

lemma *min-less-iff-disj*:
  min x y < *z* ⟷ *x* < *z* ∨ *y* < *z*
⟨*proof*⟩

lemma *less-max-iff-disj*:
  z < *max x y* ⟷ *z* < *x* ∨ *z* < *y*
⟨*proof*⟩

lemma *min-less-iff-conj* [*simp*]:
  z < *min x y* ⟷ *z* < *x* ∧ *z* < *y*
⟨*proof*⟩

lemma *max-less-iff-conj* [*simp*]:
  max x y < *z* ⟷ *x* < *z* ∧ *y* < *z*
⟨*proof*⟩

lemma *split-min* [*no-atp*]:
  P (*min i j*) ⟷ (*i* ≤ *j* ⟶ *P i*) ∧ (¬ *i* ≤ *j* ⟶ *P j*)
⟨*proof*⟩

lemma *split-max* [*no-atp*]:
  P (*max i j*) ⟷ (*i* ≤ *j* ⟶ *P j*) ∧ (¬ *i* ≤ *j* ⟶ *P i*)
⟨*proof*⟩

end

Explicit dictionaries for code generation

lemma *min-ord-min* [*code*, *code-unfold*, *code-inline del*]:
  min = *ord.min* (*op* ≤)
  ⟨*proof*⟩

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

lemma *max-ord-max* [*code*, *code-unfold*, *code-inline del*]:
max = *ord.max* (*op* ≤)
⟨*proof*⟩

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

3.5 Reasoning tools setup

⟨*ML*⟩

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

⟨*ML*⟩

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*

⟨*ML*⟩

3.7 Transitivity reasoning

context *ord*
begin

lemma *ord-le-eq-trans*: $a \le b \Longrightarrow b = c \Longrightarrow a \le c$
⟨*proof*⟩

lemma *ord-eq-le-trans*: $a = b \Longrightarrow b \le c \Longrightarrow a \le c$
⟨*proof*⟩

lemma *ord-less-eq-trans*: $a < b \Longrightarrow b = c \Longrightarrow a < c$
⟨*proof*⟩

lemma *ord-eq-less-trans*: $a = b \Longrightarrow b < c \Longrightarrow a < c$
⟨*proof*⟩

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*⟩

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*⟩

lemma *order-le-less-subst2*: $(a::'a::order) <= b \Longrightarrow f\ b < (c::'c::order) \Longrightarrow$
$(!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y) \Longrightarrow f\ a < c$
⟨*proof*⟩

lemma *order-le-less-subst1*: $(a::'a::order) <= f\ b \Longrightarrow (b::'b::order) < c \Longrightarrow$
$(!!x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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

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

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

lemma *xt2*:
$(a::'a::order) >= f\ b \Longrightarrow b >= c \Longrightarrow (!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow a >= f\ c$
⟨*proof*⟩

lemma *xt3*: $(a::'a::order) >= b ==> (f\ b::'b::order) >= c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> f\ a >= c$
⟨*proof*⟩

lemma *xt4*: $(a::'a::order) > f\ b ==> (b::'b::order) >= c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt5*: $(a::'a::order) > b ==> (f\ b::'b::order) >= c==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> f\ a > c$
⟨*proof*⟩

lemma *xt6*: $(a::'a::order) >= f\ b ==> b > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt7*: $(a::'a::order) >= b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> f\ a > c$
⟨*proof*⟩

lemma *xt8*: $(a::'a::order) > f\ b ==> (b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt9*: $(a::'a::order) > b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> f\ a > c$
⟨*proof*⟩

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 $(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow mono\ f$
⟨*proof*⟩

lemma *monoD* [*dest?*]:
fixes $f :: 'a \Rightarrow 'b::order$
shows $mono\ f \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
⟨*proof*⟩

definition *strict-mono* :: $('a \Rightarrow 'b{::}order) \Rightarrow bool$ **where**
strict-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*
⟨*proof*⟩

lemma *strict-monoD* [*dest?*]:
strict-mono f $\Longrightarrow x < y \Longrightarrow f\ x < f\ y$
⟨*proof*⟩

lemma *strict-mono-mono* [*dest?*]:
assumes *strict-mono f*
shows *mono f*
⟨*proof*⟩

end

context *linorder*
begin

lemma *strict-mono-eq*:
assumes *strict-mono f*
shows $f\ x = f\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *strict-mono-less-eq*:
assumes *strict-mono f*
shows $f\ x \leq f\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *strict-mono-less*:
assumes *strict-mono f*
shows $f\ x < f\ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *min-of-mono*:
fixes $f :: 'a \Rightarrow 'b{::}linorder$
shows *mono f* $\Longrightarrow$ *min* $(f\ m)\ (f\ n) = f\ (min\ m\ n)$
⟨*proof*⟩

lemma *max-of-mono*:
fixes $f :: 'a \Rightarrow 'b{::}linorder$
shows *mono f* $\Longrightarrow$ *max* $(f\ m)\ (f\ n) = f\ (max\ m\ n)$
⟨*proof*⟩

end

lemma *min-leastL*: $(!!x.\ least <= x) ==> min\ least\ x = least$
⟨*proof*⟩

lemma *max-leastL*: $(!!x.\ least <= x) ==> max\ least\ x = x$
⟨*proof*⟩

lemma *min-leastR*: $(\bigwedge x::'a::order.\ least \leq x) \Longrightarrow min\ x\ least = least$
⟨*proof*⟩

lemma *max-leastR*: $(\bigwedge x::'a::order.\ least \leq x) \Longrightarrow max\ x\ least = x$
⟨*proof*⟩

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*⟩

lemma *dense-le-bounded*:
  fixes $x\ y\ z :: 'a$
  assumes $x < y$
  assumes $*$: $\bigwedge w.\ [\![\ x < w\ ;\ w < y\ ]\!] \Longrightarrow w \leq z$
  shows $y \leq z$
⟨*proof*⟩

end

3.11 Wellorders

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

begin

lemma *wellorder-Least-lemma*:
fixes $k :: 'a$
assumes $P\ k$
shows *LeastI*: $P\ (LEAST\ x.\ P\ x)$ **and** *Least-le*: $(LEAST\ x.\ P\ x) \leq k$
⟨*proof*⟩
lemma *LeastI-ex*: $\exists\, x.\ P\ x \Longrightarrow P\ (Least\ P)$
⟨*proof*⟩

lemma *LeastI2*:
$P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (Least\ P)$
⟨*proof*⟩

lemma *LeastI2-ex*:
$\exists\, a.\ P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (Least\ P)$
⟨*proof*⟩

lemma *not-less-Least*: $k < (LEAST\ x.\ P\ x) \Longrightarrow \neg\ P\ k$
⟨*proof*⟩

end

3.12 Order on bool

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

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

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

definition
top-bool-eq: $top = True$

definition
bot-bool-eq: $bot = False$

instance ⟨*proof*⟩

end

lemma *le-boolI*: $(P \Longrightarrow Q) \Longrightarrow P \leq Q$
⟨*proof*⟩

lemma *le-boolI'*: $P \longrightarrow Q \Longrightarrow P \leq Q$
⟨*proof*⟩

lemma *le-boolE*: $P \leq Q \Longrightarrow P \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *le-boolD*: $P \leq Q \Longrightarrow P \longrightarrow Q$
⟨*proof*⟩

lemma *bot-boolE*: *bot* $\Longrightarrow P$
⟨*proof*⟩

lemma *top-boolI*: *top*
⟨*proof*⟩

lemma [*code*]:
$False \leq b \longleftrightarrow True$
$True \leq b \longleftrightarrow b$
$False < b \longleftrightarrow b$
$True < b \longleftrightarrow False$
⟨*proof*⟩

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 ⟨*proof*⟩

end

instance *fun* :: (*type*, *preorder*) *preorder* ⟨*proof*⟩

instance *fun* :: (*type*, *order*) *order* ⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

lemma *le-funI*: $(\bigwedge x.\ f\ x \leq g\ x) \Longrightarrow f \leq g$
⟨*proof*⟩

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

lemma *le-funD*: $f \leq g \Longrightarrow f\ x \leq g\ x$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*) $_\diamond$

lemma *Let-0* [*simp*]: *Let 0 f* = *f 0*
  ⟨*proof*⟩

lemma *Let-1* [*simp*]: *Let 1 f* = *f 1*
  ⟨*proof*⟩

⟨*ML*⟩

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

class *minus* =
  fixes *minus* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** − *65*)

class *uminus* =

fixes *uminus* :: $'a \Rightarrow 'a$ $(-\ \text{-}\ [81]\ 80)$

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

⟨*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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *zero-reorient*: *0* = *x* ⟷ *x* = *0*
⟨*proof*⟩

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

sublocale *comm-monoid-add* < *add*!: *comm-monoid plus 0* ⟨*proof*⟩

subclass (**in** *comm-monoid-add*) *monoid-add* ⟨*proof*⟩

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*⟩

lemma *one-reorient*: *1* = *x* ⟷ *x* = *1*
⟨*proof*⟩

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

sublocale *comm-monoid-mult* < *mult*!: *comm-monoid times 1* ⟨*proof*⟩

subclass (**in** *comm-monoid-mult*) *monoid-mult* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-right-cancel* [*simp*]:
  b + *a* = *c* + *a* ⟷ *b* = *c*
⟨*proof*⟩

end

class *cancel-ab-semigroup-add* = *ab-semigroup-add* +
  assumes *add-imp-eq*: $a + b = a + c \Longrightarrow b = c$
begin

subclass *cancel-semigroup-add*
⟨*proof*⟩

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*⟩

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

lemma *minus-zero* [*simp*]: $- 0 = 0$
⟨*proof*⟩

lemma *minus-minus* [*simp*]: $- (- a) = a$
⟨*proof*⟩

lemma *right-minus* [*simp*]: $a + - a = 0$
⟨*proof*⟩

lemma *minus-add-cancel*: $- a + (a + b) = b$
⟨*proof*⟩

lemma *add-minus-cancel*: $a + (- a + b) = b$
⟨*proof*⟩

lemma *minus-add*: $- (a + b) = - b + - a$
⟨*proof*⟩

lemma *right-minus-eq*: $a - b = 0 \longleftrightarrow a = b$
⟨*proof*⟩

lemma *diff-self* [*simp*]: $a - a = 0$
⟨*proof*⟩

lemma *diff-0* [*simp*]: $0 - a = - a$

⟨*proof*⟩

lemma *diff-0-right* [*simp*]: $a - 0 = a$
⟨*proof*⟩

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + b$
⟨*proof*⟩

lemma *neg-equal-iff-equal* [*simp*]:
$- a = - b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *neg-equal-0-iff-equal* [*simp*]:
$- a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *neg-0-equal-iff-equal* [*simp*]:
$0 = - a \longleftrightarrow 0 = a$
⟨*proof*⟩

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*:
$a = - b \longleftrightarrow b = - a$
⟨*proof*⟩

lemma *minus-equation-iff*:
$- a = b \longleftrightarrow - b = a$
⟨*proof*⟩

lemma *diff-add-cancel*: $a - b + b = a$
⟨*proof*⟩

lemma *add-diff-cancel*: $a + b - b = a$
⟨*proof*⟩

declare *diff-minus*[*symmetric*, *algebra-simps*, *field-simps*]

lemma *eq-neg-iff-add-eq-0*: $a = - b \longleftrightarrow a + b = 0$
⟨*proof*⟩

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*⟩

subclass *cancel-comm-monoid-add*
⟨*proof*⟩

lemma *uminus-add-conv-diff*[*algebra-simps*, *field-simps*]:
$- a + b = b - a$
⟨*proof*⟩

lemma *minus-add-distrib* [*simp*]:
$- (a + b) = - a + - b$
⟨*proof*⟩

lemma *minus-diff-eq* [*simp*]:
$- (a - b) = b - a$
⟨*proof*⟩

lemma *add-diff-eq*[*algebra-simps*, *field-simps*]: $a + (b - c) = (a + b) - c$
⟨*proof*⟩

lemma *diff-add-eq*[*algebra-simps*, *field-simps*]: $(a - b) + c = (a + c) - b$
⟨*proof*⟩

lemma *diff-eq-eq*[*algebra-simps*, *field-simps*]: $a - b = c \longleftrightarrow a = c + b$
⟨*proof*⟩

lemma *eq-diff-eq*[*algebra-simps*, *field-simps*]: $a = c - b \longleftrightarrow a + b = c$
⟨*proof*⟩

lemma *diff-diff-eq*[*algebra-simps*, *field-simps*]: $(a - b) - c = a - (b + c)$
⟨*proof*⟩

lemma *diff-diff-eq2*[*algebra-simps*, *field-simps*]: $a - (b - c) = (a + c) - b$
⟨*proof*⟩

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$
⟨*proof*⟩

lemma *diff-eq-0-iff-eq* [*simp*, *no-atp*]: $a - b = 0 \longleftrightarrow a = b$
⟨*proof*⟩

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*:
$a \leq b \Longrightarrow a + c \leq b + c$
⟨*proof*⟩

non-strict, in both arguments

lemma *add-mono*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow a + c \leq b + d$
⟨*proof*⟩

end

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

lemma *add-strict-left-mono*:
$a < b \Longrightarrow c + a < c + b$
⟨*proof*⟩

lemma *add-strict-right-mono*:
$a < b \Longrightarrow a + c < b + c$
⟨*proof*⟩

Strict monotonicity in both arguments

lemma *add-strict-mono*:
$a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$
⟨*proof*⟩

lemma *add-less-le-mono*:
$a < b \Longrightarrow c \leq d \Longrightarrow a + c < b + d$
⟨*proof*⟩

lemma *add-le-less-mono*:
$a \leq b \Longrightarrow c < d \Longrightarrow a + c < b + d$
⟨*proof*⟩

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*⟩

lemma *add-less-imp-less-right*:
$a + c < b + c \Longrightarrow a < b$
⟨*proof*⟩

lemma *add-less-cancel-left* [*simp*]:
$c + a < c + b \longleftrightarrow a < b$
⟨*proof*⟩

lemma *add-less-cancel-right* [*simp*]:
$a + c < b + c \longleftrightarrow a < b$
⟨*proof*⟩

lemma *add-le-cancel-left* [*simp*]:
$c + a \leq c + b \longleftrightarrow a \leq b$
⟨*proof*⟩

lemma *add-le-cancel-right* [*simp*]:
$a + c \leq b + c \longleftrightarrow a \leq b$
⟨*proof*⟩

lemma *add-le-imp-le-right*:
$a + c \leq b + c \Longrightarrow a \leq b$
⟨*proof*⟩

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

lemma *min-add-distrib-left*:
$min\ x\ y + z = min\ (x + z)\ (y + z)$
⟨*proof*⟩

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** $0 \leq b$ **shows** $0 < a + b$
⟨*proof*⟩

lemma *add-pos-pos*:
assumes $0 < a$ **and** $0 < b$ **shows** $0 < a + b$
⟨*proof*⟩

lemma *add-nonneg-pos*:
assumes $0 \leq a$ **and** $0 < b$ **shows** $0 < a + b$
⟨*proof*⟩

lemma *add-nonneg-nonneg* [*simp*]:
assumes $0 \leq a$ **and** $0 \leq b$ **shows** $0 \leq a + b$
⟨*proof*⟩

lemma *add-neg-nonpos*:
assumes $a < 0$ **and** $b \leq 0$ **shows** $a + b < 0$
⟨*proof*⟩

lemma *add-neg-neg*:
assumes $a < 0$ **and** $b < 0$ **shows** $a + b < 0$
⟨*proof*⟩

lemma *add-nonpos-neg*:
assumes $a \leq 0$ **and** $b < 0$ **shows** $a + b < 0$
⟨*proof*⟩

lemma *add-nonpos-nonpos*:
assumes $a \leq 0$ **and** $b \leq 0$ **shows** $a + b \leq 0$
⟨*proof*⟩

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*⟩

end

class *ordered-ab-group-add* =
ab-group-add + *ordered-ab-semigroup-add*
begin

subclass *ordered-cancel-ab-semigroup-add* ⟨*proof*⟩

subclass *ordered-ab-semigroup-add-imp-le*
⟨*proof*⟩

subclass *ordered-comm-monoid-add* ⟨*proof*⟩

lemma *max-diff-distrib-left*:
shows $max\ x\ y - z = max\ (x - z)\ (y - z)$
⟨*proof*⟩

lemma *min-diff-distrib-left*:
shows $min\ x\ y - z = min\ (x - z)\ (y - z)$
⟨*proof*⟩

lemma *le-imp-neg-le*:
assumes $a \leq b$ **shows** $-b \leq -a$
⟨*proof*⟩

lemma *neg-le-iff-le* [*simp*]: $- b \leq - a \longleftrightarrow a \leq b$
⟨*proof*⟩

lemma *neg-le-0-iff-le* [*simp*]: $- a \leq 0 \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *neg-0-le-iff-le* [*simp*]: $0 \leq - a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *neg-less-iff-less* [*simp*]: $- b < - a \longleftrightarrow a < b$
⟨*proof*⟩

lemma *neg-less-0-iff-less* [*simp*]: $- a < 0 \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *neg-0-less-iff-less* [*simp*]: $0 < - a \longleftrightarrow a < 0$
⟨*proof*⟩

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \longleftrightarrow b < - a$
⟨*proof*⟩

lemma *minus-less-iff*: $- a < b \longleftrightarrow - b < a$
⟨*proof*⟩

lemma *le-minus-iff*: $a \leq - b \longleftrightarrow b \leq - a$
⟨*proof*⟩

lemma *minus-le-iff*: $- a \leq b \longleftrightarrow - b \leq a$

⟨*proof*⟩

lemma *less-iff-diff-less-0*: $a < b \longleftrightarrow a - b < 0$
⟨*proof*⟩

lemma *diff-less-eq*[*algebra-simps*, *field-simps*]: $a - b < c \longleftrightarrow a < c + b$
⟨*proof*⟩

lemma *less-diff-eq*[*algebra-simps*, *field-simps*]: $a < c - b \longleftrightarrow a + b < c$
⟨*proof*⟩

lemma *diff-le-eq*[*algebra-simps*, *field-simps*]: $a - b \leq c \longleftrightarrow a \leq c + b$
⟨*proof*⟩

lemma *le-diff-eq*[*algebra-simps*, *field-simps*]: $a \leq c - b \longleftrightarrow a + b \leq c$
⟨*proof*⟩

lemma *le-iff-diff-le-0*: $a \leq b \longleftrightarrow a - b \leq 0$
⟨*proof*⟩

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* ⟨*proof*⟩

subclass *ordered-ab-semigroup-add-imp-le*
⟨*proof*⟩

end

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

subclass *linordered-cancel-ab-semigroup-add* ⟨*proof*⟩

lemma *neg-less-eq-nonneg* [*simp*]:
  $- a \leq a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *neg-less-nonneg* [*simp*]:
  $- a < a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *less-eq-neg-nonpos* [*simp*]:
$a \leq - a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *equal-neg-zero* [*simp*]:
$a = - a \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *neg-equal-zero* [*simp*]:
$- a = a \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *double-zero* [*simp*]:
$a + a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *double-zero-sym* [*simp*]:
$0 = a + a \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]:
$0 < a + a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]:
$0 \leq a + a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *double-add-less-zero-iff-single-add-less-zero* [*simp*]:
$a + a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]:
$a + a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *le-minus-self-iff*:
$a \leq - a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *minus-le-self-iff*:
$- a \leq a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *minus-max-eq-min*:
$- max\ x\ y = min\ (-x)\ (-y)$
⟨*proof*⟩

lemma *minus-min-eq-max*:

$- \ min\ x\ y = max\ (-x)\ (-y)$
⟨*proof*⟩

end

context *ordered-comm-monoid-add*
begin

lemma *add-increasing*:
$0 \leq a \Longrightarrow b \leq c \Longrightarrow b \leq a + c$
⟨*proof*⟩

lemma *add-increasing2*:
$0 \leq c \Longrightarrow b \leq a \Longrightarrow b \leq a + c$
⟨*proof*⟩

lemma *add-strict-increasing*:
$0 < a \Longrightarrow b \leq c \Longrightarrow b < a + c$
⟨*proof*⟩

lemma *add-strict-increasing2*:
$0 \leq a \Longrightarrow b < c \Longrightarrow b < a + c$
⟨*proof*⟩

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*]: $sgn\ 0 = 0$

⟨*proof*⟩

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*: $- |a| \leq 0$
  ⟨*proof*⟩

lemma *abs-of-nonneg* [*simp*]:
  assumes *nonneg*: $0 \leq a$ **shows** $|a| = a$
⟨*proof*⟩

lemma *abs-idempotent* [*simp*]: $||a|| = |a|$
⟨*proof*⟩

lemma *abs-eq-0* [*simp*]: $|a| = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *abs-zero* [*simp*]: $|0| = 0$
⟨*proof*⟩

lemma *abs-0-eq* [*simp*, *no-atp*]: $0 = |a| \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *abs-le-zero-iff* [*simp*]: $|a| \leq 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-less-abs-iff* [*simp*]: $0 < |a| \longleftrightarrow a \neq 0$
⟨*proof*⟩

lemma *abs-not-less-zero* [*simp*]: $\neg\ |a| < 0$
⟨*proof*⟩

lemma *abs-ge-minus-self*: $- a \leq |a|$
⟨*proof*⟩

lemma *abs-minus-commute*:
  $|a - b| = |b - a|$
⟨*proof*⟩

lemma *abs-of-pos*: $0 < a \Longrightarrow |a| = a$
⟨*proof*⟩

lemma *abs-of-nonpos* [*simp*]:
assumes $a \leq 0$ **shows** $|a| = - a$
⟨*proof*⟩

lemma *abs-of-neg*: $a < 0 \Longrightarrow |a| = - a$
⟨*proof*⟩

lemma *abs-le-D1*: $|a| \leq b \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *abs-le-D2*: $|a| \leq b \Longrightarrow - a \leq b$
⟨*proof*⟩

lemma *abs-le-iff*: $|a| \leq b \longleftrightarrow a \leq b \wedge - a \leq b$
⟨*proof*⟩

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$
⟨*proof*⟩

lemma *abs-triangle-ineq2-sym*: $|a| - |b| \leq |b - a|$
⟨*proof*⟩

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$
⟨*proof*⟩

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$
⟨*proof*⟩

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$
⟨*proof*⟩

lemma *abs-add-abs* [*simp*]:
$||a| + |b|| = |a| + |b|$ (**is** *?L = ?R*)
⟨*proof*⟩

end

Needed for abelian cancellation simprocs:

lemma *add-cancel-21*: $((x::'a::ab\text{-}group\text{-}add) + (y + z) = y + u) = (x + z = u)$
⟨*proof*⟩

lemma *add-cancel-end*: $(x + (y + z) = y) = (x = - (z::'a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *less-eqI*: $(x::'a::ordered\text{-}ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x < y) = (x' < y')$
⟨*proof*⟩

lemma *le-eqI*: $(x::'a::ordered\text{-}ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (y <= x) = (y' <= x')$
⟨*proof*⟩

lemma *eq-eqI*: $(x::'a::ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x = y) = (x' = y')$
⟨*proof*⟩

lemma *diff-def*: $(x::'a::ab\text{-}group\text{-}add) - y == x + (-y)$
⟨*proof*⟩

lemma *le-add-right-mono*:
assumes
$a <= b + (c::'a::ordered\text{-}ab\text{-}group\text{-}add)$
$c <= d$
shows $a <= b + d$
⟨*proof*⟩

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 $i \le j \wedge k \le l \Longrightarrow i + k \le j + l$
and $i = j \wedge k \le l \Longrightarrow i + k \le j + l$
and $i \le j \wedge k = l \Longrightarrow i + k \le j + l$
and $i = j \wedge k = l \Longrightarrow i + k = j + l$
⟨*proof*⟩

lemma *add-mono-thms-linordered-field* [*no-atp*]:
fixes $i\ j\ k :: 'a::ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add$
shows $i < j \wedge k = l \Longrightarrow i + k < j + l$
and $i = j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k \le l \Longrightarrow i + k < j + l$
and $i \le j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k < l \Longrightarrow i + k < j + l$
⟨*proof*⟩

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*⟩

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*]:
  $f\ a\ (f\ a\ b) = f\ a\ b$
  ⟨*proof*⟩

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*⟩

context *ab-semigroup-idem-mult*
begin

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

end

5.3 Concrete lattices

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

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*
⟨*proof*⟩

end

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

5.3.1 Intro and elim rules

context *semilattice-inf*
begin

lemma *le-infI1*:
  a ⊑ *x* ⟹ *a* ⊓ *b* ⊑ *x*
  ⟨*proof*⟩

lemma *le-infI2*:
  b ⊑ *x* ⟹ *a* ⊓ *b* ⊑ *x*
  ⟨*proof*⟩

lemma *le-infI*: *x* ⊑ *a* ⟹ *x* ⊑ *b* ⟹ *x* ⊑ *a* ⊓ *b*
  ⟨*proof*⟩

lemma *le-infE*: *x* ⊑ *a* ⊓ *b* ⟹ (*x* ⊑ *a* ⟹ *x* ⊑ *b* ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

lemma *le-inf-iff* [*simp*]:
  x ⊑ *y* ⊓ *z* ⟷ *x* ⊑ *y* ∧ *x* ⊑ *z*
  ⟨*proof*⟩

lemma *le-iff-inf*:
  x ⊑ *y* ⟷ *x* ⊓ *y* = *x*
  ⟨*proof*⟩

lemma *inf-mono*: $a \sqsubseteq c \Longrightarrow b \leq d \Longrightarrow a \sqcap b \sqsubseteq c \sqcap d$
⟨*proof*⟩

lemma *mono-inf*:
fixes $f :: 'a \Rightarrow 'b$::*semilattice-inf*
shows *mono* $f \Longrightarrow f\ (A \sqcap B) \sqsubseteq f\ A \sqcap f\ B$
⟨*proof*⟩

end

context *semilattice-sup*
begin

lemma *le-supI1*:
$x \sqsubseteq a \Longrightarrow x \sqsubseteq a \sqcup b$
⟨*proof*⟩

lemma *le-supI2*:
$x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcup b$
⟨*proof*⟩

lemma *le-supI*:
$a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow a \sqcup b \sqsubseteq x$
⟨*proof*⟩

lemma *le-supE*:
$a \sqcup b \sqsubseteq x \Longrightarrow (a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *le-sup-iff* [*simp*]:
$x \sqcup y \sqsubseteq z \longleftrightarrow x \sqsubseteq z \wedge y \sqsubseteq z$
⟨*proof*⟩

lemma *le-iff-sup*:
$x \sqsubseteq y \longleftrightarrow x \sqcup y = y$
⟨*proof*⟩

lemma *sup-mono*: $a \sqsubseteq c \Longrightarrow b \leq d \Longrightarrow a \sqcup b \sqsubseteq c \sqcup d$
⟨*proof*⟩

lemma *mono-sup*:
fixes $f :: 'a \Rightarrow 'b$::*semilattice-sup*
shows *mono* $f \Longrightarrow f\ A \sqcup f\ B \sqsubseteq f\ (A \sqcup B)$
⟨*proof*⟩

end

5.3.2 Equational laws

sublocale *semilattice-inf* < *inf*!: *semilattice inf*
⟨*proof*⟩

context *semilattice-inf*
begin

lemma *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
⟨*proof*⟩

lemma *inf-commute*: $(x \sqcap y) = (y \sqcap x)$
⟨*proof*⟩

lemma *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
⟨*proof*⟩

lemma *inf-idem*: $x \sqcap x = x$
⟨*proof*⟩

lemma *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$
⟨*proof*⟩

lemma *inf-absorb1*: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
⟨*proof*⟩

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

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

end

sublocale *semilattice-sup* < *sup*!: *semilattice sup*
⟨*proof*⟩

context *semilattice-sup*
begin

lemma *sup-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
⟨*proof*⟩

lemma *sup-commute*: $(x \sqcup y) = (y \sqcup x)$
⟨*proof*⟩

lemma *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
⟨*proof*⟩

lemma *sup-idem*: $x \sqcup x = x$
⟨*proof*⟩

lemma *sup-left-idem*: $x \sqcup (x \sqcup y) = x \sqcup y$
⟨*proof*⟩

lemma *sup-absorb1*: $y \sqsubseteq x \Longrightarrow x \sqcup y = x$
⟨*proof*⟩

lemma *sup-absorb2*: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
⟨*proof*⟩

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

end

context *lattice*
begin

lemma *dual-lattice*:
class.lattice (*op* ≥) (*op* >) *sup inf*
⟨*proof*⟩

lemma *inf-sup-absorb*: $x \sqcap (x \sqcup y) = x$
⟨*proof*⟩

lemma *sup-inf-absorb*: $x \sqcup (x \sqcap y) = x$
⟨*proof*⟩

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*: $x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap (x \sqcup z)$
⟨*proof*⟩

lemma *distrib-inf-le*: $(x \sqcap y) \sqcup (x \sqcap z) \sqsubseteq x \sqcap (y \sqcup z)$
⟨*proof*⟩

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*⟩

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*⟩

end

5.3.3 Strict order

context *semilattice-inf*
begin

lemma *less-infI1*:
$a \sqsubset x \Longrightarrow a \sqcap b \sqsubset x$
⟨*proof*⟩

lemma *less-infI2*:
$b \sqsubset x \Longrightarrow a \sqcap b \sqsubset x$
⟨*proof*⟩

end

context *semilattice-sup*
begin

lemma *less-supI1*:
$x \sqsubset a \Longrightarrow x \sqsubset a \sqcup b$
⟨*proof*⟩

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

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*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
⟨*proof*⟩

lemma *inf-sup-distrib1*:
$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
⟨*proof*⟩

lemma *inf-sup-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
⟨*proof*⟩

lemma *dual-distrib-lattice*:
class.distrib-lattice $(op \geq)$ $(op >)$ *sup inf*
⟨*proof*⟩

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*]:
$\bot \sqcap x = \bot$
⟨*proof*⟩

lemma *inf-bot-right* [*simp*]:
$x \sqcap \bot = \bot$
⟨*proof*⟩

lemma *sup-bot-left* [*simp*]:
$\bot \sqcup x = x$
⟨*proof*⟩

lemma *sup-bot-right* [*simp*]:
$x \sqcup \bot = x$
⟨*proof*⟩

lemma *sup-eq-bot-iff* [*simp*]:
$x \sqcup y = \bot \longleftrightarrow x = \bot \wedge y = \bot$
⟨*proof*⟩

end

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

lemma *sup-top-left* [*simp*]:
$\top \sqcup x = \top$
⟨*proof*⟩

lemma *sup-top-right* [*simp*]:
$x \sqcup \top = \top$
⟨*proof*⟩

lemma *inf-top-left* [*simp*]:
$\top \sqcap x = x$
⟨*proof*⟩

lemma *inf-top-right* [*simp*]:
$x \sqcap \top = x$
⟨*proof*⟩

lemma *inf-eq-top-iff* [*simp*]:
$x \sqcap y = \top \longleftrightarrow x = \top \wedge y = \top$
⟨*proof*⟩

end

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

lemma *dual-bounded-lattice*:
class.bounded-lattice (*op* ≥) (*op* >) (*op* ⊔) (*op* ⊓) ⊤ ⊥
⟨*proof*⟩

end

class *boolean-algebra* = *distrib-lattice* + *bounded-lattice* + *minus* + *uminus* +
assumes *inf-compl-bot*: $x \sqcap - x = \bot$
and *sup-compl-top*: $x \sqcup - x = \top$
assumes *diff-eq*: $x - y = x \sqcap - y$
begin

lemma *dual-boolean-algebra*:
class.boolean-algebra (λ*x y*. *x* ⊔ − *y*) *uminus* (*op* ≥) (*op* >) (*op* ⊔) (*op* ⊓) ⊤ ⊥
⟨*proof*⟩

lemma *compl-inf-bot*:
$- x \sqcap x = \bot$
⟨*proof*⟩

lemma *compl-sup-top*:
$- x \sqcup x = \top$
⟨*proof*⟩

lemma *compl-unique*:
assumes $x \sqcap y = \bot$
and $x \sqcup y = \top$
shows $- x = y$

⟨*proof*⟩

lemma *double-compl* [*simp*]:
$- (- x) = x$
⟨*proof*⟩

lemma *compl-eq-compl-iff* [*simp*]:
$- x = - y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *compl-bot-eq* [*simp*]:
$- \bot = \top$
⟨*proof*⟩

lemma *compl-top-eq* [*simp*]:
$- \top = \bot$
⟨*proof*⟩

lemma *compl-inf* [*simp*]:
$- (x \sqcap y) = - x \sqcup - y$
⟨*proof*⟩

lemma *compl-sup* [*simp*]:
$- (x \sqcup y) = - x \sqcap - y$
⟨*proof*⟩

lemma *compl-mono*:
$x \sqsubseteq y \Longrightarrow - y \sqsubseteq - x$
⟨*proof*⟩

lemma *compl-le-compl-iff*:
$- x \leq - y \longleftrightarrow y \leq x$
⟨*proof*⟩

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*⟩

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 $x \sqcup y = x \bigtriangledown y$
⟨*proof*⟩

5.7 *min*/*max* on linear orders as special case of *op* ⊓/*op* ⊔

sublocale *linorder* < *min-max*!: *distrib-lattice less-eq less min max*
⟨*proof*⟩

lemma *inf-min*: *inf* = (*min* :: 'a::{*semilattice-inf*, *linorder*} ⇒ 'a ⇒ 'a)
⟨*proof*⟩

lemma *sup-max*: *sup* = (*max* :: 'a::{*semilattice-sup*, *linorder*} ⇒ 'a ⇒ 'a)
⟨*proof*⟩

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: $A - B \longleftrightarrow A \wedge \neg B$

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

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

instance ⟨*proof*⟩

end

lemma *sup-boolI1*:
$P \Longrightarrow P \sqcup Q$
⟨*proof*⟩

lemma *sup-boolI2*:
$Q \Longrightarrow P \sqcup Q$

⟨*proof*⟩

lemma *sup-boolE*:
$P \sqcup Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

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*⟩

end

instance *fun* :: (*type*, *distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

instance *fun* :: (*type*, *bounded-lattice*) *bounded-lattice* ⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

instance *fun* :: (*type*, *boolean-algebra*) *boolean-algebra*
⟨*proof*⟩

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 == ~ (*x : A*) — non-membership

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

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

not-mem ((-/ ∉ -) [*50*, *51*] *50*)

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

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*)
⟨*proof*⟩

lemma *Collect-mem-eq* [*simp*]: {*x*. *x*:*A*} = *A*
⟨*proof*⟩

lemma *CollectI*: *P*(*a*) ==> *a* : {*x*. *P*(*x*)}
⟨*proof*⟩

lemma *CollectD*: *a* : {*x*. *P*(*x*)} ==> *P*(*a*)
⟨*proof*⟩

lemma *Collect-cong*: (!!*x*. *P x* = *Q x*) ==> {*x*. *P*(*x*)} = {*x*. *Q*(*x*)}
⟨*proof*⟩

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

⟨*ML*⟩

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

Set enumerations

abbreviation *empty* :: *'a set* ({}) **where**
{} ≡ *bot*

definition *insert* :: *'a* ⇒ *'a set* ⇒ *'a set* **where**
insert-compr: *insert a B* = {*x*. *x* = *a* ∨ *x* ∈ *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 ≡ *less*

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

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

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

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

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

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

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

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*

⟨*ML*⟩

Translate between {*e* | *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*{- |/-./ -}))

⟨*ML*⟩

lemma *ballI* [*intro*!]: (!!*x*. *x*:*A* ==> *P x*) ==> *ALL x*:*A*. *P x*
⟨*proof*⟩

lemmas *strip* = *impI allI ballI*

lemma *bspec* [*dest?*]: *ALL x*:*A*. *P x* ==> *x*:*A* ==> *P x*
⟨*proof*⟩

Gives better instantiation for bound:

⟨*ML*⟩

lemma *ballE* [*elim*]: *ALL x:A. P x ==> (P x ==> Q) ==> (x ~: A ==> Q) ==> Q*
⟨*proof*⟩

lemma *bexI* [*intro*]: *P x ==> x:A ==> EX x:A. P x*
— Normally the best argument order: *P x* constrains the choice of $x \in A$.
⟨*proof*⟩

lemma *rev-bexI* [*intro?*]: *x:A ==> P x ==> EX x:A. P x*
— The best argument order when there is only one $x \in A$.
⟨*proof*⟩

lemma *bexCI*: *(ALL x:A. ~P x ==> P a) ==> a:A ==> EX x:A. P x*
⟨*proof*⟩

lemma *bexE* [*elim!*]: *EX x:A. P x ==> (!!x. x:A ==> P x ==> Q) ==> Q*
⟨*proof*⟩

lemma *ball-triv* [*simp*]: *(ALL x:A. P) = ((EX x. x:A) --> P)*
— Trival rewrite rule.
⟨*proof*⟩

lemma *bex-triv* [*simp*]: *(EX x:A. P) = ((EX x. x:A) & P)*
— Dual form for existentials.
⟨*proof*⟩

lemma *bex-triv-one-point1* [*simp*]: *(EX x:A. x = a) = (a:A)*
⟨*proof*⟩

lemma *bex-triv-one-point2* [*simp*]: *(EX x:A. a = x) = (a:A)*
⟨*proof*⟩

lemma *bex-one-point1* [*simp*]: *(EX x:A. x = a & P x) = (a:A & P a)*
⟨*proof*⟩

lemma *bex-one-point2* [*simp*]: *(EX x:A. a = x & P x) = (a:A & P a)*
⟨*proof*⟩

lemma *ball-one-point1* [*simp*]: *(ALL x:A. x = a --> P x) = (a:A --> P a)*
⟨*proof*⟩

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

Congruence rules

lemma *ball-cong*:
A = B ==> (!!x. x:B ==> P x = Q x) ==>
(ALL x:A. P x) = (ALL x:B. Q x)

⟨*proof*⟩

lemma *strong-ball-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow P\ x = Q\ x) \Longrightarrow$
$(ALL\ x{:}A.\ P\ x) = (ALL\ x{:}B.\ Q\ x)$
⟨*proof*⟩

lemma *bex-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow P\ x = Q\ x) \Longrightarrow$
$(EX\ x{:}A.\ P\ x) = (EX\ x{:}B.\ Q\ x)$
⟨*proof*⟩

lemma *strong-bex-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow P\ x = Q\ x) \Longrightarrow$
$(EX\ x{:}A.\ P\ x) = (EX\ x{:}B.\ Q\ x)$
⟨*proof*⟩

6.3 Basic operations

6.3.1 Subsets

lemma *subsetI* [*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
⟨*proof*⟩

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

lemma *subsetD* [*elim*, *intro?*]: $A \subseteq B \Longrightarrow c \in A \Longrightarrow c \in B$
⟨*proof*⟩

lemma *rev-subsetD* [*no-atp*,*intro?*]: $c \in A \Longrightarrow A \subseteq B \Longrightarrow c \in B$
— The same, with reversed premises for use with *erule* – cf *rev-mp*.
⟨*proof*⟩

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

lemma *subsetCE* [*no-atp*,*elim*]: $A \subseteq B \Longrightarrow (c \notin A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$
— Classical elimination rule.
⟨*proof*⟩

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

lemma *contra-subsetD* [*no-atp*]: $A \subseteq B \Longrightarrow c \notin B \Longrightarrow c \notin A$
⟨*proof*⟩

lemma *subset-refl* [*simp*]: $A \subseteq A$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *set-rev-mp*: x:A ==> $A \subseteq B$ ==> x:B
⟨*proof*⟩

lemma *set-mp*: $A \subseteq B$ ==> x:A ==> x:B
⟨*proof*⟩

lemma *eq-mem-trans*: a=b ==> $b \in A$ ==> $a \in A$
⟨*proof*⟩

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** $A = B$
⟨*proof*⟩

lemma *expand-set-eq*: ($A = B$) = (*ALL* x. (x:A) = (x:B))
⟨*proof*⟩

lemma *subset-antisym* [*intro*!]: $A \subseteq B$ ==> $B \subseteq A$ ==> $A = B$
— Anti-symmetry of the subset relation.
⟨*proof*⟩

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

lemma *equalityD1*: $A = B$ ==> $A \subseteq B$
⟨*proof*⟩

lemma *equalityD2*: $A = B$ ==> $B \subseteq A$
⟨*proof*⟩

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

lemma *equalityE*: $A = B$ ==> ($A \subseteq B$ ==> $B \subseteq A$ ==> P) ==> P
⟨*proof*⟩

lemma *equalityCE* [*elim*]:
$A = B$ ==> ($c \in A$ ==> $c \in B$ ==> P) ==> ($c \notin A$ ==> $c \notin B$ ==> P)
==> P
⟨*proof*⟩

lemma *eqset-imp-iff*: $A = B$ ==> (x : A) = (x : B)
⟨*proof*⟩

lemma *eqelem-imp-iff*: $x = y$ ==> (x : A) = (y : A)
⟨*proof*⟩

6.3.3 The universal set – UNIV

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

lemma *UNIV-def*:
UNIV = {*x*. *True*}
⟨*proof*⟩

lemma *UNIV-I* [*simp*]: *x* : *UNIV*
⟨*proof*⟩

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

lemma *UNIV-witness* [*intro?*]: *EX x*. *x* : *UNIV*
⟨*proof*⟩

lemma *subset-UNIV* [*simp*]: *A* ⊆ *UNIV*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bex-UNIV* [*simp*]: *Bex UNIV P* = *Ex P*
⟨*proof*⟩

lemma *UNIV-eq-I*: (⋀*x*. *x* ∈ *A*) ⟹ *UNIV* = *A*
⟨*proof*⟩

6.3.4 The empty set

lemma *empty-def*:
{} = {*x*. *False*}
⟨*proof*⟩

lemma *empty-iff* [*simp*]: (*c* : {}) = *False*
⟨*proof*⟩

lemma *emptyE* [*elim!*]: *a* : {} ==> *P*
⟨*proof*⟩

lemma *empty-subsetI* [*iff*]: {} ⊆ *A*
— One effect is to delete the ASSUMPTION {} ⊆ *A*
⟨*proof*⟩

lemma *equals0I*: (!!*y*. *y* ∈ *A* ==> *False*) ==> *A* = {}
⟨*proof*⟩

lemma *equals0D*: $A = \{\} \Longrightarrow a \notin A$
— Use for reasoning about disjointness: $A\ Int\ B = \{\}$
⟨*proof*⟩

lemma *ball-empty* [*simp*]: $Ball\ \{\}\ P = True$
⟨*proof*⟩

lemma *bex-empty* [*simp*]: $Bex\ \{\}\ P = False$
⟨*proof*⟩

lemma *UNIV-not-empty* [*iff*]: $UNIV \neq \{\}$
⟨*proof*⟩

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*]: $(A \in Pow\ B) = (A \subseteq B)$
⟨*proof*⟩

lemma *PowI*: $A \subseteq B \Longrightarrow A \in Pow\ B$
⟨*proof*⟩

lemma *PowD*: $A \in Pow\ B \Longrightarrow A \subseteq B$
⟨*proof*⟩

lemma *Pow-bottom*: $\{\} \in Pow\ B$
⟨*proof*⟩

lemma *Pow-top*: $A \in Pow\ A$
⟨*proof*⟩

6.3.6 Set complement

lemma *Compl-iff* [*simp*]: $(c \in -A) = (c \notin A)$
⟨*proof*⟩

lemma *ComplI* [*intro!*]: $(c \in A \Longrightarrow False) \Longrightarrow c \in -A$
⟨*proof*⟩

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!*]: $c : -A \Longrightarrow c\sim:A$
⟨*proof*⟩

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

lemma *Compl-eq*: $- A = \{x.\ \sim x : A\}$ ⟨*proof*⟩

6.3.7 Binary union – Un

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

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

notation (*HTML* **output**)
union (**infixl** ∪ *65*)

lemma *Un-def*:
$A \cup B = \{x.\ x \in A \vee x \in B\}$
⟨*proof*⟩

lemma *Un-iff* [*simp*]: $(c : A\ Un\ B) = (c{:}A \mid c{:}B)$
⟨*proof*⟩

lemma *UnI1* [*elim?*]: $c{:}A \Longrightarrow c : A\ Un\ B$
⟨*proof*⟩

lemma *UnI2* [*elim?*]: $c{:}B \Longrightarrow c : A\ Un\ B$
⟨*proof*⟩

Classical introduction rule: no commitment to A vs B.

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

lemma *UnE* [*elim*!]: $c : A\ Un\ B \Longrightarrow (c{:}A \Longrightarrow P) \Longrightarrow (c{:}B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *insert-def*: $insert\ a\ B = \{x.\ x = a\} \cup B$
⟨*proof*⟩

lemma *mono-Un*: $mono\ f \Longrightarrow f\ A \cup f\ B \subseteq f\ (A \cup B)$
⟨*proof*⟩

6.3.8 Binary intersection – Int

abbreviation *inter* :: *'a set* ⇒ *'a set* ⇒ *'a set* (**infixl** *Int 70*) **where**
op Int ≡ *inf*

notation (*xsymbols*)
inter (**infixl** ∩ *70*)

notation (*HTML* **output**)
inter (**infixl** ∩ *70*)

lemma *Int-def*:
$A \cap B = \{x.\ x \in A \wedge x \in B\}$
⟨*proof*⟩

lemma *Int-iff* [*simp*]: $(c : A\ Int\ B) = (c{:}A\ \&\ c{:}B)$
⟨*proof*⟩

lemma *IntI* [*intro*!]: $c{:}A \Longrightarrow c{:}B \Longrightarrow c : A\ Int\ B$
⟨*proof*⟩

lemma *IntD1*: $c : A\ Int\ B \Longrightarrow c{:}A$
⟨*proof*⟩

lemma *IntD2*: $c : A\ Int\ B \Longrightarrow c{:}B$
⟨*proof*⟩

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

lemma *mono-Int*: $mono\ f \Longrightarrow f\ (A \cap B) \subseteq f\ A \cap f\ B$
⟨*proof*⟩

6.3.9 Set difference

lemma *Diff-iff* [*simp*]: $(c : A - B) = (c{:}A\ \&\ c{\sim}{:}B)$
⟨*proof*⟩

lemma *DiffI* [*intro*!]: $c : A \Longrightarrow c\ {\sim}{:}\ B \Longrightarrow c : A - B$
⟨*proof*⟩

lemma *DiffD1*: $c : A - B \Longrightarrow c : A$
⟨*proof*⟩

lemma *DiffD2*: $c : A - B \Longrightarrow c : B \Longrightarrow P$
⟨*proof*⟩

lemma *DiffE* [*elim*!]: $c : A - B \Longrightarrow (c{:}A \Longrightarrow c{\sim}{:}B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *set-diff-eq*: $A - B = \{x.\ x : A\ \&\ {\sim}\ x : B\}$ ⟨*proof*⟩

lemma *Compl-eq-Diff-UNIV*: $-A = (UNIV - A)$
⟨*proof*⟩

6.3.10 Augmenting a set − *insert*

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

⟨*proof*⟩

lemma *insertI1*: $a : \mathit{insert}\ a\ B$
⟨*proof*⟩

lemma *insertI2*: $a : B \Longrightarrow a : \mathit{insert}\ b\ B$
⟨*proof*⟩

lemma *insertE* [*elim*!]: $a : \mathit{insert}\ b\ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a{:}A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

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

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

lemma *set-insert*:
assumes $x \in A$
obtains B **where** $A = \mathit{insert}\ x\ B$ **and** $x \notin B$
⟨*proof*⟩

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

6.3.11 Singletons, using insert

lemma *singletonI* [*intro*!,*no-atp*]: $a : \{a\}$
— Redundant? But unlike *insertCI*, it proves the subgoal immediately!
⟨*proof*⟩

lemma *singletonD* [*dest*!,*no-atp*]: $b : \{a\} \Longrightarrow b = a$
⟨*proof*⟩

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

lemma *singleton-iff*: $(b : \{a\}) = (b = a)$
⟨*proof*⟩

lemma *singleton-inject* [*dest*!]: $\{a\} = \{b\} \Longrightarrow a = b$
⟨*proof*⟩

lemma *singleton-insert-inj-eq* [*iff*,*no-atp*]:
$(\{b\} = \mathit{insert}\ a\ A) = (a = b\ \&\ A \subseteq \{b\})$
⟨*proof*⟩

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

lemma *subset-singletonD*: *A* ⊆ {*x*} ==> *A* = {} | *A* = {*x*}
⟨*proof*⟩

lemma *singleton-conv* [*simp*]: {*x*. *x* = *a*} = {*a*}
⟨*proof*⟩

lemma *singleton-conv2* [*simp*]: {*x*. *a* = *x*} = {*a*}
⟨*proof*⟩

lemma *diff-single-insert*: *A* − {*x*} ⊆ *B* ==> *x* ∈ *A* ==> *A* ⊆ *insert x B*
⟨*proof*⟩

lemma *doubleton-eq-iff*: ({*a*,*b*} = {*c*,*d*}) = (*a*=*c* & *b*=*d* | *a*=*d* & *b*=*c*)
⟨*proof*⟩

6.3.12 Image of a set under a function

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

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

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

lemma *image-eqI* [*simp*, *intro*]: *b* = *f x* ==> *x*:*A* ==> *b* : *f*'*A*
⟨*proof*⟩

lemma *imageI*: *x* : *A* ==> *f x* : *f* ' *A*
⟨*proof*⟩

lemma *rev-image-eqI*: *x*:*A* ==> *b* = *f x* ==> *b* : *f*'*A*
— This version's more effective when we already have the required *x*.
⟨*proof*⟩

lemma *imageE* [*elim*!]:
b : (%*x*. *f x*)'*A* ==> (!!*x*. *b* = *f x* ==> *x*:*A* ==> *P*) ==> *P*
— The eta-expansion gives variable-name preservation.
⟨*proof*⟩

lemma *image-Un*: *f*'(*A Un B*) = *f*'*A Un f*'*B*
⟨*proof*⟩

lemma *image-iff*: (*z* : *f*'*A*) = (*EX x*:*A*. *z* = *f x*)

⟨*proof*⟩

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.
⟨*proof*⟩

lemma *subset-image-iff*: $(B \subseteq f`A) = (EX\ AA.\ AA \subseteq A\ \&\ B = f`AA)$
⟨*proof*⟩

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.
⟨*proof*⟩

Range of a function – just a translation for image!

lemma *range-eqI*: $b = f\ x \Longrightarrow b \in range\ f$
⟨*proof*⟩

lemma *rangeI*: $f\ x \in range\ f$
⟨*proof*⟩

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

6.3.13 Some rules with *if*

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

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

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

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

lemma *split-if-eq1*: $((if\ Q\ then\ x\ else\ y) = b) = ((Q \dashrightarrow x = b)\ \&\ (\sim Q \dashrightarrow y = b))$
⟨*proof*⟩

lemma *split-if-eq2*: $(a = (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a = x)\ \&\ (\sim Q \dashrightarrow a = y))$
⟨*proof*⟩

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

lemma *split-if-mem1*: $((if\ Q\ then\ x\ else\ y) : b) = ((Q \dashrightarrow x : b)\ \&\ (\sim Q \dashrightarrow y : b))$
⟨*proof*⟩

lemma *split-if-mem2*: $(a : (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y)) = ((Q \dashrightarrow a : x)\ \&\ (\sim Q \dashrightarrow a : y))$
⟨*proof*⟩

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*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
⟨*proof*⟩

lemma *psubsetE* [*elim*!,*no-atp*]:
$[|A \subset B;\ [|A \subseteq B;\ \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
⟨*proof*⟩

lemma *psubset-insert-iff*:
$(A \subset \mathit{insert}\ x\ B) = (\mathit{if}\ x \in B\ \mathit{then}\ A \subset B\ \mathit{else}\ \mathit{if}\ x \in A\ \mathit{then}\ A - \{x\} \subset B\ \mathit{else}\ A \subseteq B)$
⟨*proof*⟩

lemma *psubset-eq*: $(A \subset B) = (A \subseteq B\ \&\ A \neq B)$
⟨*proof*⟩

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$
⟨*proof*⟩

lemma *psubset-trans*: $[|\ A \subset B;\ B \subset C\ |] \Longrightarrow A \subset C$
⟨*proof*⟩

lemma *psubsetD*: $[|\ A \subset B;\ c \in A\ |] \Longrightarrow c \in B$
⟨*proof*⟩

lemma *psubset-subset-trans*: $A \subset B \Longrightarrow B \subseteq C \Longrightarrow A \subset C$
⟨*proof*⟩

lemma *subset-psubset-trans*: $A \subseteq B \Longrightarrow B \subset C \Longrightarrow A \subset C$
⟨*proof*⟩

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

lemma *atomize-ball*:
$(!!x.\ x \in A \Longrightarrow P\ x) == \mathit{Trueprop}\ (\forall x \in A.\ P\ x)$
⟨*proof*⟩

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

6.4.2 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq \mathit{insert}\ a\ B$
⟨*proof*⟩

lemma *subset-insertI2*: $A \subseteq B \Longrightarrow A \subseteq \mathit{insert}\ b\ B$
⟨*proof*⟩

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

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
⟨*proof*⟩

lemma *Un-upper2*: $B \subseteq A \cup B$
⟨*proof*⟩

lemma *Un-least*: $A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow A \cup B \subseteq C$
⟨*proof*⟩

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
⟨*proof*⟩

lemma *Int-lower2*: $A \cap B \subseteq B$
⟨*proof*⟩

lemma *Int-greatest*: $C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B$
⟨*proof*⟩

Set difference.

lemma *Diff-subset*: $A - B \subseteq A$
⟨*proof*⟩

lemma *Diff-subset-conv*: $(A - B \subseteq C) = (A \subseteq B \cup C)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subset-empty* [*simp*]: $(A \subseteq \{\}) = (A = \{\})$
⟨*proof*⟩

lemma *not-psubset-empty* [*iff*]: $\neg (A < \{\})$
⟨*proof*⟩

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

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

lemma *Collect-neg-eq*: $\{x.\ \neg\ P\ x\} = -\ \{x.\ P\ x\}$
⟨*proof*⟩

lemma *Collect-disj-eq*: $\{x.\ P\ x \mid Q\ x\} = \{x.\ P\ x\} \cup \{x.\ Q\ x\}$
⟨*proof*⟩

lemma *Collect-imp-eq*: $\{x.\ P\ x \longrightarrow Q\ x\} = -\{x.\ P\ x\} \cup \{x.\ Q\ x\}$
⟨*proof*⟩

lemma *Collect-conj-eq*: $\{x.\ P\ x\ \&\ Q\ x\} = \{x.\ P\ x\} \cap \{x.\ Q\ x\}$
⟨*proof*⟩

insert.

lemma *insert-is-Un*: *insert a A* = {*a*} *Un A*
— NOT SUITABLE FOR REWRITING since {*a*} == *insert a* {}
⟨*proof*⟩

lemma *insert-not-empty* [*simp*]: *insert a A* ≠ {}
⟨*proof*⟩

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

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

lemma *insert-absorb2* [*simp*]: *insert x* (*insert x A*) = *insert x A*
⟨*proof*⟩

lemma *insert-commute*: *insert x* (*insert y A*) = *insert y* (*insert x A*)
⟨*proof*⟩

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

lemma *mk-disjoint-insert*: $a \in A \Longrightarrow \exists B.\ A = insert\ a\ B\ \&\ a \notin B$
— use new B rather than $A - \{a\}$ to avoid infinite unfolding
⟨*proof*⟩

lemma *insert-Collect*: $insert\ a\ (Collect\ P) = \{u.\ u \neq a \longrightarrow P\ u\}$
⟨*proof*⟩

lemma *insert-inter-insert*[*simp*]: $insert\ a\ A \cap insert\ a\ B = insert\ a\ (A \cap B)$
⟨*proof*⟩

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

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

image.

lemma *image-empty* [*simp*]: $f`\{\} = \{\}$
⟨*proof*⟩

lemma *image-insert* [*simp*]: $f\ `\ insert\ a\ B = insert\ (f\ a)\ (f`B)$
⟨*proof*⟩

lemma *image-constant*: $x \in A \Longrightarrow (\lambda x.\ c)\ `\ A = \{c\}$
⟨*proof*⟩

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

lemma *image-image*: $f\ `\ (g\ `\ A) = (\lambda x.\ f\ (g\ x))\ `\ A$
⟨*proof*⟩

lemma *insert-image* [*simp*]: $x \in A \Longrightarrow insert\ (f\ x)\ (f`A) = f`A$
⟨*proof*⟩

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

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

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.
⟨*proof*⟩

lemma *if-image-distrib* [*simp*]:
$(\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ f\ x\ \mathit{else}\ g\ x)\ `\ S$
$= (f\ `\ (S \cap \{x.\ P\ x\})) \cup (g\ `\ (S \cap \{x.\ \neg\ P\ x\}))$
⟨*proof*⟩

lemma *image-cong*: $M = N \Longrightarrow (!!x.\ x \in N \Longrightarrow f\ x = g\ x) \Longrightarrow f`M = g`N$
⟨*proof*⟩

range.

lemma *full-SetCompr-eq* [*no-atp*]: $\{u.\ \exists x.\ u = f\ x\} = \mathit{range}\ f$
⟨*proof*⟩

lemma *range-composition*: $\mathit{range}\ (\lambda x.\ f\ (g\ x)) = f`\mathit{range}\ g$
⟨*proof*⟩

Int

lemma *Int-absorb* [*simp*]: $A \cap A = A$
⟨*proof*⟩

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$
⟨*proof*⟩

lemma *Int-commute*: $A \cap B = B \cap A$
⟨*proof*⟩

lemma *Int-left-commute*: $A \cap (B \cap C) = B \cap (A \cap C)$
⟨*proof*⟩

lemma *Int-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
⟨*proof*⟩

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

lemma *Int-absorb1*: $B \subseteq A \Longrightarrow A \cap B = B$
⟨*proof*⟩

lemma *Int-absorb2*: $A \subseteq B \Longrightarrow A \cap B = A$
⟨*proof*⟩

lemma *Int-empty-left* [*simp*]: $\{\} \cap B = \{\}$
⟨*proof*⟩

lemma *Int-empty-right* [*simp*]: $A \cap \{\} = \{\}$
⟨*proof*⟩

lemma *disjoint-eq-subset-Compl*: $(A \cap B = \{\}) = (A \subseteq -B)$
⟨*proof*⟩

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

lemma *Int-UNIV-left* [*simp*]: $UNIV \cap B = B$
⟨*proof*⟩

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

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
⟨*proof*⟩

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
⟨*proof*⟩

lemma *Int-UNIV* [*simp*,*no-atp*]: $(A \cap B = UNIV) = (A = UNIV\ \&\ B = UNIV)$
⟨*proof*⟩

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

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

Un.

lemma *Un-absorb* [*simp*]: $A \cup A = A$
⟨*proof*⟩

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
⟨*proof*⟩

lemma *Un-commute*: $A \cup B = B \cup A$
⟨*proof*⟩

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
⟨*proof*⟩

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
⟨*proof*⟩

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

lemma *Un-absorb1*: $A \subseteq B \Longrightarrow A \cup B = B$
⟨*proof*⟩

lemma *Un-absorb2*: $B \subseteq A ==> A \cup B = A$
⟨*proof*⟩

lemma *Un-empty-left* [*simp*]: $\{\} \cup B = B$
⟨*proof*⟩

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

lemma *Un-UNIV-left* [*simp*]: $UNIV \cup B = UNIV$
⟨*proof*⟩

lemma *Un-UNIV-right* [*simp*]: $A \cup UNIV = UNIV$
⟨*proof*⟩

lemma *Un-insert-left* [*simp*]: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
⟨*proof*⟩

lemma *Un-insert-right* [*simp*]: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
⟨*proof*⟩

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

lemma *Int-insert-left-if0*[*simp*]:
$a \notin C \Longrightarrow (insert\ a\ B)\ Int\ C = B \cap C$
⟨*proof*⟩

lemma *Int-insert-left-if1*[*simp*]:
$a \in C \Longrightarrow (insert\ a\ B)\ Int\ C = insert\ a\ (B\ Int\ C)$
⟨*proof*⟩

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

lemma *Int-insert-right-if0*[*simp*]:
$a \notin A \Longrightarrow A\ Int\ (insert\ a\ B) = A\ Int\ B$
⟨*proof*⟩

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

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
⟨*proof*⟩

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
⟨*proof*⟩

lemma *Un-Int-crazy*:
$(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
⟨*proof*⟩

lemma *subset-Un-eq*: $(A \subseteq B) = (A \cup B = B)$
⟨*proof*⟩

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

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

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
⟨*proof*⟩

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
⟨*proof*⟩

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap -A = \{\}$
⟨*proof*⟩

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
⟨*proof*⟩

lemma *Compl-partition*: $A \cup -A = UNIV$
⟨*proof*⟩

lemma *Compl-partition2*: $-A \cup A = UNIV$
⟨*proof*⟩

lemma *double-complement* [*simp*]: $-(-A) = (A::'a\ set)$
⟨*proof*⟩

lemma *Compl-Un* [*simp*]: $-(A \cup B) = (-A) \cap (-B)$
⟨*proof*⟩

lemma *Compl-Int* [*simp*]: $-(A \cap B) = (-A) \cup (-B)$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *Compl-UNIV-eq* [*simp*]: $-UNIV = \{\}$
⟨*proof*⟩

lemma *Compl-empty-eq* [*simp*]: $-\{\} = UNIV$
⟨*proof*⟩

lemma *Compl-subset-Compl-iff* [*iff*]: $(-A \subseteq -B) = (B \subseteq A)$
⟨*proof*⟩

lemma *Compl-eq-Compl-iff* [*iff*]: $(-A = -B) = (A = (B::'a\ set))$
⟨*proof*⟩

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*: $(\forall x \in A \cup B.\ P\ x) = ((\forall x{\in}A.\ P\ x)\ \&\ (\forall x{\in}B.\ P\ x))$
⟨*proof*⟩

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

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
⟨*proof*⟩

lemma *Diff-eq-empty-iff* [*simp*,*no-atp*]: $(A - B = \{\}) = (A \subseteq B)$
⟨*proof*⟩

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
⟨*proof*⟩

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - (B::'a\ set)$
⟨*proof*⟩

lemma *Diff-triv*: $A \cap B = \{\} \Longrightarrow A - B = A$
⟨*proof*⟩

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

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

lemma *Diff-UNIV* [*simp*]: $A - UNIV = \{\}$
⟨*proof*⟩

lemma *Diff-insert0* [*simp*,*no-atp*]: $x \notin A \Longrightarrow A - \mathit{insert}\ x\ B = A - B$
⟨*proof*⟩

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

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

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

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

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

lemma *insert-Diff*: $a \in A \Longrightarrow \mathit{insert}\ a\ (A - \{a\}) = A$
⟨*proof*⟩

lemma *Diff-insert-absorb*: $x \notin A \Longrightarrow (\mathit{insert}\ x\ A) - \{x\} = A$
⟨*proof*⟩

lemma *Diff-disjoint* [*simp*]: $A \cap (B - A) = \{\}$
⟨*proof*⟩

lemma *Diff-partition*: $A \subseteq B \Longrightarrow A \cup (B - A) = B$
⟨*proof*⟩

lemma *double-diff*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow B - (C - A) = A$
⟨*proof*⟩

lemma *Un-Diff-cancel* [*simp*]: $A \cup (B - A) = A \cup B$
⟨*proof*⟩

lemma *Un-Diff-cancel2* [*simp*]: $(B - A) \cup A = B \cup A$
⟨*proof*⟩

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *Int-Diff*: $(A \cap B) - C = A \cap (B - C)$
⟨*proof*⟩

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$
⟨*proof*⟩

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
⟨*proof*⟩

lemma *Diff-Compl* [*simp*]: $A - (- B) = A \cap B$
⟨*proof*⟩

lemma *Compl-Diff-eq* [*simp*]: $- (A - B) = -A \cup B$
⟨*proof*⟩

Quantification over type *bool*.

lemma *bool-induct*: $P\ True \Longrightarrow P\ False \Longrightarrow P\ x$
⟨*proof*⟩

lemma *all-bool-eq*: $(\forall b.\ P\ b) \longleftrightarrow P\ True \wedge P\ False$
⟨*proof*⟩

lemma *bool-contrapos*: $P\ x \Longrightarrow \neg\ P\ False \Longrightarrow P\ True$
⟨*proof*⟩

lemma *ex-bool-eq*: $(\exists b.\ P\ b) \longleftrightarrow P\ True \vee P\ False$
⟨*proof*⟩

Pow

lemma *Pow-empty* [*simp*]: $Pow\ \{\} = \{\{\}\}$
⟨*proof*⟩

lemma *Pow-insert*: $Pow\ (insert\ a\ A) = Pow\ A \cup (insert\ a\ `\ Pow\ A)$
⟨*proof*⟩

lemma *Pow-Compl*: $Pow\ (-\ A) = \{-B \mid B.\ A \in Pow\ B\}$
⟨*proof*⟩

lemma *Pow-UNIV* [*simp*]: $Pow\ UNIV = UNIV$
⟨*proof*⟩

lemma *Un-Pow-subset*: $Pow\ A \cup Pow\ B \subseteq Pow\ (A \cup B)$
⟨*proof*⟩

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

Miscellany.

lemma *set-eq-subset*: $(A = B) = (A \subseteq B \ \& \ B \subseteq A)$
⟨*proof*⟩

lemma *subset-iff*: $(A \subseteq B) = (\forall t.\ t \in A \longrightarrow t \in B)$
⟨*proof*⟩

lemma *subset-iff-psubset-eq*: $(A \subseteq B) = ((A \subset B) \mid (A = B))$
⟨*proof*⟩

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

lemma *ex-in-conv*: $(\exists x.\ x \in A) = (A \neq \{\})$
⟨*proof*⟩

lemma *distinct-lemma*: $f\ x \neq f\ y \Longrightarrow x \neq y$
⟨*proof*⟩

6.4.4 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \Longrightarrow f`A \subseteq f`B$
⟨*proof*⟩

lemma *Pow-mono*: $A \subseteq B \Longrightarrow Pow\ A \subseteq Pow\ B$
⟨*proof*⟩

lemma *insert-mono*: $C \subseteq D \Longrightarrow insert\ a\ C \subseteq insert\ a\ D$
⟨*proof*⟩

lemma *Un-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cup B \subseteq C \cup D$
⟨*proof*⟩

lemma *Int-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cap B \subseteq C \cap D$
⟨*proof*⟩

lemma *Diff-mono*: $A \subseteq C \Longrightarrow D \subseteq B \Longrightarrow A - B \subseteq C - D$
⟨*proof*⟩

lemma *Compl-anti-mono*: $A \subseteq B \Longrightarrow -B \subseteq -A$
⟨*proof*⟩

Monotonicity of implications.

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

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

⟨*proof*⟩

lemma *disj-mono*: *P1* −−> *Q1* ==> *P2* −−> *Q2* ==> (*P1* | *P2*) −−> (*Q1* | *Q2*)
⟨*proof*⟩

lemma *imp-mono*: *Q1* −−> *P1* ==> *P2* −−> *Q2* ==> (*P1* −−> *P2*) −−> (*Q1* −−> *Q2*)
⟨*proof*⟩

lemma *imp-refl*: *P* −−> *P* ⟨*proof*⟩

lemma *not-mono*: *Q* −−> *P* ==> ~ *P* −−> ~ *Q*
⟨*proof*⟩

lemma *ex-mono*: (!!*x*. *P x* −−> *Q x*) ==> (*EX x*. *P x*) −−> (*EX x*. *Q x*)
⟨*proof*⟩

lemma *all-mono*: (!!*x*. *P x* −−> *Q x*) ==> (*ALL x*. *P x*) −−> (*ALL x*. *Q x*)
⟨*proof*⟩

lemma *Collect-mono*: (!!*x*. *P x* −−> *Q x*) ==> *Collect P* ⊆ *Collect Q*
⟨*proof*⟩

lemma *Int-Collect-mono*:
A ⊆ *B* ==> (!!*x*. *x* ∈ *A* ==> *P x* −−> *Q x*) ==> *A* ∩ *Collect P* ⊆ *B* ∩ *Collect Q*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vimage-singleton-eq*: (*a* : *f* −‘ {*b*}) = (*f a* = *b*)
⟨*proof*⟩

lemma *vimageI* [*intro*]: *f a* = *b* ==> *b*:*B* ==> *a* : *f* −‘ *B*
⟨*proof*⟩

lemma *vimageI2*: $f\ a : A \Longrightarrow a : f\ -`\ A$
⟨*proof*⟩

lemma *vimageE* [*elim*!]: a: $f\ -`\ B \Longrightarrow (!!x.\ f\ a = x \Longrightarrow x{:}B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *vimageD*: $a : f\ -`\ A \Longrightarrow f\ a : A$
⟨*proof*⟩

lemma *vimage-empty* [*simp*]: $f\ -`\ \{\} = \{\}$
⟨*proof*⟩

lemma *vimage-Compl*: $f\ -`\ (-A) = -(f\ -`\ A)$
⟨*proof*⟩

lemma *vimage-Un* [*simp*]: $f\ -`\ (A\ Un\ B) = (f\ -`\ A)\ Un\ (f\ -`\ B)$
⟨*proof*⟩

lemma *vimage-Int* [*simp*]: $f\ -`\ (A\ Int\ B) = (f\ -`\ A)\ Int\ (f\ -`\ B)$
⟨*proof*⟩

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

lemma *vimage-Collect*: $(!!x.\ P\ (f\ x) = Q\ x) \Longrightarrow f\ -`\ (Collect\ P) = Collect\ Q$
⟨*proof*⟩

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

lemma *vimage-Diff*: $f\ -`\ (A - B) = (f\ -`\ A) - (f\ -`\ B)$
⟨*proof*⟩

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

lemma *vimage-mono*: $A \subseteq B \Longrightarrow f\ -`\ A \subseteq f\ -`\ B$
— monotonicity
⟨*proof*⟩

lemma *vimage-image-eq* [*no-atp*]: $f\ -`\ (f\ `\ A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
⟨*proof*⟩

lemma *image-vimage-subset*: $f\ `\ (f\ -`\ A) <= A$
⟨*proof*⟩

lemma *image-vimage-eq* [*simp*]: f ‘ $(f -‘ A) = A$ *Int range* f
⟨*proof*⟩

lemma *vimage-const* [*simp*]: $((\lambda x.\ c) -‘ A) = (\textit{if } c \in A \textit{ then } \textit{UNIV else } \{\})$
⟨*proof*⟩

lemma *vimage-if* [*simp*]: $((\lambda x.\ \textit{if } x \in B \textit{ then } c \textit{ else } d) -‘ A) =$
$(\textit{if } c \in A \textit{ then } (\textit{if } d \in A \textit{ then UNIV else } B)$
$\textit{else if } d \in A \textit{ then } -B \textit{ else } \{\})$
⟨*proof*⟩

lemma *vimage-inter-cong*:
$(\bigwedge w.\ w \in S \Longrightarrow f\ w = g\ w) \Longrightarrow f -‘ y \cap S = g -‘ y \cap S$
⟨*proof*⟩

lemma *image-Int-subset*: $f‘(A\ Int\ B) <= f‘A\ Int\ f‘B$
⟨*proof*⟩

lemma *image-diff-subset*: $f‘A - f‘B <= f‘(A - B)$
⟨*proof*⟩

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*]: $contents\ \{x\} = x$
⟨*proof*⟩

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
⟨*proof*⟩

6.5 Misc

Rudimentary code generation

lemma *insert-code* [*code*]: $insert\ y\ A\ x \longleftrightarrow y = x \vee A\ x$
⟨*proof*⟩

lemma *vimage-code* [*code*]: $(f -‘ A)\ x = A\ (f\ x)$
⟨*proof*⟩

Misc theorem and ML bindings

lemmas *equalityI* = *subset-antisym*

⟨*ML*⟩

end

7 Typedef: HOL type definitions

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

⟨*ML*⟩

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*⟩

lemma *Abs-inject*:
assumes *x*: *x* ∈ *A* **and** *y*: *y* ∈ *A*
shows (*Abs x* = *Abs y*) = (*x* = *y*)
⟨*proof*⟩

lemma *Rep-cases* [*cases set*]:
assumes *y*: *y* ∈ *A*
and *hyp*: !!*x*. *y* = *Rep x* ==> *P*
shows *P*
⟨*proof*⟩

lemma *Abs-cases* [*cases type*]:
assumes *r*: !!*y*. *x* = *Abs y* ==> *y* ∈ *A* ==> *P*
shows *P*
⟨*proof*⟩

lemma *Rep-induct* [*induct set*]:
assumes *y*: *y* ∈ *A*
and *hyp*: !!*x*. *P* (*Rep x*)
shows *P y*

⟨*proof*⟩

lemma *Abs-induct* [*induct type*]:
  assumes *r*: !!*y*. *y* ∈ *A* ==> *P* (*Abs y*)
  shows *P x*
⟨*proof*⟩

lemma *Rep-range*: *range Rep* = *A*
⟨*proof*⟩

lemma *Abs-image*: *Abs* ‘ *A* = *UNIV*
⟨*proof*⟩

end

⟨*ML*⟩

end

8 Complete-Lattice: Complete lattices, with special focus on sets

theory *Complete-Lattice*
imports *Set*
begin

notation
  less-eq (**infix** ⊑ *50*) **and**
  less (**infix** ⊏ *50*) **and**
  inf (**infixl** ⊓ *70*) **and**
  sup (**infixl** ⊔ *65*) **and**
  top (⊤) **and**
  bot (⊥)

8.1 Syntactic infimum and supremum operations

class *Inf* =
  fixes *Inf* :: *'a set* ⇒ *'a* (⨅- [*900*] *900*)

class *Sup* =
  fixes *Sup* :: *'a set* ⇒ *'a* (⨆- [*900*] *900*)

8.2 Abstract complete lattices

class *complete-lattice* = *bounded-lattice* + *Inf* + *Sup* +
  assumes *Inf-lower*: *x* ∈ *A* ⟹ ⨅*A* ⊑ *x*
    and *Inf-greatest*: (⋀*x*. *x* ∈ *A* ⟹ *z* ⊑ *x*) ⟹ *z* ⊑ ⨅*A*
  assumes *Sup-upper*: *x* ∈ *A* ⟹ *x* ⊑ ⨆*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 $(op \geq)$ $(op >)$ $(op \sqcup)$ $(op \sqcap)$ $\top$ $\bot$
⟨*proof*⟩

lemma *Inf-Sup*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \sqsubseteq a\}$
⟨*proof*⟩

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \sqsubseteq b\}$
⟨*proof*⟩

lemma *Inf-empty*:
$\bigsqcap \{\} = \top$
⟨*proof*⟩

lemma *Sup-empty*:
$\bigsqcup \{\} = \bot$
⟨*proof*⟩

lemma *Inf-insert*: $\bigsqcap insert\ a\ A = a \sqcap \bigsqcap A$
⟨*proof*⟩

lemma *Sup-insert*: $\bigsqcup insert\ a\ A = a \sqcup \bigsqcup A$
⟨*proof*⟩

lemma *Inf-singleton* [*simp*]:
$\bigsqcap \{a\} = a$
⟨*proof*⟩

lemma *Sup-singleton* [*simp*]:
$\bigsqcup \{a\} = a$
⟨*proof*⟩

lemma *Inf-binary*:
$\bigsqcap \{a,\ b\} = a \sqcap b$
⟨*proof*⟩

lemma *Sup-binary*:
$\bigsqcup \{a,\ b\} = a \sqcup b$
⟨*proof*⟩

lemma *Inf-UNIV*:
$\bigsqcap UNIV = bot$
⟨*proof*⟩

lemma *Sup-UNIV*:
$\bigsqcup UNIV = top$

⟨*proof*⟩

lemma *Sup-le-iff*: *Sup A* ⊑ *b* ⟷ (∀ *a*∈*A*. *a* ⊑ *b*)
⟨*proof*⟩

lemma *le-Inf-iff*: *b* ⊑ *Inf A* ⟷ (∀ *a*∈*A*. *b* ⊑ *a*)
⟨*proof*⟩

definition *SUPR* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
SUPR A f = ⨆ (*f* ‘ *A*)

definition *INFI* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
INFI A f = ⨅ (*f* ‘ *A*)

end

syntax
-SUP1 :: *pttrns* => *'b* => *'b* (*3SUP -./ -*) [*0*, *10*] *10*)
-SUP :: *pttrn* => *'a set* => *'b* => *'b* (*3SUP -:-./ -*) [*0*, *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*)

⟨*ML*⟩

context *complete-lattice*
begin

lemma *le-SUPI*: *i* : *A* ⟹ *M i* ⊑ (*SUP i*:*A. M i*)
⟨*proof*⟩

lemma *SUP-leI*: (⋀*i. i* : *A* ⟹ *M i* ⊑ *u*) ⟹ (*SUP i*:*A. M i*) ⊑ *u*
⟨*proof*⟩

lemma *INF-leI*: *i* : *A* ⟹ (*INF i*:*A. M i*) ⊑ *M i*
⟨*proof*⟩

lemma *le-INFI*: (⋀*i. i* : *A* ⟹ *u* ⊑ *M i*) ⟹ *u* ⊑ (*INF i*:*A. M i*)
⟨*proof*⟩

lemma *SUP-le-iff*: $(SUP\ i{:}A.\ M\ i) \sqsubseteq u \longleftrightarrow (\forall\, i \in A.\ M\ i \sqsubseteq u)$
⟨*proof*⟩

lemma *le-INF-iff*: $u \sqsubseteq (INF\ i{:}A.\ M\ i) \longleftrightarrow (\forall\, i \in A.\ u \sqsubseteq M\ i)$
⟨*proof*⟩

lemma *SUP-const*[*simp*]: $A \neq \{\} \Longrightarrow (SUP\ i{:}A.\ M) = M$
⟨*proof*⟩

lemma *INF-const*[*simp*]: $A \neq \{\} \Longrightarrow (INF\ i{:}A.\ M) = M$
⟨*proof*⟩

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*⟩

end

lemma *Inf-empty-bool* [*simp*]:
$\bigsqcap \{\}$
⟨*proof*⟩

lemma *not-Sup-empty-bool* [*simp*]:
$\neg \bigsqcup \{\}$
⟨*proof*⟩

lemma *INFI-bool-eq*:
INFI = *Ball*
⟨*proof*⟩

lemma *SUPR-bool-eq*:
SUPR = *Bex*
⟨*proof*⟩

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*⟩

end

lemma *Inf-empty-fun*:
$\bigsqcap \{\} = (\lambda\text{-}.\ \bigsqcap \{\})$
⟨*proof*⟩

lemma *Sup-empty-fun*:
$\bigsqcup \{\} = (\lambda\text{-}.\ \bigsqcup \{\})$
⟨*proof*⟩

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*⟩

lemma *Union-iff* [*simp*, *no-atp*]:
$A \in \bigcup C \longleftrightarrow (\exists X{\in}C.\ A{\in}X)$
⟨*proof*⟩

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.
⟨*proof*⟩

lemma *UnionE* [*elim*!]:
$A \in \bigcup C \Longrightarrow (\bigwedge X.\ A{\in}X \Longrightarrow X{\in}C \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *Union-upper*: $B \in A$ ==> $B \subseteq$ *Union A*
⟨*proof*⟩

lemma *Union-least*: (!!$X.\ X \in A$ ==> $X \subseteq C$) ==> *Union A* $\subseteq C$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *Union-empty* [*simp*]: *Union*({}) = {}
⟨*proof*⟩

lemma *Union-UNIV* [*simp*]: *Union UNIV* = *UNIV*
⟨*proof*⟩

lemma *Union-insert* [*simp*]: *Union* (*insert a B*) = *a* ∪ ⋃ *B*
⟨*proof*⟩

lemma *Union-Un-distrib* [*simp*]: ⋃(*A Un B*) = ⋃ *A* ∪ ⋃ *B*
⟨*proof*⟩

lemma *Union-Int-subset*: ⋃(*A* ∩ *B*) ⊆ ⋃ *A* ∩ ⋃ *B*
⟨*proof*⟩

lemma *Union-empty-conv* [*simp*,*no-atp*]: (⋃ *A* = {}) = (∀ *x*∈*A*. *x* = {})
⟨*proof*⟩

lemma *empty-Union-conv* [*simp*,*no-atp*]: ({} = ⋃ *A*) = (∀ *x*∈*A*. *x* = {})
⟨*proof*⟩

lemma *Union-disjoint*: (⋃ *C* ∩ *A* = {}) = (∀ *B*∈*C*. *B* ∩ *A* = {})
⟨*proof*⟩

lemma *subset-Pow-Union*: *A* ⊆ *Pow* (⋃ *A*)
⟨*proof*⟩

lemma *Union-Pow-eq* [*simp*]: ⋃(*Pow A*) = *A*
⟨*proof*⟩

lemma *Union-mono*: *A* ⊆ *B* ==> ⋃ *A* ⊆ ⋃ *B*
⟨*proof*⟩

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.

⟨*ML*⟩

lemma *UNION-eq-Union-image*:
(⋃*x*∈*A. B x*) = ⋃(*B*‘*A*)
⟨*proof*⟩

lemma *Union-def*:
⋃*S* = (⋃*x*∈*S. x*)
⟨*proof*⟩

lemma *UNION-def* [*no-atp*]:
(⋃*x*∈*A. B x*) = {*y*. ∃*x*∈*A. y* ∈ *B x*}
⟨*proof*⟩

lemma *Union-image-eq* [*simp*]:
⋃(*B*‘*A*) = (⋃*x*∈*A. B x*)
⟨*proof*⟩

lemma *UN-iff* [*simp*]: (*b*: (*UN x*:*A. B x*)) = (*EX x*:*A. b*: *B x*)
⟨*proof*⟩

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.
⟨*proof*⟩

lemma *UN-E* [*elim*!]: *b* : (*UN x*:*A. B x*) ==> (!!*x. x*:*A* ==> *b*: *B x* ==> *R*) ==> *R*
⟨*proof*⟩

lemma *UN-cong* [*cong*]:
A = *B* ==> (!!*x. x*:*B* ==> *C x* = *D x*) ==> (*UN x*:*A. C x*) = (*UN x*:*B. D x*)
⟨*proof*⟩

lemma *strong-UN-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B =simp\Rightarrow C\ x = D\ x) \Longrightarrow (UN\ x{:}A.\ C\ x) = (UN\ x{:}B.\ D\ x)$
⟨*proof*⟩

lemma *image-eq-UN*: $f`A = (UN\ x{:}A.\ \{f\ x\})$
⟨*proof*⟩

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

lemma *UN-least*: $(!!x.\ x \in A \Longrightarrow B\ x \subseteq C) \Longrightarrow (\bigcup x{\in}A.\ B\ x) \subseteq C$
⟨*proof*⟩

lemma *Collect-bex-eq* [*no-atp*]: $\{x.\ \exists y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
⟨*proof*⟩

lemma *UN-insert-distrib*: $u \in A \Longrightarrow (\bigcup x{\in}A.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *UN-empty* [*simp*,*no-atp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *UN-eq*: $(\bigcup x{\in}A.\ B\ x) = \bigcup(\{\,Y.\ \exists x{\in}A.\ Y = B\ x\})$
⟨*proof*⟩

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

lemma *Collect-ex-eq* [*no-atp*]: $\{x.\ \exists y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
⟨*proof*⟩

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

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

lemma *Un-eq-UN*: $A \cup B = (\bigcup b.\ \mathit{if}\ b\ \mathit{then}\ A\ \mathit{else}\ B)$
⟨*proof*⟩

lemma *UN-bool-eq*: $(\bigcup b{::}\mathit{bool}.\ A\ b) = (A\ \mathit{True} \cup A\ \mathit{False})$
⟨*proof*⟩

lemma *UN-Pow-subset*: $(\bigcup x{\in}A.\ \mathit{Pow}\ (B\ x)) \subseteq \mathit{Pow}\ (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *UN-mono*:
$A \subseteq B \Longrightarrow (!!x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcup x{\in}A.\ f\ x) \subseteq (\bigcup x{\in}B.\ g\ x)$
⟨*proof*⟩

lemma *vimage-Union*: $f\ -`\ (\mathit{Union}\ A) = (\mathit{UN}\ X{:}A.\ f\ -`\ X)$
⟨*proof*⟩

lemma *vimage-UN*: $f-`(\mathit{UN}\ x{:}A.\ B\ x) = (\mathit{UN}\ x{:}A.\ f\ -`\ B\ x)$
⟨*proof*⟩

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

lemma *image-UN*: $(f\ `\ (\mathit{UNION}\ A\ B)) = (\mathit{UN}\ x{:}A.(f\ `\ (B\ x)))$
⟨*proof*⟩

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*⟩

lemma *Inter-iff* [*simp*,*no-atp*]: (*A* : *Inter C*) = (*ALL X*:*C*. *A*:*X*)
⟨*proof*⟩

lemma *InterI* [*intro*!]: (!!*X*. *X*:*C* ==> *A*:*X*) ==> *A* : *Inter C*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *InterE* [*elim*]: *A* : *Inter C* ==> (*X*~:*C* ==> *R*) ==> (*A*:*X* ==> *R*) ==> *R*
— "Classical" elimination rule – does not require proving *X* ∈ *C*.
⟨*proof*⟩

lemma *Inter-lower*: *B* ∈ *A* ==> *Inter A* ⊆ *B*
⟨*proof*⟩

lemma *Inter-subset*:
[| !!*X*. *X* ∈ *A* ==> *X* ⊆ *B*; *A* ~= {} |] ==> ⋂ *A* ⊆ *B*
⟨*proof*⟩

lemma *Inter-greatest*: (!!*X*. *X* ∈ *A* ==> *C* ⊆ *X*) ==> *C* ⊆ *Inter A*
⟨*proof*⟩

lemma *Int-eq-Inter*: *A* ∩ *B* = ⋂{*A*, *B*}
⟨*proof*⟩

lemma *Inter-empty* [*simp*]: ⋂{} = *UNIV*
⟨*proof*⟩

lemma *Inter-UNIV* [*simp*]: ⋂ *UNIV* = {}
⟨*proof*⟩

lemma *Inter-insert* [*simp*]: ⋂(*insert a B*) = *a* ∩ ⋂ *B*
⟨*proof*⟩

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$
⟨*proof*⟩

lemma *Inter-Un-distrib*: $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$
⟨*proof*⟩

lemma *Inter-UNIV-conv* [*simp*,*no-atp*]:
$(\bigcap A = UNIV) = (\forall x{\in}A.\ x = UNIV)$
$(UNIV = \bigcap A) = (\forall x{\in}A.\ x = UNIV)$
⟨*proof*⟩

lemma *Inter-anti-mono*: $B \subseteq A \Longrightarrow \bigcap A \subseteq \bigcap B$
⟨*proof*⟩

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*)

⟨*ML*⟩

lemma *INTER-eq-Inter-image*:
$(\bigcap x{\in}A.\ B\ x) = \bigcap (B`A)$
⟨*proof*⟩

lemma *Inter-def*:
$\bigcap S = (\bigcap x{\in}S.\ x)$
⟨*proof*⟩

lemma *INTER-def*:
$(\bigcap x{\in}A.\ B\ x) = \{y.\ \forall x{\in}A.\ y \in B\ x\}$
⟨*proof*⟩

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

lemma *INT-iff* [*simp*]: $(b{:}\ (INT\ x{:}A.\ B\ x)) = (ALL\ x{:}A.\ b{:}\ B\ x)$
⟨*proof*⟩

lemma *INT-I* [*intro!*]: $(!!x.\ x{:}A \Longrightarrow b{:}\ B\ x) \Longrightarrow b : (INT\ x{:}A.\ B\ x)$
⟨*proof*⟩

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

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 $a \in A$.
⟨*proof*⟩

lemma *INT-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (INT\ x{:}A.\ C\ x) = (INT\ x{:}B.\ D\ x)$
⟨*proof*⟩

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

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

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

lemma *INT-greatest*: $(!!x.\ x \in A \Longrightarrow C \subseteq B\ x) \Longrightarrow C \subseteq (\bigcap x{\in}A.\ B\ x)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *INT-bool-eq*: $(\bigcap b{::}bool.\ A\ b) = (A\ True \cap A\ False)$
⟨*proof*⟩

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

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!
⟨*proof*⟩

lemma *vimage-INT*: $f{-}`(INT\ x{:}A.\ B\ x) = (INT\ x{:}A.\ f\ -`\ B\ x)$
⟨*proof*⟩

8.8 Distributive laws

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

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

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
⟨*proof*⟩

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
⟨*proof*⟩

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

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

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
⟨*proof*⟩

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.
⟨*proof*⟩

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

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

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

8.9 Complement

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

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

8.10 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

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

!!$A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Int\ B) = ((UN\ x{:}C.\ A\ x)\ Int\ B)$
!!$A\ B\ C.\ (UN\ x{:}C.\ A\ Int\ B\ x) = (A\ Int\ (UN\ x{:}C.\ B\ x))$
!!$A\ B\ C.\ (UN\ x{:}C.\ A\ x - B) = ((UN\ x{:}C.\ A\ x) - B)$
!!$A\ B\ C.\ (UN\ x{:}C.\ A - B\ x) = (A - (INT\ x{:}C.\ B\ x))$
!!$A\ B.\ (UN\ x{:}\ Union\ A.\ B\ x) = (UN\ y{:}A.\ UN\ x{:}y.\ B\ x)$
!!$A\ B\ C.\ (UN\ z{:}\ UNION\ A\ B.\ C\ z) = (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z)$
!!$A\ B\ f.\ (UN\ x{:}f`A.\ B\ x) = (UN\ a{:}A.\ B\ (f\ a))$
⟨*proof*⟩

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

lemma *ball-simps* [*simp*,*no-atp*]:
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ |\ Q) = ((ALL\ x{:}A.\ P\ x)\ |\ Q)$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ |\ Q\ x) = (P\ |\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ --{>}\ Q\ x) = (P\ --{>}\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ --{>}\ Q) = ((EX\ x{:}A.\ P\ x)\ --{>}\ Q)$
!!$P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
!!$P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
!!$a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
!!$A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$
!!$P\ Q.\ (ALL\ x{:}Collect\ Q.\ P\ x) = (ALL\ x.\ Q\ x\ --{>}\ P\ x)$
!!$A\ P\ f.\ (ALL\ x{:}f`A.\ P\ x) = (ALL\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(ALL\ x{:}A.\ P\ x)) = (EX\ x{:}A.\ \sim P\ x)$
⟨*proof*⟩

lemma *bex-simps* [*simp*,*no-atp*]:
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ x\ \&\ Q) = ((EX\ x{:}A.\ P\ x)\ \&\ Q)$
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x{:}A.\ Q\ x))$
!!$P.\ (EX\ x{:}\{\}.\ P\ x) = False$
!!$P.\ (EX\ x{:}UNIV.\ P\ x) = (EX\ x.\ P\ x)$
!!$a\ B\ P.\ (EX\ x{:}insert\ a\ B.\ P\ x) = (P(a)\ |\ (EX\ x{:}B.\ P\ x))$
!!$A\ P.\ (EX\ x{:}Union\ A.\ P\ x) = (EX\ y{:}A.\ EX\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (EX\ x{:}\ UNION\ A\ B.\ P\ x) = (EX\ a{:}A.\ EX\ x{:}B\ a.\ P\ x)$

$!!P\ Q.\ (EX\ x{:}Collect\ Q.\ P\ x) = (EX\ x.\ Q\ x\ \&\ P\ x)$
$!!A\ P\ f.\ (EX\ x{:}f`A.\ P\ x) = (EX\ x{:}A.\ P\ (f\ x))$
$!!A\ P.\ (\sim(EX\ x{:}A.\ P\ x)) = (ALL\ x{:}A.\ \sim P\ x)$
⟨*proof*⟩

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

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

Maxiscoping: pulling out big Unions and Intersections.

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

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

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 (⨆ - [*900*] *900*) **and**
top (⊤) **and**
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* ⇒ '*a*) ⇒ '*a* **where**
lfp f = *Inf* {*u*. *f u* ≤ *u*} — least fixed point

definition
gfp :: ('*a* ⇒ '*a*) ⇒ '*a* **where**
gfp f = *Sup* {*u*. *u* ≤ *f u*} — greatest fixed point

9.2 Proof of Knaster-Tarski Theorem using *lfp*

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

lemma *lfp-lowerbound*: *f A* ≤ *A* ==> *lfp f* ≤ *A*

⟨*proof*⟩

lemma *lfp-greatest*: (!!*u*. *f u* ≤ *u* ==> *A* ≤ *u*) ==> *A* ≤ *lfp f*
⟨*proof*⟩

end

lemma *lfp-lemma2*: *mono f* ==> *f* (*lfp f*) ≤ *lfp f*
⟨*proof*⟩

lemma *lfp-lemma3*: *mono f* ==> *lfp f* ≤ *f* (*lfp f*)
⟨*proof*⟩

lemma *lfp-unfold*: *mono f* ==> *lfp f* = *f* (*lfp f*)
⟨*proof*⟩

lemma *lfp-const*: *lfp* (λ*x*. *t*) = *t*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *lfp-ordinal-induct*:
fixes *f* :: 'a::*complete-lattice* ⇒ 'a
assumes *mono*: *mono f*
and *P-f*: ⋀*S*. *P S* ⟹ *P* (*f S*)
and *P-Union*: ⋀*M*. ∀ *S*∈*M*. *P S* ⟹ *P* (*Sup M*)
shows *P* (*lfp f*)
⟨*proof*⟩

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*)
⟨*proof*⟩

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

lemma *def-lfp-unfold*: [| h==*lfp*(f); *mono*(f) |] ==> $h = f(h)$
⟨*proof*⟩

lemma *def-lfp-induct*:
[| A == *lfp*(f); *mono*(f);
f (*inf* A P) ≤ P
|] ==> $A \leq P$
⟨*proof*⟩

lemma *def-lfp-induct-set*:
[| A == *lfp*(f); *mono*(f); a:A;
!!x. [| x: f(A *Int* {x. $P(x)$}) |] ==> $P(x)$
|] ==> $P(a)$
⟨*proof*⟩

lemma *lfp-mono*: (!!Z. f Z ≤ g Z) ==> *lfp* f ≤ *lfp* g
⟨*proof*⟩

9.4 Proof of Knaster-Tarski Theorem using *gfp*

gfp f is the greatest lower bound of the set {u. $u \leq f\ u$}

lemma *gfp-upperbound*: $X \leq f\ X$ ==> $X \leq$ *gfp* f
⟨*proof*⟩

lemma *gfp-least*: (!!u. $u \leq f\ u$ ==> $u \leq X$) ==> *gfp* $f \leq X$
⟨*proof*⟩

lemma *gfp-lemma2*: *mono* f ==> *gfp* $f \leq f$ (*gfp* f)
⟨*proof*⟩

lemma *gfp-lemma3*: *mono* f ==> f (*gfp* f) ≤ *gfp* f
⟨*proof*⟩

lemma *gfp-unfold*: *mono* f ==> *gfp* $f = f$ (*gfp* f)
⟨*proof*⟩

9.5 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: [| a: X; $X \subseteq f(X)$ |] ==> a : *gfp*(f)
⟨*proof*⟩

lemma *weak-coinduct-image*: !!X. [| a : X; g‘$X \subseteq f$ (g‘X) |] ==> $g\ a$: *gfp* f
⟨*proof*⟩

lemma *coinduct-lemma*:
[| $X \leq f$ (*sup* X (*gfp* f)); *mono* f |] ==> *sup* X (*gfp* f) ≤ f (*sup* X (*gfp* f))

⟨*proof*⟩

strong version, thanks to Coen and Frost

lemma *coinduct-set*: [| *mono*(*f*); *a*: *X*; *X* ⊆ *f*(*X Un gfp*(*f*)) |] ==> *a* : *gfp*(*f*)
⟨*proof*⟩

lemma *coinduct*: [| *mono*(*f*); *X* ≤ *f* (*sup X* (*gfp f*)) |] ==> *X* ≤ *gfp*(*f*)
⟨*proof*⟩

lemma *gfp-fun-UnI2*: [| *mono*(*f*); *a*: *gfp*(*f*) |] ==> *a*: *f*(*X Un gfp*(*f*))
⟨*proof*⟩

9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition *X* ⊆ *f X* to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: *mono*(*f*) ==> *mono*(%*x*. *f*(*x*) *Un X Un B*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *coinduct3*:
[| *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*))) |] ==> *a* : *gfp*(*f*)
⟨*proof*⟩

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

lemma *def-gfp-unfold*: [| *A*==*gfp*(*f*); *mono*(*f*) |] ==> *A* = *f*(*A*)
⟨*proof*⟩

lemma *def-coinduct*:
[| *A*==*gfp*(*f*); *mono*(*f*); *X* ≤ *f*(*sup X A*) |] ==> *X* ≤ *A*
⟨*proof*⟩

lemma *def-coinduct-set*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*X Un A*) |] ==> *a*: *A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *def-coinduct3*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un A*)) |] ==> *a*: *A*

⟨*proof*⟩

Monotonicity of *gfp*!

lemma *gfp-mono*: $(!!Z.\ f\ Z \leq g\ Z) ==> gfp\ f \leq gfp\ g$
⟨*proof*⟩

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

⟨*ML*⟩

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.

⟨*ML*⟩

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*)

⟨*ML*⟩

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*: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$
⟨*proof*⟩

lemma *apply-inverse*:
$f\ x = u \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow g\ (f\ x) = x) \Longrightarrow P\ x \Longrightarrow x = g\ u$
⟨*proof*⟩

10.1 The Identity Function *id*

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

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

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

lemma *image-id* [*simp*]: $id\ `\ Y = Y$
⟨*proof*⟩

lemma *vimage-ident* [*simp*]: $(\%x.\ x) -` Y = Y$
⟨*proof*⟩

lemma *vimage-id* [*simp*]: $id -` A = A$
⟨*proof*⟩

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*]: $(f\ o\ g)\ x = f\ (g\ x)$
⟨*proof*⟩

lemma *o-assoc*: $f\ o\ (g\ o\ h) = f\ o\ g\ o\ h$
⟨*proof*⟩

lemma *id-o* [*simp*]: $id\ o\ g = g$
⟨*proof*⟩

lemma *o-id* [*simp*]: $f\ o\ id = f$
⟨*proof*⟩

lemma *o-eq-dest*:
$a\ o\ b = c\ o\ d \Longrightarrow a\ (b\ v) = c\ (d\ v)$
⟨*proof*⟩

lemma *o-eq-elim*:
$a\ o\ b = c\ o\ d \Longrightarrow ((\bigwedge v.\ a\ (b\ v) = c\ (d\ v)) \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *image-compose*: $(f\ o\ g)\ `\ r = f`(g`r)$
⟨*proof*⟩

lemma *vimage-compose*: $(g \circ f)\ -`\ x = f\ -`\ (g\ -`\ x)$
⟨*proof*⟩

lemma *UN-o*: $UNION\ A\ (g\ o\ f) = UNION\ (f`A)\ g$
⟨*proof*⟩

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*: $(f\ o>\ g)\ x = g\ (f\ x)$
⟨*proof*⟩

lemma *fcomp-assoc*: $(f\ o>\ g)\ o>\ h = f\ o>\ (g\ o>\ h)$
⟨*proof*⟩

lemma *id-fcomp* [*simp*]: $id\ o>\ g = g$
⟨*proof*⟩

lemma *fcomp-id* [*simp*]: $f\ o>\ id = f$
⟨*proof*⟩

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 ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *inj f*
⟨*proof*⟩

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

lemma *datatype-injI*:
(!! *x*. *ALL y*. *f*(*x*) = *f*(*y*) −−> *x*=*y*) ==> *inj*(*f*)
⟨*proof*⟩

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

lemma *injD*: [| *inj*(*f*); *f*(*x*) = *f*(*y*) |] ==> *x*=*y*
⟨*proof*⟩

lemma *inj-on-eq-iff*: *inj-on f A* ==> *x*:*A* ==> *y*:*A* ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
⟨*proof*⟩

lemma *inj-eq*: *inj f* ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
⟨*proof*⟩

lemma *inj-on-id*[*simp*]: *inj-on id A*
⟨*proof*⟩

lemma *inj-on-id2*[*simp*]: *inj-on* (%*x*. *x*) *A*
⟨*proof*⟩

lemma *surj-id*[*simp*]: *surj id*
⟨*proof*⟩

lemma *bij-id*[*simp*]: *bij id*
⟨*proof*⟩

lemma *inj-onI*:
(!! *x y*. [| *x*:*A*; *y*:*A*; *f*(*x*) = *f*(*y*) |] ==> *x*=*y*) ==> *inj-on f A*
⟨*proof*⟩

lemma *inj-on-inverseI*: (!!*x*. *x*:*A* ==> *g*(*f*(*x*)) = *x*) ==> *inj-on f A*
⟨*proof*⟩

lemma *inj-onD*: [| *inj-on f A*; *f*(*x*)=*f*(*y*); *x*:*A*; *y*:*A* |] ==> *x*=*y*
⟨*proof*⟩

lemma *inj-on-iff*: [| *inj-on f A*; *x*:*A*; *y*:*A* |] ==> (*f*(*x*)=*f*(*y*)) = (*x*=*y*)
⟨*proof*⟩

lemma *comp-inj-on*:
[| *inj-on f A*; *inj-on g* (*f*‘*A*) |] ==> *inj-on* (*g o f*) *A*
⟨*proof*⟩

lemma *inj-on-imageI*: *inj-on* (*g o f*) *A* ⟹ *inj-on g* (*f* ‘ *A*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inj-on-contraD*: [| *inj-on f A*; ~*x*=*y*; *x*:*A*; *y*:*A* |] ==> ~ *f*(*x*)=*f*(*y*)
⟨*proof*⟩

lemma *inj-singleton*: *inj* (%*s*. {*s*})
⟨*proof*⟩

lemma *inj-on-empty*[*iff*]: *inj-on f* {}
⟨*proof*⟩

lemma *subset-inj-on*: [| *inj-on f B*; *A* <= *B* |] ==> *inj-on f A*
⟨*proof*⟩

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*) = {})
⟨*proof*⟩

lemma *inj-on-insert*[*iff*]:

inj-on f (insert a A) = (inj-on f A & f a ~: f`(A-{a}))
⟨*proof*⟩

lemma *inj-on-diff*: *inj-on f A ==> inj-on f (A-B)*
⟨*proof*⟩

lemma *surjI*: *(!! x. g(f x) = x) ==> surj g*
⟨*proof*⟩

lemma *surj-range*: *surj f ==> range f = UNIV*
⟨*proof*⟩

lemma *surjD*: *surj f ==> EX x. y = f x*
⟨*proof*⟩

lemma *surjE*: *surj f ==> (!!x. y = f x ==> C) ==> C*
⟨*proof*⟩

lemma *comp-surj*: *[| surj f; surj g |] ==> surj (g o f)*
⟨*proof*⟩

lemma *bijI*: *[| inj f; surj f |] ==> bij f*
⟨*proof*⟩

lemma *bij-is-inj*: *bij f ==> inj f*
⟨*proof*⟩

lemma *bij-is-surj*: *bij f ==> surj f*
⟨*proof*⟩

lemma *bij-betw-imp-inj-on*: *bij-betw f A B ⟹ inj-on f A*
⟨*proof*⟩

lemma *bij-comp*: *bij f ⟹ bij g ⟹ bij (g o f)*
⟨*proof*⟩

lemma *bij-betw-trans*:
bij-betw f A B ⟹ bij-betw g B C ⟹ bij-betw (g o f) A C
⟨*proof*⟩

lemma *bij-betw-inv*: **assumes** *bij-betw f A B* **shows** *EX g. bij-betw g B A*
⟨*proof*⟩

lemma *surj-image-vimage-eq*: *surj f ==> f ` (f -` A) = A*
⟨*proof*⟩

lemma *inj-vimage-image-eq*: *inj f ==> f -` (f ` A) = A*
⟨*proof*⟩

lemma *vimage-subsetD*: *surj f ==> f −‘ B <= A ==> B <= f ‘ A*
⟨*proof*⟩

lemma *vimage-subsetI*: *inj f ==> B <= f ‘ A ==> f −‘ B <= A*
⟨*proof*⟩

lemma *vimage-subset-eq*: *bij f ==> (f −‘ B <= A) = (B <= f ‘ A)*
⟨*proof*⟩

lemma *inj-on-Un-image-eq-iff*: *inj-on f (A ∪ B) ⟹ f ‘ A = f ‘ B ⟷ A = B*
⟨*proof*⟩

lemma *inj-on-image-Int*:
[| *inj-on f C*; *A<=C*; *B<=C* |] *==> f‘(A Int B) = f‘A Int f‘B*
⟨*proof*⟩

lemma *inj-on-image-set-diff*:
[| *inj-on f C*; *A<=C*; *B<=C* |] *==> f‘(A−B) = f‘A − f‘B*
⟨*proof*⟩

lemma *image-Int*: *inj f ==> f‘(A Int B) = f‘A Int f‘B*
⟨*proof*⟩

lemma *image-set-diff*: *inj f ==> f‘(A−B) = f‘A − f‘B*
⟨*proof*⟩

lemma *inj-image-mem-iff*: *inj f ==> (f a : f‘A) = (a : A)*
⟨*proof*⟩

lemma *inj-image-subset-iff*: *inj f ==> (f‘A <= f‘B) = (A<=B)*
⟨*proof*⟩

lemma *inj-image-eq-iff*: *inj f ==> (f‘A = f‘B) = (A = B)*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *bij-image-INT*: *bij f ==> f ‘ (INTER A B) = (INT x:A. f ‘ B x)*
⟨*proof*⟩

lemma *surj-Compl-image-subset*: *surj f ==> −(f‘A) <= f‘(−A)*
⟨*proof*⟩

lemma *inj-image-Compl-subset*: *inj f ==> f‘(−A) <= −(f‘A)*

⟨*proof*⟩

lemma *bij-image-Compl-eq*: *bij f* ==> *f*‘(−*A*) = −(*f*‘*A*)
⟨*proof*⟩

lemma (**in** *ordered-ab-group-add*) *inj-uminus*[*simp*, *intro*]: *inj-on uminus A*
⟨*proof*⟩

lemma (**in** *linorder*) *strict-mono-imp-inj-on*: *strict-mono f* ⟹ *inj-on f A*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fun-upd-same*: (*f*(*x*:=*y*)) *x* = *y*
⟨*proof*⟩

lemma *fun-upd-other*: *z*~=*x* ==> (*f*(*x*:=*y*)) *z* = *f z*

⟨*proof*⟩

lemma *fun-upd-upd* [*simp*]: $f(x{:=}y,x{:=}z) = f(x{:=}z)$
⟨*proof*⟩

lemma *fun-upd-twist*: $a \neq c \Longrightarrow (m(a{:=}b))(c{:=}d) = (m(c{:=}d))(a{:=}b)$
⟨*proof*⟩

lemma *inj-on-fun-updI*: $\llbracket$ *inj-on f A*; $y \notin f`A$ $\rrbracket \Longrightarrow$ *inj-on* $(f(x{:=}y))$ A
⟨*proof*⟩

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

lemma *fun-upd-comp*: $f \circ (g(x := y)) = (f \circ g)(x := f\ y)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *override-on-apply-notin*[*simp*]: $a \notin A \Longrightarrow$ (*override-on f g A*) *a* = *f a*
⟨*proof*⟩

lemma *override-on-apply-in*[*simp*]: $a \in A \Longrightarrow$ (*override-on f g A*) *a* = *g a*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *swap-commute*: *swap a b f* = *swap b a f*
⟨*proof*⟩

lemma *swap-nilpotent* [*simp*]: *swap a b* (*swap a b f*) = *f*
⟨*proof*⟩

lemma *swap-triple*:
assumes $a \neq c$ **and** $b \neq c$
shows $swap\ a\ b\ (swap\ b\ c\ (swap\ a\ b\ f)) = swap\ a\ c\ f$
⟨*proof*⟩

lemma *comp-swap*: $f \circ swap\ a\ b\ g = swap\ a\ b\ (f \circ g)$
⟨*proof*⟩

lemma *inj-on-imp-inj-on-swap*:
$[|inj\text{-}on\ f\ A;\ a \in A;\ b \in A|] ==> inj\text{-}on\ (swap\ a\ b\ f)\ A$
⟨*proof*⟩

lemma *inj-on-swap-iff* [*simp*]:
assumes *A*: $a \in A\ b \in A$ **shows** $inj\text{-}on\ (swap\ a\ b\ f)\ A = inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *surj-imp-surj-swap*: $surj\ f ==> surj\ (swap\ a\ b\ f)$
⟨*proof*⟩

lemma *surj-swap-iff* [*simp*]: $surj\ (swap\ a\ b\ f) = surj\ f$
⟨*proof*⟩

lemma *bij-swap-iff*: $bij\ (swap\ a\ b\ f) = bij\ f$
⟨*proof*⟩

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

10.8 Inversion of injective functions

definition *the-inv-into* :: $'a\ set => ('a => 'b) => ('b => 'a)$ **where**
$the\text{-}inv\text{-}into\ A\ f == \%x.\ THE\ y.\ y : A\ \&\ f\ y = x$

lemma *the-inv-into-f-f*:
$[|\ inj\text{-}on\ f\ A;\ \ x : A\ |] ==> the\text{-}inv\text{-}into\ A\ f\ (f\ x) = x$
⟨*proof*⟩

lemma *f-the-inv-into-f*:
$inj\text{-}on\ f\ A ==> y : f`A\ \ ==> f\ (the\text{-}inv\text{-}into\ A\ f\ y) = y$
⟨*proof*⟩

lemma *the-inv-into-into*:
$[|\ inj\text{-}on\ f\ A;\ x : f\ `\ A;\ A <= B\ |] ==> the\text{-}inv\text{-}into\ A\ f\ x : B$
⟨*proof*⟩

lemma *the-inv-into-onto*[*simp*]:
$inj\text{-}on\ f\ A ==> the\text{-}inv\text{-}into\ A\ f\ `\ (f\ `\ A) = A$
⟨*proof*⟩

lemma *the-inv-into-f-eq*:

[| *inj-on f A*; *f x* = *y*; *x* : *A* |] ==> *the-inv-into A f y* = *x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inj-on-the-inv-into*:
inj-on f A ⟹ *inj-on* (*the-inv-into A f*) (*f* ' *A*)
⟨*proof*⟩

lemma *bij-betw-the-inv-into*:
bij-betw f A B ⟹ *bij-betw* (*the-inv-into A f*) *B A*
⟨*proof*⟩

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* ⟨*proof*⟩

10.9 Proof tool setup

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

⟨*ML*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

definition

Unity :: *unit* (′(′))
where
() = *Abs-unit True*

lemma *unit-eq* [*no-atp*]: *u* = ()
⟨*proof*⟩

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

⟨*ML*⟩

rep-datatype () ⟨*proof*⟩

lemma *unit-all-eq1*: (!!*x*::*unit*. *PROP P x*) == *PROP P* ()
⟨*proof*⟩

lemma *unit-all-eq2*: (!!*x*::*unit*. *PROP P*) == *PROP P*
⟨*proof*⟩

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*
⟨*proof*⟩

instantiation *unit* :: *default*
begin

definition *default* = ()

instance ⟨*proof*⟩

end

lemma [*code*]:
eq-class.eq (*u*::*unit*) *v* ⟷ *True* ⟨*proof*⟩

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* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ **where**
$Pair\text{-}Rep\ a\ b = (\lambda x\ y.\ x = a \wedge y = b)$

global

typedef (*Prod*)
$('a, 'b) *$ (**infixr** $*$ *20*)
$= \{f.\ \exists\, a\ b.\ f = Pair\text{-}Rep\ (a{::}'a)\ (b{::}'b)\}$
⟨*proof*⟩

type-notation (*xsymbols*)
$*$ ((- ×/ -) [*21*, *20*] *20*)
type-notation (*HTML* **output**)
$*$ ((- ×/ -) [*21*, *20*] *20*)

consts
Pair :: $'a \Rightarrow 'b \Rightarrow 'a \times 'b$

local

defs
Pair-def: $Pair\ a\ b == Abs\text{-}Prod\ (Pair\text{-}Rep\ a\ b)$

rep-datatype (*prod*) *Pair* ⟨*proof*⟩

11.3.2 Tuple syntax

global consts
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$

local defs
split-prod-case: $split == prod\text{-}case$

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

⟨*ML*⟩

11.3.3 Code generator setup

lemma *split-case-cert*:
assumes *CASE* ≡ *split f*
shows *CASE* (*a*, *b*) ≡ *f a b*
⟨*proof*⟩

⟨*ML*⟩

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 ((-,/ -))

⟨*ML*⟩

11.3.4 Fundamental operations and properties

lemma *surj-pair* [*simp*]: *EX x y*. *p* = (*x*, *y*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *snd-conv* [*simp*, *code*]: *snd* (*a*, *b*) = *b*
  ⟨*proof*⟩

code-const *fst* **and** *snd*
  (*Haskell fst* **and** *snd*)

lemma *prod-case-unfold*: *prod-case* = (%*c p*. *c* (*fst p*) (*snd p*))
  ⟨*proof*⟩

lemma *fst-eqD*: *fst* (*x*, *y*) = *a* ==> *x* = *a*
  ⟨*proof*⟩

lemma *snd-eqD*: $snd\ (x,\ y) = a \Longrightarrow y = a$
⟨*proof*⟩

lemma *pair-collapse* [*simp*]: $(fst\ p,\ snd\ p) = p$
⟨*proof*⟩

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

lemma *Pair-fst-snd-eq*: $s = t \longleftrightarrow fst\ s = fst\ t \wedge snd\ s = snd\ t$
⟨*proof*⟩

lemma *prod-eqI* [*intro?*]: $fst\ p = fst\ q \Longrightarrow snd\ p = snd\ q \Longrightarrow p = q$
⟨*proof*⟩

lemma *split-conv* [*simp*, *code*]: $split\ f\ (a,\ b) = f\ a\ b$
⟨*proof*⟩

lemma *splitI*: $f\ a\ b \Longrightarrow split\ f\ (a,\ b)$
⟨*proof*⟩

lemma *splitD*: $split\ f\ (a,\ b) \Longrightarrow f\ a\ b$
⟨*proof*⟩

lemma *split-Pair* [*simp*]: $(\lambda(x,\ y).\ (x,\ y)) = id$
⟨*proof*⟩

lemma *split-eta*: $(\lambda(x,\ y).\ f\ (x,\ y)) = f$
— Subsumes the old *split-Pair* when f is the identity function.
⟨*proof*⟩

lemma *split-comp*: $split\ (f \circ g)\ x = f\ (g\ (fst\ x))\ (snd\ x)$
⟨*proof*⟩

lemma *split-twice*: $split\ f\ (split\ g\ p) = split\ (\lambda x\ y.\ split\ f\ (g\ x\ y))\ p$
⟨*proof*⟩

lemma *The-split*: $The\ (split\ P) = (THE\ xy.\ P\ (fst\ xy)\ (snd\ xy))$
⟨*proof*⟩

lemma *split-weak-cong*: $p = q \Longrightarrow split\ c\ p = split\ c\ q$
— Prevents simplification of c: much faster
⟨*proof*⟩

lemma *cond-split-eta*: $(!!x\ y.\ f\ x\ y = g\ (x,\ y)) \Longrightarrow (\%(x,\ y).\ f\ x\ y) = g$
⟨*proof*⟩

lemma *split-paired-all*: $(!!x.\ PROP\ P\ x) \equiv (!!a\ b.\ PROP\ P\ (a,\ b))$
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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

lemma *split-paired-The*: (*THE x*. *P x*) = (*THE* (*a*, *b*). *P* (*a*, *b*))
— Can't be added to simpset: loops!
⟨*proof*⟩

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*⟩

lemma *split-beta* [*mono*]: (%(*x*, *y*). *P x y*) *z* = *P* (*fst z*) (*snd z*)
⟨*proof*⟩

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.
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *splitI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *split c p x*
⟨*proof*⟩

lemma *splitE*: *split c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*

⟨*proof*⟩

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
⟨*proof*⟩

lemma *splitE2*:
[| *Q* (*split P z*); !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R* |] ==> *R*
⟨*proof*⟩

lemma *splitD'*: *split R* (*a*,*b*) *c* ==> *R a b c*
⟨*proof*⟩

lemma *mem-splitI*: *z*: *c a b* ==> *z*: *split c* (*a*, *b*)
⟨*proof*⟩

lemma *mem-splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *z*: *c a b* |] ==> *z*: *split c p*
⟨*proof*⟩

lemma *mem-splitE*:
assumes *major*: *z* ∈ *split c p*
and *cases*: ⋀*x y*. *p* = (*x*, *y*) ⟹ *z* ∈ *c x y* ⟹ *Q*
shows *Q*
⟨*proof*⟩

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

⟨*ML*⟩

lemma *split-eta-SetCompr* [*simp*,*no-atp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
⟨*proof*⟩

lemma *split-eta-SetCompr2* [*simp*,*no-atp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P x y*) = *split P*
⟨*proof*⟩

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.
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *pair-imageI* [*intro*]: (*a*, *b*) : *A* ==> *f a b* : (%(*a*, *b*). *f a b*) ' *A*

⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *prod-caseI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *prod-case c p x*
⟨*proof*⟩

lemma *prod-caseE*: *prod-case c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
⟨*proof*⟩

lemma *prod-caseE'*: *prod-case c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
⟨*proof*⟩

declare *prod-caseI2'* [*intro*!] *prod-caseI2* [*intro*!] *prod-caseI* [*intro*!]
declare *prod-caseE'* [*elim*!] *prod-caseE* [*elim*!]

lemma *prod-case-split*:
prod-case = *split*
⟨*proof*⟩

lemma *prod-case-beta*:
prod-case f p = *f* (*fst p*) (*snd p*)
⟨*proof*⟩

lemma *prod-cases3* [*cases type*]:
obtains (*fields*) *a b c* **where** *y* = (*a*, *b*, *c*)
⟨*proof*⟩

lemma *prod-induct3* [*case-names fields*, *induct type*]:
(!!*a b c*. *P* (*a*, *b*, *c*)) ==> *P x*
⟨*proof*⟩

lemma *prod-cases4* [*cases type*]:
obtains (*fields*) *a b c d* **where** *y* = (*a*, *b*, *c*, *d*)
⟨*proof*⟩

lemma *prod-induct4* [*case-names fields*, *induct type*]:
(!!*a b c d*. *P* (*a*, *b*, *c*, *d*)) ==> *P x*
⟨*proof*⟩

lemma *prod-cases5* [*cases type*]:
obtains (*fields*) *a b c d e* **where** $y = (a, b, c, d, e)$
⟨*proof*⟩

lemma *prod-induct5* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e.\ P\ (a, b, c, d, e)) ==> P\ x$
⟨*proof*⟩

lemma *prod-cases6* [*cases type*]:
obtains (*fields*) *a b c d e f* **where** $y = (a, b, c, d, e, f)$
⟨*proof*⟩

lemma *prod-induct6* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f.\ P\ (a, b, c, d, e, f)) ==> P\ x$
⟨*proof*⟩

lemma *prod-cases7* [*cases type*]:
obtains (*fields*) *a b c d e f g* **where** $y = (a, b, c, d, e, f, g)$
⟨*proof*⟩

lemma *prod-induct7* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f\ g.\ P\ (a, b, c, d, e, f, g)) ==> P\ x$
⟨*proof*⟩

lemma *split-def*:
split = $(\lambda c\ p.\ c\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

definition *internal-split* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$ **where**
internal-split == *split*

lemma *internal-split-conv*: *internal-split c* $(a, b) = c\ a\ b$
⟨*proof*⟩

⟨*ML*⟩

hide-const *internal-split*

11.3.5 Derived operations

global consts
curry :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

local defs
curry-def: *curry* == $(\%c\ x\ y.\ c\ (Pair\ x\ y))$

lemma *curry-conv* [*simp*, *code*]: *curry f a b* = $f\ (a, b)$
⟨*proof*⟩

lemma *curryI* [*intro*!]: $f\ (a,\ b) \Longrightarrow curry\ f\ a\ b$
⟨*proof*⟩

lemma *curryD* [*dest*!]: $curry\ f\ a\ b \Longrightarrow f\ (a,\ b)$
⟨*proof*⟩

lemma *curryE*: $curry\ f\ a\ b \Longrightarrow (f\ (a,\ b) \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *curry-split* [*simp*]: $curry\ (split\ f) = f$
⟨*proof*⟩

lemma *split-curry* [*simp*]: $split\ (curry\ f) = f$
⟨*proof*⟩

The composition-uncurry combinator.

notation *fcomp* (**infixl** *o>* *60*)

definition *scomp* :: $('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (**infixl** *o→* *60*)
where
$f\ o\!\!\rightarrow\ g = (\lambda x.\ split\ g\ (f\ x))$

lemma *scomp-apply*: $(f\ o\!\!\rightarrow\ g)\ x = split\ g\ (f\ x)$
⟨*proof*⟩

lemma *Pair-scomp*: $Pair\ x\ o\!\!\rightarrow\ f = f\ x$
⟨*proof*⟩

lemma *scomp-Pair*: $x\ o\!\!\rightarrow\ Pair = x$
⟨*proof*⟩

lemma *scomp-scomp*: $(f\ o\!\!\rightarrow\ g)\ o\!\!\rightarrow\ h = f\ o\!\!\rightarrow\ (\lambda x.\ g\ x\ o\!\!\rightarrow\ h)$
⟨*proof*⟩

lemma *scomp-fcomp*: $(f\ o\!\!\rightarrow\ g)\ o\!>\ h = f\ o\!\!\rightarrow\ (\lambda x.\ g\ x\ o\!>\ h)$
⟨*proof*⟩

lemma *fcomp-scomp*: $(f\ o\!>\ g)\ o\!\!\rightarrow\ h = f\ o\!>\ (g\ o\!\!\rightarrow\ h)$
⟨*proof*⟩

code-const *scomp*
(*Eval* **infixl** *3* #−>)

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

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

definition *prod-fun* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'd$ **where**
[*code del*]: $prod\text{-}fun\ f\ g = (\lambda(x,\ y).\ (f\ x,\ g\ y))$

lemma *prod-fun* [*simp*, *code*]: *prod-fun f g* (*a*, *b*) = (*f a*, *g b*)
⟨*proof*⟩

lemma *fst-prod-fun*[*simp*]: *fst* (*prod-fun f g x*) = *f* (*fst x*)
⟨*proof*⟩

lemma *snd-prod-fun*[*simp*]: *snd* (*prod-fun f g x*) = *g* (*snd x*)
⟨*proof*⟩

lemma *fst-comp-prod-fun*[*simp*]: *fst* ∘ *prod-fun f g* = *f* ∘ *fst*
⟨*proof*⟩

lemma *snd-comp-prod-fun*[*simp*]: *snd* ∘ *prod-fun f g* = *g* ∘ *snd*
⟨*proof*⟩

lemma *prod-fun-compose*:
prod-fun (*f1 o f2*) (*g1 o g2*) = (*prod-fun f1 g1 o prod-fun f2 g2*)
⟨*proof*⟩

lemma *prod-fun-ident* [*simp*]: *prod-fun* (%*x*. *x*) (%*y*. *y*) = (%*z*. *z*)
⟨*proof*⟩

lemma *prod-fun-imageI* [*intro*]: (*a*, *b*) : *r* ==> (*f a*, *g b*) : *prod-fun f g* ‘ *r*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *apsnd-conv* [*simp*, *code*]:
apsnd f (*x*, *y*) = (*x*, *f y*)
⟨*proof*⟩

lemma *fst-apfst* [*simp*]:

fst (*apfst f x*) = *f* (*fst x*)
⟨*proof*⟩

lemma *fst-apsnd* [*simp*]:
fst (*apsnd f x*) = *fst x*
⟨*proof*⟩

lemma *snd-apfst* [*simp*]:
snd (*apfst f x*) = *snd x*
⟨*proof*⟩

lemma *snd-apsnd* [*simp*]:
snd (*apsnd f x*) = *f* (*snd x*)
⟨*proof*⟩

lemma *apfst-compose*:
apfst f (*apfst g x*) = *apfst* (*f* ∘ *g*) *x*
⟨*proof*⟩

lemma *apsnd-compose*:
apsnd f (*apsnd g x*) = *apsnd* (*f* ∘ *g*) *x*
⟨*proof*⟩

lemma *apfst-apsnd* [*simp*]:
apfst f (*apsnd g x*) = (*f* (*fst x*), *g* (*snd x*))
⟨*proof*⟩

lemma *apsnd-apfst* [*simp*]:
apsnd f (*apfst g x*) = (*g* (*fst x*), *f* (*snd x*))
⟨*proof*⟩

lemma *apfst-id* [*simp*] :
apfst id = *id*
⟨*proof*⟩

lemma *apsnd-id* [*simp*] :
apsnd id = *id*
⟨*proof*⟩

lemma *apfst-eq-conv* [*simp*]:
apfst f x = *apfst g x* ⟷ *f* (*fst x*) = *g* (*fst x*)
⟨*proof*⟩

lemma *apsnd-eq-conv* [*simp*]:
apsnd f x = *apsnd g x* ⟷ *f* (*snd x*) = *g* (*snd x*)
⟨*proof*⟩

lemma *apsnd-apfst-commute*:
apsnd f (*apfst g p*) = *apfst g* (*apsnd f p*)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *SigmaE* [*elim*!]:
[| *c*: *Sigma A B*;
!!*x y*.[| *x*:*A*; *y*:*B*(*x*); *c*=(*x*,*y*) |] ==> *P*
|] ==> *P*
— The general elimination rule.
⟨*proof*⟩

Elimination of (*a*, *b*) ∈ *A* × *B* – introduces no eigenvariables.

lemma *SigmaD1*: (*a*, *b*) : *Sigma A B* ==> *a* : *A*
⟨*proof*⟩

lemma *SigmaD2*: (*a*, *b*) : *Sigma A B* ==> *b* : *B a*
⟨*proof*⟩

lemma *SigmaE2*:
[| (*a*, *b*) : *Sigma A B*;
[| *a*:*A*; *b*:*B*(*a*) |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *Sigma-cong*:
⟦*A* = *B*; !!*x*. *x* ∈ *B* ⟹ *C x* = *D x*⟧
⟹ (*SIGMA x*: *A*. *C x*) = (*SIGMA x*: *B*. *D x*)
⟨*proof*⟩

lemma *Sigma-mono*: [| $A <= C$; !!x. x:A ==> $B\ x <= D\ x$ |] ==> *Sigma* $A\ B$ <= *Sigma* $C\ D$
⟨*proof*⟩

lemma *Sigma-empty1* [*simp*]: *Sigma* {} $B = \{\}$
⟨*proof*⟩

lemma *Sigma-empty2* [*simp*]: $A <*> \{\} = \{\}$
⟨*proof*⟩

lemma *UNIV-Times-UNIV* [*simp*]: *UNIV* <*> *UNIV* = *UNIV*
⟨*proof*⟩

lemma *Compl-Times-UNIV1* [*simp*]: $-(UNIV <*> A) = UNIV <*> (-A)$
⟨*proof*⟩

lemma *Compl-Times-UNIV2* [*simp*]: $-(A <*> UNIV) = (-A) <*> UNIV$
⟨*proof*⟩

lemma *mem-Sigma-iff* [*iff*]: $((a,b)$: *Sigma* $A\ B) = (a$:A & b:$B(a))$
⟨*proof*⟩

lemma *Times-subset-cancel2*: x:C ==> $(A <*> C <= B <*> C) = (A <= B)$
⟨*proof*⟩

lemma *Times-eq-cancel2*: x:C ==> $(A <*> C = B <*> C) = (A = B)$
⟨*proof*⟩

lemma *SetCompr-Sigma-eq*:
Collect (*split* (%$x\ y$. $P\ x$ & $Q\ x\ y$)) = (*SIGMA* x:*Collect* P. *Collect* $(Q\ x)$)
⟨*proof*⟩

lemma *Collect-split* [*simp*]: $\{(a,b).\ P\ a$ & $Q\ b\} =$ *Collect* $P <*>$ *Collect* Q
⟨*proof*⟩

lemma *UN-Times-distrib*:
$(UN\ (a,b)$:$(A <*> B).\ E\ a <*> F\ b) = (UNION\ A\ E) <*> (UNION\ B\ F)$
— Suggested by Pierre Chartier
⟨*proof*⟩

lemma *split-paired-Ball-Sigma* [*simp*,*no-atp*]:
(*ALL* z: *Sigma* $A\ B$. $P\ z) = (ALL\ x$:A. *ALL* y: $B\ x$. $P(x,y))$
⟨*proof*⟩

lemma *split-paired-Bex-Sigma* [*simp*,*no-atp*]:
(*EX* z: *Sigma* $A\ B$. $P\ z) = (EX\ x$:A. *EX* y: $B\ x$. $P(x,y))$
⟨*proof*⟩

lemma *Sigma-Un-distrib1*: $(SIGMA\ i{:}I\ Un\ J.\ C(i)) = (SIGMA\ i{:}I.\ C(i))\ Un\ (SIGMA\ j{:}J.\ C(j))$
⟨*proof*⟩

lemma *Sigma-Un-distrib2*: $(SIGMA\ i{:}I.\ A(i)\ Un\ B(i)) = (SIGMA\ i{:}I.\ A(i))\ Un\ (SIGMA\ i{:}I.\ B(i))$
⟨*proof*⟩

lemma *Sigma-Int-distrib1*: $(SIGMA\ i{:}I\ Int\ J.\ C(i)) = (SIGMA\ i{:}I.\ C(i))\ Int\ (SIGMA\ j{:}J.\ C(j))$
⟨*proof*⟩

lemma *Sigma-Int-distrib2*: $(SIGMA\ i{:}I.\ A(i)\ Int\ B(i)) = (SIGMA\ i{:}I.\ A(i))\ Int\ (SIGMA\ i{:}I.\ B(i))$
⟨*proof*⟩

lemma *Sigma-Diff-distrib1*: $(SIGMA\ i{:}I - J.\ C(i)) = (SIGMA\ i{:}I.\ C(i)) - (SIGMA\ j{:}J.\ C(j))$
⟨*proof*⟩

lemma *Sigma-Diff-distrib2*: $(SIGMA\ i{:}I.\ A(i) - B(i)) = (SIGMA\ i{:}I.\ A(i)) - (SIGMA\ i{:}I.\ B(i))$
⟨*proof*⟩

lemma *Sigma-Union*: $Sigma\ (Union\ X)\ B = (UN\ A{:}X.\ Sigma\ A\ B)$
⟨*proof*⟩

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A\ Un\ B) <*> C = (A <*> C)\ Un\ (B <*> C)$
⟨*proof*⟩

lemma *Times-Int-distrib1*: $(A\ Int\ B) <*> C = (A <*> C)\ Int\ (B <*> C)$
⟨*proof*⟩

lemma *Times-Diff-distrib1*: $(A - B) <*> C = (A <*> C) - (B <*> C)$
⟨*proof*⟩

lemma *Times-empty*[*simp*]: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
⟨*proof*⟩

lemma *fst-image-times*[*simp*]: $fst\ `\ (A \times B) = (if\ B = \{\}\ then\ \{\}\ else\ A)$
⟨*proof*⟩

lemma *snd-image-times*[*simp*]: $snd\ `\ (A \times B) = (if\ A = \{\}\ then\ \{\}\ else\ B)$
⟨*proof*⟩

lemma *insert-times-insert*[*simp*]:
$insert\ a\ A \times insert\ b\ B =$

insert (*a*,*b*) (*A* × *insert b B* ∪ *insert a A* × *B*)
⟨*proof*⟩

lemma *vimage-Times*: *f* −‘ (*A* × *B*) = ((*fst* ∘ *f*) −‘ *A*) ∩ ((*snd* ∘ *f*) −‘ *B*)
⟨*proof*⟩

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* × *B*)
⟨*proof*⟩

lemma *prod-fun-surj*:
assumes *surj f* **and** *surj g*
shows *surj* (*prod-fun f g*)
⟨*proof*⟩

lemma *prod-fun-surj-on*:
assumes *f* ‘ *A* = *A*′ **and** *g* ‘ *B* = *B*′
shows *prod-fun f g* ‘ (*A* × *B*) = *A*′ × *B*′
⟨*proof*⟩

lemma *swap-inj-on*:
inj-on (λ(*i*, *j*). (*j*, *i*)) *A*
⟨*proof*⟩

lemma *swap-product*:
(%(*i*, *j*). (*j*, *i*)) ‘ (*A* × *B*) = *B* × *A*
⟨*proof*⟩

lemma *image-split-eq-Sigma*:
(λ*x*. (*f x*, *g x*)) ‘ *A* = *Sigma* (*f* ‘ *A*) (λ*x*. *g* ‘ (*f* −‘ {*x*} ∩ *A*))
⟨*proof*⟩

11.4 Inductively defined sets

⟨*ML*⟩

11.5 Legacy theorem bindings and duplicates

lemma *PairE*:
obtains *x y* **where** *p* = (*x*, *y*)
⟨*proof*⟩

lemma *Pair-inject*:
assumes (*a*, *b*) = (*a*′, *b*′)
and *a* = *a*′ ==> *b* = *b*′ ==> *R*
shows *R*
⟨*proof*⟩

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*) = $\{f.\ (\exists\, a.\ f = \textit{Inl-Rep}\ (a::'a)) \vee (\exists\, b.\ f = \textit{Inr-Rep}\ (b::'b))\}$
⟨*proof*⟩

local

lemma *Inl-RepI*: *Inl-Rep a* $\in$ *Sum*
⟨*proof*⟩

lemma *Inr-RepI*: *Inr-Rep b* $\in$ *Sum*
⟨*proof*⟩

lemma *inj-on-Abs-Sum*: $A \subseteq \textit{Sum} \Longrightarrow \textit{inj-on Abs-Sum } A$
⟨*proof*⟩

lemma *Inl-Rep-inject*: *inj-on Inl-Rep A*
⟨*proof*⟩

lemma *Inr-Rep-inject*: *inj-on Inr-Rep A*
⟨*proof*⟩

lemma *Inl-Rep-not-Inr-Rep*: *Inl-Rep a* $\neq$ *Inr-Rep b*
⟨*proof*⟩

definition *Inl* :: $'a \Rightarrow 'a + 'b$ **where**
Inl = *Abs-Sum* ∘ *Inl-Rep*

definition *Inr* :: $'b \Rightarrow 'a + 'b$ **where**
Inr = *Abs-Sum* ∘ *Inr-Rep*

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
⟨*proof*⟩

lemma *Inl-inject*: *Inl x* = *Inl y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
⟨*proof*⟩

lemma *Inr-inject*: *Inr x* = *Inr y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *Inl-not-Inr*: *Inl a* ≠ *Inr b*
⟨*proof*⟩

lemma *Inr-not-Inl*: *Inr b* ≠ *Inl a*
⟨*proof*⟩

lemma *sumE*:
assumes ⋀*x*::$'a$. *s* = *Inl x* ⟹ *P*
and ⋀*y*::$'b$. *s* = *Inr y* ⟹ *P*
shows *P*
⟨*proof*⟩

rep-datatype (*sum*) *Inl Inr*
⟨*proof*⟩

12.2 Projections

lemma *sum-case-KK* [*simp*]: *sum-case* (λ*x*. *a*) (λ*x*. *a*) = (λ*x*. *a*)
⟨*proof*⟩

lemma *surjective-sum*: *sum-case* (λ*x*::$'a$. *f* (*Inl x*)) (λ*y*::$'b$. *f* (*Inr y*)) = *f*
⟨*proof*⟩

lemma *sum-case-inject*:
assumes *a*: *sum-case f1 f2* = *sum-case g1 g2*
assumes *r*: *f1* = *g1* ⟹ *f2* = *g2* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *sum-case-weak-cong*:
s = *t* ⟹ *sum-case f g s* = *sum-case f g t*

— Prevents simplification of f and g: much faster.
⟨*proof*⟩

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*⟩

lemma *Sumr-inject*:
assumes $Sumr\ f = Sumr\ g$ **shows** $f = g$
⟨*proof*⟩

12.3 The Disjoint Sum of Sets

definition *Plus* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a + 'b)\ set$ (**infixr** <+> *65*) **where**
$A <+> B = Inl\ `\ A \cup Inr\ `\ B$

lemma *InlI* [*intro*!]: $a \in A \Longrightarrow Inl\ a \in A <+> B$
⟨*proof*⟩

lemma *InrI* [*intro*!]: $b \in B \Longrightarrow Inr\ b \in A <+> B$
⟨*proof*⟩

Exhaustion rule for sums, a degenerate form of induction

lemma *PlusE* [*elim*!]:
$u \in A <+> B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow u = Inl\ x \Longrightarrow P) \Longrightarrow (\bigwedge y.\ y \in B \Longrightarrow u = Inr\ y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Plus-eq-empty-conv* [*simp*]: $A <+> B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
⟨*proof*⟩

lemma *UNIV-Plus-UNIV* [*simp*]: $UNIV <+> UNIV = UNIV$
⟨*proof*⟩

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*:
  $a * e + (b * e + c) = (a + b) * e + c$
⟨*proof*⟩

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*⟩

end

class *comm-semiring* = *ab-semigroup-add* + *ab-semigroup-mult* +
  assumes *distrib*: $(a + b) * c = a * c + b * c$
begin

subclass *semiring*
⟨*proof*⟩

end

class *comm-semiring-0* = *comm-semiring* + *comm-monoid-add* + *mult-zero*
begin

subclass *semiring-0* ⟨*proof*⟩

end

class *comm-semiring-0-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *comm-semiring-0* ⟨*proof*⟩

end

class *zero-neq-one* = *zero* + *one* +
  assumes *zero-neq-one* [*simp*]: *0* ≠ *1*
begin

lemma *one-neq-zero* [*simp*]: *1* ≠ *0*
⟨*proof*⟩

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** *dvd 50*) **where**
  [*code del*]: *b dvd a* ⟷ (∃ *k*. *a* = *b* ∗ *k*)

lemma *dvdI* [*intro?*]: *a* = *b* ∗ *k* ⟹ *b dvd a*
  ⟨*proof*⟩

lemma *dvdE* [*elim?*]: *b dvd a* ⟹ (⋀*k*. *a* = *b* ∗ *k* ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

end

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*
+ *dvd*

begin

subclass *semiring-1* ⟨*proof*⟩

lemma *dvd-refl*[*simp*]: *a dvd a*
⟨*proof*⟩

lemma *dvd-trans*:
  assumes *a dvd b* **and** *b dvd c*
  shows *a dvd c*

⟨*proof*⟩

lemma *dvd-0-left-iff* [*no-atp*, *simp*]: *0 dvd a* ⟷ *a = 0*
⟨*proof*⟩

lemma *dvd-0-right* [*iff*]: *a dvd 0*
⟨*proof*⟩

lemma *one-dvd* [*simp*]: *1 dvd a*
⟨*proof*⟩

lemma *dvd-mult*[*simp*]: *a dvd c* ⟹ *a dvd* (*b* ∗ *c*)
⟨*proof*⟩

lemma *dvd-mult2*[*simp*]: *a dvd b* ⟹ *a dvd* (*b* ∗ *c*)
⟨*proof*⟩

lemma *dvd-triv-right* [*simp*]: *a dvd b* ∗ *a*
⟨*proof*⟩

lemma *dvd-triv-left* [*simp*]: *a dvd a* ∗ *b*
⟨*proof*⟩

lemma *mult-dvd-mono*:
assumes *a dvd b*
and *c dvd d*
shows *a* ∗ *c dvd b* ∗ *d*
⟨*proof*⟩

lemma *dvd-mult-left*: *a* ∗ *b dvd c* ⟹ *a dvd c*
⟨*proof*⟩

lemma *dvd-mult-right*: *a* ∗ *b dvd c* ⟹ *b dvd c*
⟨*proof*⟩

lemma *dvd-0-left*: *0 dvd a* ⟹ *a = 0*
⟨*proof*⟩

lemma *dvd-add*[*simp*]:
assumes *a dvd b* **and** *a dvd c* **shows** *a dvd* (*b* + *c*)
⟨*proof*⟩

end

class *no-zero-divisors = zero + times +*
assumes *no-zero-divisors*: *a* ≠ *0* ⟹ *b* ≠ *0* ⟹ *a* ∗ *b* ≠ *0*
begin

lemma *divisors-zero*:

assumes $a * b = 0$
shows $a = 0 \vee b = 0$
⟨*proof*⟩

end

class *semiring-1-cancel* = *semiring* + *cancel-comm-monoid-add*
+ *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *semiring-1* ⟨*proof*⟩

end

class *comm-semiring-1-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
+ *zero-neq-one* + *comm-monoid-mult*
begin

subclass *semiring-1-cancel* ⟨*proof*⟩
subclass *comm-semiring-0-cancel* ⟨*proof*⟩
subclass *comm-semiring-1* ⟨*proof*⟩

end

class *ring* = *semiring* + *ab-group-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

Distribution rules

lemma *minus-mult-left*: $- (a * b) = - a * b$
⟨*proof*⟩

lemma *minus-mult-right*: $- (a * b) = a * - b$
⟨*proof*⟩

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*]: $- a * - b = a * b$
⟨*proof*⟩

lemma *minus-mult-commute*: $- a * b = a * - b$
⟨*proof*⟩

lemma *right-diff-distrib*[*algebra-simps*, *field-simps*]: $a * (b - c) = a * b - a * c$

⟨*proof*⟩

lemma *left-diff-distrib*[*algebra-simps*, *field-simps*]: $(a - b) * c = a * c - b * c$
⟨*proof*⟩

lemmas *ring-distribs*[*no-atp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

lemma *eq-add-iff1*:
$a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
⟨*proof*⟩

lemma *eq-add-iff2*:
$a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$
⟨*proof*⟩

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* ⟨*proof*⟩
subclass *comm-semiring-0-cancel* ⟨*proof*⟩

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* ⟨*proof*⟩

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*

begin

subclass *ring-1* ⟨*proof*⟩
subclass *comm-semiring-1-cancel* ⟨*proof*⟩

lemma *dvd-minus-iff* [*simp*]: *x dvd* − *y* ⟷ *x dvd y*
⟨*proof*⟩

lemma *minus-dvd-iff* [*simp*]: − *x dvd y* ⟷ *x dvd y*
⟨*proof*⟩

lemma *dvd-diff*[*simp*]: $x\ dvd\ y \Longrightarrow x\ dvd\ z \Longrightarrow x\ dvd\ (y - z)$
⟨*proof*⟩

end

class *ring-no-zero-divisors* = *ring* + *no-zero-divisors*
begin

lemma *mult-eq-0-iff* [*simp*]:
shows $a * b = 0 \longleftrightarrow (a = 0 \vee b = 0)$
⟨*proof*⟩

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*, *no-atp*]:
$a * c = b * c \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

lemma *mult-cancel-left* [*simp*, *no-atp*]:
$c * a = c * b \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *square-eq-1-iff*:
$x * x = 1 \longleftrightarrow x = 1 \vee x = -1$
⟨*proof*⟩

lemma *mult-cancel-right1* [*simp*]:
$c = b * c \longleftrightarrow c = 0 \vee b = 1$
⟨*proof*⟩

lemma *mult-cancel-right2* [*simp*]:
$a * c = c \longleftrightarrow c = 0 \vee a = 1$
⟨*proof*⟩

lemma *mult-cancel-left1* [*simp*]:
$c = c * b \longleftrightarrow c = 0 \vee b = 1$
⟨*proof*⟩

lemma *mult-cancel-left2* [*simp*]:
$c * a = c \longleftrightarrow c = 0 \vee a = 1$
⟨*proof*⟩

end

class *idom* = *comm-ring-1* + *no-zero-divisors*

begin

subclass *ring-1-no-zero-divisors* ⟨*proof*⟩

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow (a = b \vee a = - b)$
⟨*proof*⟩

lemma *dvd-mult-cancel-right* [*simp*]:
$a * c$ *dvd* $b * c \longleftrightarrow c = 0 \vee a$ *dvd* b
⟨*proof*⟩

lemma *dvd-mult-cancel-left* [*simp*]:
$c * a$ *dvd* $c * b \longleftrightarrow c = 0 \vee a$ *dvd* b
⟨*proof*⟩

end

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

class *division-ring* = *ring-1* + *inverse* +
assumes *left-inverse* [*simp*]: $a \neq 0 \Longrightarrow$ *inverse* $a * a = 1$
assumes *right-inverse* [*simp*]: $a \neq 0 \Longrightarrow a *$ *inverse* $a = 1$
assumes *divide-inverse*: a / $b = a *$ *inverse* b
begin

subclass *ring-1-no-zero-divisors*
⟨*proof*⟩

lemma *nonzero-imp-inverse-nonzero*:
$a \neq 0 \Longrightarrow$ *inverse* $a \neq 0$
⟨*proof*⟩

lemma *inverse-zero-imp-zero*:
inverse $a = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inverse-unique*:
assumes *ab*: $a * b = 1$
shows *inverse* $a = b$
⟨*proof*⟩

lemma *nonzero-inverse-minus-eq*:
$a \neq 0 \Longrightarrow$ *inverse* $(- a) = -$ *inverse* a
⟨*proof*⟩

lemma *nonzero-inverse-inverse-eq*:
$a \neq 0 \Longrightarrow$ *inverse* (*inverse* a) $= a$

⟨*proof*⟩

lemma *nonzero-inverse-eq-imp-eq*:
assumes *inverse a* = *inverse b* **and** *a* ≠ *0* **and** *b* ≠ *0*
shows *a* = *b*
⟨*proof*⟩

lemma *inverse-1* [*simp*]: *inverse 1* = *1*
⟨*proof*⟩

lemma *nonzero-inverse-mult-distrib*:
assumes *a* ≠ *0* **and** *b* ≠ *0*
shows *inverse* (*a* ∗ *b*) = *inverse b* ∗ *inverse a*
⟨*proof*⟩

lemma *division-ring-inverse-add*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *inverse a* + *inverse b* = *inverse a* ∗ (*a* + *b*) ∗ *inverse b*
⟨*proof*⟩

lemma *division-ring-inverse-diff*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *inverse a* − *inverse b* = *inverse a* ∗ (*b* − *a*) ∗ *inverse b*
⟨*proof*⟩

lemma *right-inverse-eq*: *b* ≠ *0* ⟹ *a* / *b* = *1* ⟷ *a* = *b*
⟨*proof*⟩

lemma *nonzero-inverse-eq-divide*: *a* ≠ *0* ⟹ *inverse a* = *1* / *a*
⟨*proof*⟩

lemma *divide-self* [*simp*]: *a* ≠ *0* ⟹ *a* / *a* = *1*
⟨*proof*⟩

lemma *divide-zero-left* [*simp*]: *0* / *a* = *0*
⟨*proof*⟩

lemma *inverse-eq-divide*: *inverse a* = *1* / *a*
⟨*proof*⟩

lemma *add-divide-distrib*: (*a*+*b*) / *c* = *a*/*c* + *b*/*c*
⟨*proof*⟩

lemma *divide-1* [*simp*]: *a* / *1* = *a*
⟨*proof*⟩

lemma *times-divide-eq-right* [*simp*]: *a* ∗ (*b* / *c*) = (*a* ∗ *b*) / *c*
⟨*proof*⟩

lemma *minus-divide-left*: − (*a* / *b*) = (−*a*) / *b*
⟨*proof*⟩

lemma *nonzero-minus-divide-right*: $b \neq 0 ==> -(a / b) = a / (-b)$
⟨*proof*⟩

lemma *nonzero-minus-divide-divide*: $b \neq 0 ==> (-a) / (-b) = a / b$
⟨*proof*⟩

lemma *divide-minus-left* [*simp*, *no-atp*]: $(-a) / b = -(a / b)$
⟨*proof*⟩

lemma *diff-divide-distrib*: $(a - b) / c = a / c - b / c$
⟨*proof*⟩

lemma *nonzero-eq-divide-eq* [*field-simps*]: $c \neq 0 \Longrightarrow a = b / c \longleftrightarrow a * c = b$
⟨*proof*⟩

lemma *nonzero-divide-eq-eq* [*field-simps*]: $c \neq 0 \Longrightarrow b / c = a \longleftrightarrow b = a * c$
⟨*proof*⟩

lemma *divide-eq-imp*: $c \neq 0 \Longrightarrow b = a * c \Longrightarrow b / c = a$
⟨*proof*⟩

lemma *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b / c$
⟨*proof*⟩

end

class *division-ring-inverse-zero* = *division-ring* +
assumes *inverse-zero* [*simp*]: *inverse 0* = *0*
begin

lemma *divide-zero* [*simp*]:
$a / 0 = 0$
⟨*proof*⟩

lemma *divide-self-if* [*simp*]:
$a / a = (\textit{if } a = 0 \textit{ then } 0 \textit{ else } 1)$
⟨*proof*⟩

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
$\textit{inverse } a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *inverse-minus-eq* [*simp*]:
$\textit{inverse } (-a) = -\textit{ inverse } a$
⟨*proof*⟩

lemma *inverse-eq-imp-eq*:
$\textit{inverse } a = \textit{inverse } b \Longrightarrow a = b$

⟨*proof*⟩

lemma *inverse-eq-iff-eq* [*simp*]:
  inverse a = *inverse b* ⟷ *a* = *b*
  ⟨*proof*⟩

lemma *inverse-inverse-eq* [*simp*]:
  inverse (*inverse a*) = *a*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mult-mono′*:
  a ≤ *b* ⟹ *c* ≤ *d* ⟹ 0 ≤ *a* ⟹ 0 ≤ *c*
    ⟹ *a* ∗ *c* ≤ *b* ∗ *d*
⟨*proof*⟩

end

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

subclass *semiring-0-cancel* ⟨*proof*⟩
subclass *ordered-semiring* ⟨*proof*⟩

lemma *mult-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * b$
⟨*proof*⟩

lemma *mult-nonneg-nonpos*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow a * b \leq 0$
⟨*proof*⟩

lemma *mult-nonpos-nonneg*: $a \leq 0 \Longrightarrow 0 \leq b \Longrightarrow a * b \leq 0$
⟨*proof*⟩

Legacy - use *mult-nonpos-nonneg*

lemma *mult-nonneg-nonpos2*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow b * a \leq 0$
⟨*proof*⟩

lemma *split-mult-neg-le*: $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \Longrightarrow a * b \leq 0$
⟨*proof*⟩

end

class *linordered-semiring* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add* + *mult-mono*
begin

subclass *ordered-cancel-semiring* ⟨*proof*⟩

subclass *ordered-comm-monoid-add* ⟨*proof*⟩

lemma *mult-left-less-imp-less*:
$c * a < c * b \Longrightarrow 0 \leq c \Longrightarrow a < b$
⟨*proof*⟩

lemma *mult-right-less-imp-less*:
$a * c < b * c \Longrightarrow 0 \leq c \Longrightarrow a < b$
⟨*proof*⟩

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*⟩

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* ⟨*proof*⟩

subclass *linordered-semiring*
⟨*proof*⟩

lemma *mult-left-le-imp-le*:
$c * a \leq c * b \Longrightarrow 0 < c \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *mult-right-le-imp-le*:
$a * c \leq b * c \Longrightarrow 0 < c \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *mult-pos-pos*: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
⟨*proof*⟩

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
⟨*proof*⟩

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$
⟨*proof*⟩

Legacy - use *mult-neg-pos*

lemma *mult-pos-neg2*: $0 < a \Longrightarrow b < 0 \Longrightarrow b * a < 0$
⟨*proof*⟩

lemma *zero-less-mult-pos*:
$0 < a * b \Longrightarrow 0 < a \Longrightarrow 0 < b$
⟨*proof*⟩

lemma *zero-less-mult-pos2*:
$0 < b * a \Longrightarrow 0 < a \Longrightarrow 0 < b$
⟨*proof*⟩

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ **and** $c < d$ **and** $0 < b$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

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 $a * c < b * d$
⟨*proof*⟩

lemma *mult-less-le-imp-less*:
assumes $a < b$ **and** $c \leq d$ **and** $0 \leq a$ **and** $0 < c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-less-imp-less-left*:
assumes *less*: $c * a < c * b$ **and** *nonneg*: $0 \leq c$
shows $a < b$
⟨*proof*⟩

lemma *mult-less-imp-less-right*:
assumes *less*: $a * c < b * c$ **and** *nonneg*: $0 \leq c$
shows $a < b$
⟨*proof*⟩

end

class *linordered-semiring-1-strict* = *linordered-semiring-strict* + *semiring-1*
begin

subclass *linordered-semiring-1* ⟨*proof*⟩

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*⟩

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*⟩

end

class *ordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *ordered-ab-semigroup-add* + *mult-mono1*

begin

subclass *ordered-comm-semiring* ⟨*proof*⟩
subclass *ordered-cancel-semiring* ⟨*proof*⟩

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*⟩

subclass *ordered-cancel-comm-semiring*
⟨*proof*⟩

end

class *ordered-ring* = *ring* + *ordered-cancel-semiring*
begin

subclass *ordered-ab-group-add* ⟨*proof*⟩

lemma *less-add-iff1*:
  $a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
⟨*proof*⟩

lemma *less-add-iff2*:
  $a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$
⟨*proof*⟩

lemma *le-add-iff1*:
  $a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
⟨*proof*⟩

lemma *le-add-iff2*:
  $a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
⟨*proof*⟩

lemma *mult-left-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow c * a \leq c * b$
  ⟨*proof*⟩

lemma *mult-right-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a * c \leq b * c$
  ⟨*proof*⟩

lemma *mult-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a * b$
⟨*proof*⟩

lemma *split-mult-pos-le*:
$(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a * b$
⟨*proof*⟩

end

class *linordered-ring* = *ring* + *linordered-semiring* + *linordered-ab-group-add* + *abs-if*
begin

subclass *ordered-ring* ⟨*proof*⟩

subclass *ordered-ab-group-add-abs*
⟨*proof*⟩

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
⟨*proof*⟩

lemma *not-square-less-zero* [*simp*]: $\neg (a * a < 0)$
⟨*proof*⟩

end

class *linordered-ring-strict* = *ring* + *linordered-semiring-strict*
+ *ordered-ab-group-add* + *abs-if*
begin

subclass *linordered-ring* ⟨*proof*⟩

lemma *mult-strict-left-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow c * a < c * b$
⟨*proof*⟩

lemma *mult-strict-right-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow a * c < b * c$
⟨*proof*⟩

lemma *mult-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow 0 < a * b$
⟨*proof*⟩

subclass *ring-no-zero-divisors*
⟨*proof*⟩

lemma *zero-less-mult-iff*:
$0 < a * b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
⟨*proof*⟩

lemma *zero-le-mult-iff*:
$0 \leq a * b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$
⟨*proof*⟩

lemma *mult-less-0-iff*:
$a * b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
⟨*proof*⟩

lemma *mult-le-0-iff*:
$a * b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
⟨*proof*⟩

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*:
$a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
⟨*proof*⟩

lemma *mult-less-cancel-left-disj*:
$c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
⟨*proof*⟩

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*:
$a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
⟨*proof*⟩

lemma *mult-less-cancel-left*:
$c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
⟨*proof*⟩

lemma *mult-le-cancel-right*:
$a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
⟨*proof*⟩

lemma *mult-le-cancel-left*:
$c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
⟨*proof*⟩

lemma *mult-le-cancel-left-pos*:
$0 < c \Longrightarrow c * a \leq c * b \longleftrightarrow a \leq b$
⟨*proof*⟩

lemma *mult-le-cancel-left-neg*:
$c < 0 \Longrightarrow c * a \leq c * b \longleftrightarrow b \leq a$

⟨*proof*⟩

lemma *mult-less-cancel-left-pos*:
  $0 < c \Longrightarrow c * a < c * b \longleftrightarrow a < b$
⟨*proof*⟩

lemma *mult-less-cancel-left-neg*:
  $c < 0 \Longrightarrow c * a < c * b \longleftrightarrow b < a$
⟨*proof*⟩

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* ⟨*proof*⟩
subclass *ordered-cancel-comm-semiring* ⟨*proof*⟩

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 $0 < a \Longrightarrow b < c \Longrightarrow b < a + c$
  ⟨*proof*⟩

lemma *zero-le-one* [*simp*]: $0 \leq 1$
⟨*proof*⟩

lemma *not-one-le-zero* [*simp*]: $\neg\ 1 \leq 0$
⟨*proof*⟩

lemma *not-one-less-zero* [*simp*]: $\neg\ 1 < 0$
⟨*proof*⟩

lemma *less-1-mult*:
  assumes $1 < m$ **and** $1 < n$
  shows $1 < m * n$
  ⟨*proof*⟩

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* ⟨*proof*⟩
subclass *linordered-ring-strict* ⟨*proof*⟩
subclass *ordered-comm-ring* ⟨*proof*⟩
subclass *idom* ⟨*proof*⟩

subclass *linordered-semidom*
⟨*proof*⟩

lemma *linorder-neqE-linordered-idom*:
assumes $x \neq y$ **obtains** $x < y \mid y < x$
⟨*proof*⟩

These cancellation simprules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
$c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
⟨*proof*⟩

lemma *mult-le-cancel-right2*:
$a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
⟨*proof*⟩

lemma *mult-le-cancel-left1*:
$c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
⟨*proof*⟩

lemma *mult-le-cancel-left2*:
$c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
⟨*proof*⟩

lemma *mult-less-cancel-right1*:
$c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
⟨*proof*⟩

lemma *mult-less-cancel-right2*:
$a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
⟨*proof*⟩

lemma *mult-less-cancel-left1*:
$c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

⟨*proof*⟩

lemma *mult-less-cancel-left2*:
  $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
⟨*proof*⟩

lemma *sgn-sgn* [*simp*]:
  $sgn\ (sgn\ a) = sgn\ a$
⟨*proof*⟩

lemma *sgn-0-0*:
  $sgn\ a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *sgn-1-pos*:
  $sgn\ a = 1 \longleftrightarrow a > 0$
⟨*proof*⟩

lemma *sgn-1-neg*:
  $sgn\ a = -\ 1 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *sgn-pos* [*simp*]:
  $0 < a \Longrightarrow sgn\ a = 1$
⟨*proof*⟩

lemma *sgn-neg* [*simp*]:
  $a < 0 \Longrightarrow sgn\ a = -\ 1$
⟨*proof*⟩

lemma *sgn-times*:
  $sgn\ (a * b) = sgn\ a * sgn\ b$
⟨*proof*⟩

lemma *abs-sgn*: $|k| = k * sgn\ k$
⟨*proof*⟩

lemma *sgn-greater* [*simp*]:
  $0 < sgn\ a \longleftrightarrow 0 < a$
  ⟨*proof*⟩

lemma *sgn-less* [*simp*]:
  $sgn\ a < 0 \longleftrightarrow a < 0$
  ⟨*proof*⟩

lemma *abs-dvd-iff* [*simp*]: $|m|\ dvd\ k \longleftrightarrow m\ dvd\ k$
  ⟨*proof*⟩

lemma *dvd-abs-iff* [*simp*]: $m\ dvd\ |k| \longleftrightarrow m\ dvd\ k$

⟨*proof*⟩

lemma *dvd-if-abs-eq*:
$|l| = |k| \Longrightarrow l$ *dvd* k
⟨*proof*⟩

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*⟩

lemma *zero-less-two*: $0 < 1 + 1$
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *mult-right-le-one-le*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow x * y \leq x$
⟨*proof*⟩

lemma *mult-left-le-one-le*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow y * x \leq x$
⟨*proof*⟩

end

Absolute Value

context *linordered-idom*
begin

lemma *mult-sgn-abs*:
*sgn x * |x| = x*
⟨*proof*⟩

lemma *abs-one* [*simp*]:
|1| = 1
⟨*proof*⟩

end

class *ordered-ring-abs = ordered-ring + ordered-ab-group-add-abs +*
assumes *abs-eq-mult*:
*(0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0) ⟹ |a * b| = |a| * |b|*

context *linordered-idom*
begin

subclass *ordered-ring-abs* ⟨*proof*⟩

lemma *abs-mult*:
*|a * b| = |a| * |b|*
⟨*proof*⟩

lemma *abs-mult-self*:
*|a| * |a| = a * a*
⟨*proof*⟩

lemma *abs-mult-less*:
*|a| < c ⟹ |b| < d ⟹ |a| * |b| < c * d*
⟨*proof*⟩

lemma *less-minus-self-iff*:
a < - a ⟷ a < 0
⟨*proof*⟩

lemma *abs-less-iff*:
|a| < b ⟷ a < b ∧ - a < b
⟨*proof*⟩

lemma *abs-mult-pos*:
*0 ≤ x ⟹ |y| * x = |y * x|*
⟨*proof*⟩

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*⟩

subclass *idom* ⟨*proof*⟩

There is no slick version using division by zero.

lemma *inverse-add*:
$[|\ a \neq 0;\ \ b \neq 0\ |]$
$\Longrightarrow inverse\ a + inverse\ b = (a + b) * inverse\ a * inverse\ b$
⟨*proof*⟩

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*⟩

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*, *no-atp*]:
$[\![b \neq 0;\ c \neq 0]\!] \Longrightarrow (a * c)\ /\ (b * c) = a\ /\ b$
⟨*proof*⟩

lemma *times-divide-eq-left* [*simp*]: $(b\ /\ c) * a = (b * a)\ /\ c$
⟨*proof*⟩

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*⟩

Special Cancellation Simprules for Division

lemma *nonzero-mult-divide-cancel-right* [*simp*, *no-atp*]:
$b \neq 0 \Longrightarrow a * b \mathbin{/} b = a$
⟨*proof*⟩

lemma *nonzero-mult-divide-cancel-left* [*simp*, *no-atp*]:
$a \neq 0 \Longrightarrow a * b \mathbin{/} a = b$
⟨*proof*⟩

lemma *nonzero-divide-mult-cancel-right* [*simp*, *no-atp*]:
$[\![a \neq 0; b \neq 0]\!] \Longrightarrow b \mathbin{/} (a * b) = 1 \mathbin{/} a$
⟨*proof*⟩

lemma *nonzero-divide-mult-cancel-left* [*simp*, *no-atp*]:
$[\![a \neq 0; b \neq 0]\!] \Longrightarrow a \mathbin{/} (a * b) = 1 \mathbin{/} b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*, *no-atp*]:
$[\![b \neq 0; c \neq 0]\!] \Longrightarrow (c * a) \mathbin{/} (b * c) = a \mathbin{/} b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*, *no-atp*]:
$[\![b \neq 0; c \neq 0]\!] \Longrightarrow (a * c) \mathbin{/} (c * b) = a \mathbin{/} b$
⟨*proof*⟩

lemma *add-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x + y \mathbin{/} z = (z * x + y) \mathbin{/} z$
⟨*proof*⟩

lemma *divide-add-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x \mathbin{/} z + y = (x + z * y) \mathbin{/} z$
⟨*proof*⟩

lemma *diff-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x - y \mathbin{/} z = (z * x - y) \mathbin{/} z$
⟨*proof*⟩

lemma *divide-diff-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x \mathbin{/} z - y = (x - z * y) \mathbin{/} z$
⟨*proof*⟩

lemma *diff-frac-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y - w \mathbin{/} z = (x * z - w * y) \mathbin{/} (y * z)$
⟨*proof*⟩

lemma *frac-eq-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow (x \mathbin{/} y = w \mathbin{/} z) = (x * z = w * y)$
⟨*proof*⟩

end

class *field-inverse-zero* = *field* +
  assumes *field-inverse-zero*: *inverse 0 = 0*
begin

subclass *division-ring-inverse-zero* ⟨*proof*⟩

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*⟩

lemma *inverse-divide* [*simp*]:
  inverse (*a* / *b*) = *b* / *a*
  ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mult-divide-mult-cancel-right*:
  c ≠ *0* ⟹ (*a* ∗ *c*) / (*b* ∗ *c*) = *a* / *b*
⟨*proof*⟩

lemma *divide-divide-eq-right* [*simp*, *no-atp*]:
  a / (*b* / *c*) = (*a* ∗ *c*) / *b*
  ⟨*proof*⟩

lemma *divide-divide-eq-left* [*simp*, *no-atp*]:
  (*a* / *b*) / *c* = *a* / (*b* ∗ *c*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

Division and Unary Minus

lemma *minus-divide-right*:
  − (*a* / *b*) = *a* / − *b*
  ⟨*proof*⟩

lemma *divide-minus-right* [*simp*, *no-atp*]:
$a \mathbin{/} - b = - (a \mathbin{/} b)$
⟨*proof*⟩

lemma *minus-divide-divide*:
$(- a) \mathbin{/} (- b) = a \mathbin{/} b$
⟨*proof*⟩

lemma *eq-divide-eq*:
$a = b \mathbin{/} c \longleftrightarrow (\textit{if } c \neq 0 \textit{ then } a * c = b \textit{ else } a = 0)$
⟨*proof*⟩

lemma *divide-eq-eq*:
$b \mathbin{/} c = a \longleftrightarrow (\textit{if } c \neq 0 \textit{ then } b = a * c \textit{ else } a = 0)$
⟨*proof*⟩

lemma *inverse-eq-1-iff* [*simp*]:
$\textit{inverse } x = 1 \longleftrightarrow x = 1$
⟨*proof*⟩

lemma *divide-eq-0-iff* [*simp*, *no-atp*]:
$a \mathbin{/} b = 0 \longleftrightarrow a = 0 \vee b = 0$
⟨*proof*⟩

lemma *divide-cancel-right* [*simp*, *no-atp*]:
$a \mathbin{/} c = b \mathbin{/} c \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

lemma *divide-cancel-left* [*simp*, *no-atp*]:
$c \mathbin{/} a = c \mathbin{/} b \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

lemma *divide-eq-1-iff* [*simp*, *no-atp*]:
$a \mathbin{/} b = 1 \longleftrightarrow b \neq 0 \wedge a = b$
⟨*proof*⟩

lemma *one-eq-divide-iff* [*simp*, *no-atp*]:
$1 = a \mathbin{/} b \longleftrightarrow b \neq 0 \wedge a = b$
⟨*proof*⟩

lemma *times-divide-times-eq*:
$(x \mathbin{/} y) * (z \mathbin{/} w) = (x * z) \mathbin{/} (y * w)$
⟨*proof*⟩

lemma *add-frac-num*:
$y \neq 0 \Longrightarrow x \mathbin{/} y + z = (x + z * y) \mathbin{/} y$
⟨*proof*⟩

lemma *add-num-frac*:

$y \neq 0 \Longrightarrow z + x \mathbin{/} y = (x + z * y) \mathbin{/} y$
⟨*proof*⟩

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*⟩

lemma *negative-imp-inverse-negative*:
$a < 0 \Longrightarrow$ *inverse* $a < 0$
⟨*proof*⟩

lemma *inverse-le-imp-le*:
assumes *invle*: *inverse* $a \leq$ *inverse* b **and** *apos*: $0 < a$
shows $b \leq a$
⟨*proof*⟩

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 <$ *inverse* a **and** *nz*: $a \neq 0$
shows $0 < a$
⟨*proof*⟩

lemma *inverse-negative-imp-negative*:
assumes *inv-less-0*: *inverse* $a < 0$ **and** *nz*: $a \neq 0$
shows $a < 0$
⟨*proof*⟩

lemma *linordered-field-no-lb*:
$\forall x.\ \exists y.\ y < x$
⟨*proof*⟩

lemma *linordered-field-no-ub*:
$\forall\ x.\ \exists y.\ y > x$
⟨*proof*⟩

lemma *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows *inverse* $b <$ *inverse* a
⟨*proof*⟩

lemma *inverse-less-imp-less*:
inverse $a <$ *inverse* $b \Longrightarrow 0 < a \Longrightarrow b < a$
⟨*proof*⟩

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*,*no-atp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a < inverse\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *le-imp-inverse-le*:
$a \leq b \Longrightarrow 0 < a \Longrightarrow inverse\ b \leq inverse\ a$
⟨*proof*⟩

lemma *inverse-le-iff-le* [*simp*,*no-atp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
$inverse\ a \leq inverse\ b \Longrightarrow b < 0 \Longrightarrow b \leq a$
⟨*proof*⟩

lemma *less-imp-inverse-less-neg*:
$a < b \Longrightarrow b < 0 \Longrightarrow inverse\ b < inverse\ a$
⟨*proof*⟩

lemma *inverse-less-imp-less-neg*:
$inverse\ a < inverse\ b \Longrightarrow b < 0 \Longrightarrow b < a$
⟨*proof*⟩

lemma *inverse-less-iff-less-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow inverse\ a < inverse\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *le-imp-inverse-le-neg*:
$a \leq b \Longrightarrow b < 0 \Longrightarrow inverse\ b \leq inverse\ a$
⟨*proof*⟩

lemma *inverse-le-iff-le-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

lemma *one-less-inverse*:
$0 < a \Longrightarrow a < 1 \Longrightarrow 1 < inverse\ a$
⟨*proof*⟩

lemma *one-le-inverse*:
$0 < a \Longrightarrow a \leq 1 \Longrightarrow 1 \leq inverse\ a$
⟨*proof*⟩

lemma *pos-le-divide-eq* [*field-simps*]: $0 < c \Longrightarrow (a \leq b/c) = (a*c \leq b)$
⟨*proof*⟩

lemma *neg-le-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow (a \leq b/c) = (b \leq a*c)$
⟨*proof*⟩

lemma *pos-less-divide-eq* [*field-simps*]:
$0 < c \Longrightarrow (a < b/c) = (a*c < b)$
⟨*proof*⟩

lemma *neg-less-divide-eq* [*field-simps*]:
$c < 0 \Longrightarrow (a < b/c) = (b < a*c)$
⟨*proof*⟩

lemma *pos-divide-less-eq* [*field-simps*]:
$0 < c \Longrightarrow (b/c < a) = (b < a*c)$
⟨*proof*⟩

lemma *neg-divide-less-eq* [*field-simps*]:
$c < 0 \Longrightarrow (b/c < a) = (a*c < b)$
⟨*proof*⟩

lemma *pos-divide-le-eq* [*field-simps*]: $0 < c \Longrightarrow (b/c \leq a) = (b \leq a*c)$
⟨*proof*⟩

lemma *neg-divide-le-eq* [*field-simps*]: $c < 0 \Longrightarrow (b/c \leq a) = (a*c \leq b)$
⟨*proof*⟩

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*:
$0 < x \Longrightarrow 0 < y \Longrightarrow 0 < x \mathbin{/} y$
⟨*proof*⟩

lemma *divide-nonneg-pos*:
$0 <= x \Longrightarrow 0 < y \Longrightarrow 0 <= x \mathbin{/} y$
⟨*proof*⟩

lemma *divide-neg-pos*:
$x < 0 \Longrightarrow 0 < y \Longrightarrow x \mathbin{/} y < 0$
⟨*proof*⟩

lemma *divide-nonpos-pos*:
$x <= 0 ==> 0 < y ==> x / y <= 0$
⟨*proof*⟩

lemma *divide-pos-neg*:
$0 < x ==> y < 0 ==> x / y < 0$
⟨*proof*⟩

lemma *divide-nonneg-neg*:
$0 <= x ==> y < 0 ==> x / y <= 0$
⟨*proof*⟩

lemma *divide-neg-neg*:
$x < 0 ==> y < 0 ==> 0 < x / y$
⟨*proof*⟩

lemma *divide-nonpos-neg*:
$x <= 0 ==> y < 0 ==> 0 <= x / y$
⟨*proof*⟩

lemma *divide-strict-right-mono*:
$[|a < b; 0 < c|] ==> a / c < b / c$
⟨*proof*⟩

lemma *divide-strict-right-mono-neg*:
$[|b < a; c < 0|] ==> a / c < b / c$
⟨*proof*⟩

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
$[|b < a; 0 < c; 0 < a*b|] ==> c / a < c / b$
⟨*proof*⟩

lemma *divide-left-mono*:
$[|b \leq a; 0 \leq c; 0 < a*b|] ==> c / a \leq c / b$
⟨*proof*⟩

lemma *divide-strict-left-mono-neg*:
$[|a < b; c < 0; 0 < a*b|] ==> c / a < c / b$
⟨*proof*⟩

lemma *mult-imp-div-pos-le*: $0 < y ==> x <= z * y ==>$
$x / y <= z$
⟨*proof*⟩

lemma *mult-imp-le-div-pos*: $0 < y ==> z * y <= x ==>$
$z <= x / y$
⟨*proof*⟩

lemma *mult-imp-div-pos-less*: $0 < y \Longrightarrow x < z * y \Longrightarrow x / y < z$
⟨*proof*⟩

lemma *mult-imp-less-div-pos*: $0 < y \Longrightarrow z * y < x \Longrightarrow z < x / y$
⟨*proof*⟩

lemma *frac-le*: $0 <= x \Longrightarrow x <= y \Longrightarrow 0 < w \Longrightarrow w <= z \Longrightarrow x / z <= y / w$
⟨*proof*⟩

lemma *frac-less*: $0 <= x \Longrightarrow x < y \Longrightarrow 0 < w \Longrightarrow w <= z \Longrightarrow x / z < y / w$
⟨*proof*⟩

lemma *frac-less2*: $0 < x \Longrightarrow x <= y \Longrightarrow 0 < w \Longrightarrow w < z \Longrightarrow x / z < y / w$
⟨*proof*⟩

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*: $a < b \Longrightarrow a < (a+b) / (1+1)$
⟨*proof*⟩

lemma *gt-half-sum*: $a < b \Longrightarrow (a+b)/(1+1) < b$
⟨*proof*⟩

subclass *dense-linorder*
⟨*proof*⟩

lemma *nonzero-abs-inverse*:
$a \neq 0 \Longrightarrow |inverse\ a| = inverse\ |a|$
⟨*proof*⟩

lemma *nonzero-abs-divide*:
$b \neq 0 \Longrightarrow |a / b| = |a| / |b|$
⟨*proof*⟩

lemma *field-le-epsilon*:
assumes e: $\bigwedge e.\ 0 < e \Longrightarrow x \leq y + e$
shows $x \leq y$
⟨*proof*⟩

end

class *linordered-field-inverse-zero* = *linordered-field* + *field-inverse-zero*

begin

lemma *le-divide-eq*:
$(a \leq b/c) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ a*c \leq b$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ b \leq a*c$
$\mathit{else}\ \ a \leq 0)$
⟨*proof*⟩

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 < \mathit{inverse}\ a) = (0 < a)$
⟨*proof*⟩

lemma *inverse-negative-iff-negative* [*simp*]:
$(\mathit{inverse}\ a < 0) = (a < 0)$
⟨*proof*⟩

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
$0 \leq \mathit{inverse}\ a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
$\mathit{inverse}\ a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *one-less-inverse-iff*:
$1 < \mathit{inverse}\ x \longleftrightarrow 0 < x \wedge x < 1$
⟨*proof*⟩

lemma *one-le-inverse-iff*:
$1 \leq \mathit{inverse}\ x \longleftrightarrow 0 < x \wedge x \leq 1$
⟨*proof*⟩

lemma *inverse-less-1-iff*:
$\mathit{inverse}\ x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$
⟨*proof*⟩

lemma *inverse-le-1-iff*:
$\mathit{inverse}\ x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$
⟨*proof*⟩

lemma *divide-le-eq*:
$(b/c \leq a) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ b \leq a*c$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ a*c \leq b$
$\mathit{else}\ 0 \leq a)$
⟨*proof*⟩

lemma *less-divide-eq*:

$(a < b/c) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ a*c < b$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ b < a*c$
$\mathit{else}\ \ a < 0)$
⟨*proof*⟩

lemma *divide-less-eq*:
$(b/c < a) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ b < a*c$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ a*c < b$
$\mathit{else}\ 0 < a)$
⟨*proof*⟩

Division and Signs

lemma *zero-less-divide-iff*:
$(0 < a/b) = (0 < a\ \&\ 0 < b \mid a < 0\ \&\ b < 0)$
⟨*proof*⟩

lemma *divide-less-0-iff*:
$(a/b < 0) =$
$(0 < a\ \&\ b < 0 \mid a < 0\ \&\ 0 < b)$
⟨*proof*⟩

lemma *zero-le-divide-iff*:
$(0 \leq a/b) =$
$(0 \leq a\ \&\ 0 \leq b \mid a \leq 0\ \&\ b \leq 0)$
⟨*proof*⟩

lemma *divide-le-0-iff*:
$(a/b \leq 0) =$
$(0 \leq a\ \&\ b \leq 0 \mid a \leq 0\ \&\ 0 \leq b)$
⟨*proof*⟩

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp*,*no-atp*]:
$(0 = 1/a) = (a = 0)$
⟨*proof*⟩

lemma *one-divide-eq-0-iff* [*simp*,*no-atp*]:
$(1/a = 0) = (a = 0)$
⟨*proof*⟩

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma *zero-le-divide-1-iff* [*simp*, *no-atp*]:
$0 \leq 1\ /\ a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-less-divide-1-iff* [*simp*, *no-atp*]:
$0 < 1 \,/\, a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *divide-le-0-1-iff* [*simp*, *no-atp*]:
$1 \,/\, a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *divide-less-0-1-iff* [*simp*, *no-atp*]:
$1 \,/\, a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *divide-right-mono*:
$[|a \leq b;\ 0 \leq c|] \Longrightarrow a/c \leq b/c$
⟨*proof*⟩

lemma *divide-right-mono-neg*: $a <= b$
$\Longrightarrow c <= 0 \Longrightarrow b \,/\, c <= a \,/\, c$
⟨*proof*⟩

lemma *divide-left-mono-neg*: $a <= b$
$\Longrightarrow c <= 0 \Longrightarrow 0 < a * b \Longrightarrow c \,/\, a <= c \,/\, b$
⟨*proof*⟩

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*no-atp*]:
$(1 \leq b \,/\, a) = ((0 < a \ \&\ a \leq b) \mid (a < 0 \ \&\ b \leq a))$
⟨*proof*⟩

lemma *divide-le-eq-1* [*no-atp*]:
$(b \,/\, a \leq 1) = ((0 < a \ \&\ b \leq a) \mid (a < 0 \ \&\ a \leq b) \mid a=0)$
⟨*proof*⟩

lemma *less-divide-eq-1* [*no-atp*]:
$(1 < b \,/\, a) = ((0 < a \ \&\ a < b) \mid (a < 0 \ \&\ b < a))$
⟨*proof*⟩

lemma *divide-less-eq-1* [*no-atp*]:
$(b \,/\, a < 1) = ((0 < a \ \&\ b < a) \mid (a < 0 \ \&\ a < b) \mid a=0)$
⟨*proof*⟩

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
⟨*proof*⟩

lemma *le-divide-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
⟨*proof*⟩

lemma *less-divide-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (1 < b/a) = (a < b)$
⟨*proof*⟩

lemma *less-divide-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (1 < b/a) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (b/a < 1) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b/a < 1 <-> a < b$
⟨*proof*⟩

lemma *eq-divide-eq-1* [*simp*,*no-atp*]:
$(1 = b/a) = ((a \neq 0 \ \&\ a = b))$
⟨*proof*⟩

lemma *divide-eq-eq-1* [*simp*,*no-atp*]:
$(b/a = 1) = ((a \neq 0 \ \&\ a = b))$
⟨*proof*⟩

lemma *abs-inverse* [*simp*]:
$|inverse\ a| =$
$inverse\ |a|$
⟨*proof*⟩

lemma *abs-divide* [*simp*]:
$|a\ /\ b| = |a|\ /\ |b|$
⟨*proof*⟩

lemma *abs-div-pos*: $0 < y ==>$
$|x|\ /\ y = |x\ /\ y|$
⟨*proof*⟩

lemma *field-le-mult-one-interval*:
assumes $*$: $\bigwedge z.\ [\![\ 0 < z\ ;\ z < 1\ ]\!] \Longrightarrow z * x \leq y$
shows $x \leq y$

⟨*proof*⟩

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*
⟨*proof*⟩

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 ⟨*proof*⟩

end

lemma *Suc-not-Zero*: *Suc m* ≠ *0*
⟨*proof*⟩

lemma *Zero-not-Suc*: *0* ≠ *Suc m*
⟨*proof*⟩

lemma *Suc-Rep-inject'*: *Suc-Rep x* = *Suc-Rep y* ⟷ *x* = *y*
⟨*proof*⟩

rep-datatype *0* :: *nat Suc*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Suc-neq-Zero*: *Suc m = 0 ⟹ R*
⟨*proof*⟩

lemma *Zero-neq-Suc*: *0 = Suc m ⟹ R*
⟨*proof*⟩

lemma *Suc-inject*: *Suc x = Suc y ⟹ x = y*
⟨*proof*⟩

lemma *n-not-Suc-n*: *n ≠ Suc n*
⟨*proof*⟩

lemma *Suc-n-not-n*: *Suc n ≠ n*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *add-Suc-right* [*simp*]: *m + Suc n* = *Suc* (*m + n*)
⟨*proof*⟩

declare *add-0* [*code*]

lemma *add-Suc-shift* [*code*]: *Suc m + n* = *m + Suc n*
⟨*proof*⟩

primrec *minus-nat*
where
diff-0: *m − 0* = (*m*::*nat*)
| *diff-Suc*: *m − Suc n* = (*case m − n of 0* => *0* | *Suc k* => *k*)

declare *diff-Suc* [*simp del*]
declare *diff-0* [*code*]

lemma *diff-0-eq-0* [*simp*, *code*]: $0 - n = (0::nat)$
⟨*proof*⟩

lemma *diff-Suc-Suc* [*simp*, *code*]: $Suc\ m - Suc\ n = m - n$
⟨*proof*⟩

instance ⟨*proof*⟩

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*]: $(m::nat) * 0 = 0$
⟨*proof*⟩

lemma *mult-Suc-right* [*simp*]: $m * Suc\ n = m + (m * n)$
⟨*proof*⟩

lemma *add-mult-distrib*: $(m + n) * k = (m * k) + ((n * k)::nat)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

15.3.1 Addition

lemma *nat-add-assoc*: $(m + n) + k = m + ((n + k)::nat)$
⟨*proof*⟩

lemma *nat-add-commute*: $m + n = n + (m::nat)$
⟨*proof*⟩

lemma *nat-add-left-commute*: $x + (y + z) = y + ((x + z)::nat)$

⟨*proof*⟩

lemma *nat-add-left-cancel* [*simp*]: $(k + m = k + n) = (m = (n::nat))$
⟨*proof*⟩

lemma *nat-add-right-cancel* [*simp*]: $(m + k = n + k) = (m=(n::nat))$
⟨*proof*⟩

Reasoning about $m + 0 = 0$, etc.

lemma *add-is-0* [*iff*]:
fixes *m n* :: *nat*
shows $(m + n = 0) = (m = 0 \ \& \ n = 0)$
⟨*proof*⟩

lemma *add-is-1*:
$(m+n= Suc\ 0) = (m= Suc\ 0 \ \& \ n=0 \mid m=0 \ \& \ n= Suc\ 0)$
⟨*proof*⟩

lemma *one-is-add*:
$(Suc\ 0 = m + n) = (m = Suc\ 0 \ \& \ n = 0 \mid m = 0 \ \& \ n = Suc\ 0)$
⟨*proof*⟩

lemma *add-eq-self-zero*:
fixes *m n* :: *nat*
shows $m + n = m \Longrightarrow n = 0$
⟨*proof*⟩

lemma *inj-on-add-nat*[*simp*]: *inj-on* $(\%n::nat.\ n+k)\ N$
⟨*proof*⟩

15.3.2 Difference

lemma *diff-self-eq-0* [*simp*]: $(m::nat) - m = 0$
⟨*proof*⟩

lemma *diff-diff-left*: $(i::nat) - j - k = i - (j + k)$
⟨*proof*⟩

lemma *Suc-diff-diff* [*simp*]: $(Suc\ m - n) - Suc\ k = m - n - k$
⟨*proof*⟩

lemma *diff-commute*: $(i::nat) - j - k = i - k - j$
⟨*proof*⟩

lemma *diff-add-inverse*: $(n + m) - n = (m::nat)$
⟨*proof*⟩

lemma *diff-add-inverse2*: $(m + n) - n = (m::nat)$
⟨*proof*⟩

lemma *diff-cancel*: $(k + m) - (k + n) = m - (n::nat)$
⟨*proof*⟩

lemma *diff-cancel2*: $(m + k) - (n + k) = m - (n::nat)$
⟨*proof*⟩

lemma *diff-add-0*: $n - (n + m) = (0::nat)$
⟨*proof*⟩

lemma *diff-Suc-1* [*simp*]: $Suc\ n - 1 = n$
⟨*proof*⟩

Difference distributes over multiplication

lemma *diff-mult-distrib*: $((m::nat) - n) * k = (m * k) - (n * k)$
⟨*proof*⟩

lemma *diff-mult-distrib2*: $k * ((m::nat) - n) = (k * m) - (k * n)$
⟨*proof*⟩

15.3.3 Multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
⟨*proof*⟩

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
⟨*proof*⟩

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
⟨*proof*⟩

lemma *mult-is-0* [*simp*]: $((m::nat) * n = 0) = (m=0 \mid n=0)$
⟨*proof*⟩

lemmas *nat-distrib* =
add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

lemma *mult-eq-1-iff* [*simp*]: $(m * n = Suc\ 0) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
⟨*proof*⟩

lemma *one-eq-mult-iff* [*simp*,*no-atp*]: $(Suc\ 0 = m * n) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
⟨*proof*⟩

lemma *nat-mult-eq-1-iff* [*simp*]: $m * n = (1::nat) \longleftrightarrow m = 1 \wedge n = 1$
⟨*proof*⟩

lemma *nat-1-eq-mult-iff* [*simp*]: $(1::nat) = m * n \longleftrightarrow m = 1 \wedge n = 1$
⟨*proof*⟩

lemma *mult-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
⟨*proof*⟩

lemma *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
⟨*proof*⟩

lemma *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
⟨*proof*⟩

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*]: $(0::nat) \leq n \longleftrightarrow True$ ⟨*proof*⟩
lemma *le0* [*iff*]: $0 \leq (n::nat)$ ⟨*proof*⟩

definition *less-nat* **where**
less-eq-Suc-le: $n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *Suc-le-eq* [*code*]: $Suc\ m \leq n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *le-0-eq* [*iff*]: $(n::nat) \leq 0 \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *not-less0* [*iff*]: $\neg\ n < (0::nat)$
⟨*proof*⟩

lemma *less-nat-zero-code* [*code*]: $n < (0::nat) \longleftrightarrow False$
⟨*proof*⟩

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$

⟨*proof*⟩

lemma *Suc-leD*: *Suc m* ≤ *n* ⟹ *m* ≤ *n*
⟨*proof*⟩

lemma *less-SucI*: *m* < *n* ⟹ *m* < *Suc n*
⟨*proof*⟩

lemma *Suc-lessD*: *Suc m* < *n* ⟹ *m* < *n*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *nat* :: *bot*
begin

definition *bot-nat* :: *nat* **where**
bot-nat = *0*

instance ⟨*proof*⟩

end

15.4.2 Introduction properties

lemma *lessI* [*iff*]: *n* < *Suc n*
⟨*proof*⟩

lemma *zero-less-Suc* [*iff*]: *0* < *Suc n*
⟨*proof*⟩

15.4.3 Elimination properties

lemma *less-not-refl*: ~ *n* < (*n*::*nat*)
⟨*proof*⟩

lemma *less-not-refl2*: *n* < *m* ==> *m* ≠ (*n*::*nat*)
⟨*proof*⟩

lemma *less-not-refl3*: (*s*::*nat*) < *t* ==> *s* ≠ *t*
⟨*proof*⟩

lemma *less-irrefl-nat*: (*n*::*nat*) < *n* ==> *R*
⟨*proof*⟩

lemma *less-zeroE*: (*n*::*nat*) < *0* ==> *R*
⟨*proof*⟩

lemma *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
⟨*proof*⟩

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
⟨*proof*⟩

lemma *less-one* [*iff*, *no-atp*]: $(n < (1::nat)) = (n = 0)$
⟨*proof*⟩

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
⟨*proof*⟩

"Less than" is antisymmetric, sort of

lemma *less-antisym*: $\llbracket \neg\ n < m;\ n < Suc\ m \rrbracket \Longrightarrow m = n$
⟨*proof*⟩

lemma *nat-neq-iff*: $((m::nat) \neq n) = (m < n \mid n < m)$
⟨*proof*⟩

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 $P\ n\ m$
⟨*proof*⟩

15.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \Longrightarrow Suc\ m \neq n \Longrightarrow Suc\ m < n$
⟨*proof*⟩

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*⟩

lemma *less-SucE*: **assumes** *major*: $m < Suc\ n$
and *less*: $m < n \Longrightarrow P$ **and** *eq*: $m = n \Longrightarrow P$ **shows** P
⟨*proof*⟩

lemma *Suc-lessE*: **assumes** *major*: $Suc\ i < k$
and *minor*: $!!j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$ **shows** P
⟨*proof*⟩

lemma *Suc-less-SucD*: $Suc\ m < Suc\ n \Longrightarrow m < n$
⟨*proof*⟩

lemma *less-trans-Suc*:
assumes *le*: $i < j$ **shows** $j < k \Longrightarrow Suc\ i < k$

⟨*proof*⟩

Can be used with *less-Suc-eq* to get $n = m \vee n < m$

lemma *not-less-eq*: $\neg\ m < n \longleftrightarrow n < Suc\ m$
⟨*proof*⟩

lemma *not-less-eq-eq*: $\neg\ m \leq n \longleftrightarrow Suc\ n \leq m$
⟨*proof*⟩

Properties of "less than or equal"

lemma *le-imp-less-Suc*: $m \leq n \Longrightarrow m < Suc\ n$
⟨*proof*⟩

lemma *Suc-n-not-le-n*: $\sim\ Suc\ n \leq n$
⟨*proof*⟩

lemma *le-Suc-eq*: $(m \leq Suc\ n) = (m \leq n \mid m = Suc\ n)$
⟨*proof*⟩

lemma *le-SucE*: $m \leq Suc\ n \Longrightarrow (m \leq n \Longrightarrow R) \Longrightarrow (m = Suc\ n \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *Suc-leI*: $m < n \Longrightarrow Suc(m) \leq n$
⟨*proof*⟩

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $Suc\ m \leq n \Longrightarrow m < n$
⟨*proof*⟩

lemma *less-imp-le-nat*: $m < n \Longrightarrow m \leq (n::nat)$
⟨*proof*⟩

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*: $m < n \mid m = n \Longrightarrow m \leq (n::nat)$
⟨*proof*⟩

lemma *le-eq-less-or-eq*: $(m \leq (n::nat)) = (m < n \mid m=n)$
⟨*proof*⟩

Useful with *blast*.

lemma *eq-imp-le*: $(m::nat) = n \Longrightarrow m \leq n$
⟨*proof*⟩

lemma *le-refl*: $n \leq (n::nat)$

⟨*proof*⟩

lemma *le-trans*: [| $i \leq j$; $j \leq k$ |] ==> $i \leq (k::nat)$
⟨*proof*⟩

lemma *le-antisym*: [| $m \leq n$; $n \leq m$ |] ==> $m = (n::nat)$
⟨*proof*⟩

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \;\&\; m \neq n)$
⟨*proof*⟩

lemma *le-neq-implies-less*: $(m::nat) \leq n$ ==> $m \neq n$ ==> $m < n$
⟨*proof*⟩

lemma *nat-le-linear*: $(m::nat) \leq n \mid n \leq m$
⟨*proof*⟩

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = nat$]

lemma *le-less-Suc-eq*: $m \leq n$ ==> $(n < Suc\ m) = (n = m)$
⟨*proof*⟩

lemma *not-less-less-Suc-eq*: ~ $n < m$ ==> $(n < Suc\ m) = (n = m)$
⟨*proof*⟩

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

These two rules ease the use of primitive recursion. NOTE USE OF ==

lemma *def-nat-rec-0*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ 0 = c$
⟨*proof*⟩

lemma *def-nat-rec-Suc*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ (Suc\ n) = h\ n\ (f\ n)$
⟨*proof*⟩

lemma *not0-implies-Suc*: $n \neq 0$ ==> $\exists m.\ n = Suc\ m$
⟨*proof*⟩

lemma *gr0-implies-Suc*: $n > 0$ ==> $\exists m.\ n = Suc\ m$
⟨*proof*⟩

lemma *gr-implies-not0*: **fixes** $n :: nat$ **shows** $m<n$ ==> $n \neq 0$
⟨*proof*⟩

lemma *neq0-conv*[*iff*]: **fixes** $n :: nat$ **shows** $(n \neq 0) = (0 < n)$
⟨*proof*⟩

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0$ ==> $False)$ ==> $0 < n$
⟨*proof*⟩

lemma *gr0-conv-Suc*: $(0 < n) = (\exists m.\ n = Suc\ m)$
⟨*proof*⟩

lemma *not-gr0* [*iff*,*no-atp*]: $!!n{::}nat.\ (\sim\ (0 < n)) = (n = 0)$
⟨*proof*⟩

lemma *Suc-le-D*: $(Suc\ n \leq m') ==> (?\ m.\ m' = Suc\ m)$
⟨*proof*⟩

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $(m < Suc\ n) = (m = 0 \mid (\exists j.\ m = Suc\ j\ \&\ j < n))$
⟨*proof*⟩

15.4.5 *min* and *max*

lemma *mono-Suc*: *mono Suc*
⟨*proof*⟩

lemma *min-0L* [*simp*]: $min\ 0\ n = (0{::}nat)$
⟨*proof*⟩

lemma *min-0R* [*simp*]: $min\ n\ 0 = (0{::}nat)$
⟨*proof*⟩

lemma *min-Suc-Suc* [*simp*]: $min\ (Suc\ m)\ (Suc\ n) = Suc\ (min\ m\ n)$
⟨*proof*⟩

lemma *min-Suc1*:
$min\ (Suc\ n)\ m = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc(min\ n\ m'))$
⟨*proof*⟩

lemma *min-Suc2*:
$min\ m\ (Suc\ n) = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc(min\ m'\ n))$
⟨*proof*⟩

lemma *max-0L* [*simp*]: $max\ 0\ n = (n{::}nat)$
⟨*proof*⟩

lemma *max-0R* [*simp*]: $max\ n\ 0 = (n{::}nat)$
⟨*proof*⟩

lemma *max-Suc-Suc* [*simp*]: $max\ (Suc\ m)\ (Suc\ n) = Suc(max\ m\ n)$
⟨*proof*⟩

lemma *max-Suc1*:
$max\ (Suc\ n)\ m = (case\ m\ of\ 0 => Suc\ n \mid Suc\ m' => Suc(max\ n\ m'))$
⟨*proof*⟩

lemma *max-Suc2*:
max m (Suc n) = (case m of 0 => Suc n | Suc m' => Suc(max m' n))
⟨*proof*⟩

15.4.6 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: *n>0 ==> Suc (n - Suc 0) = n*
⟨*proof*⟩

lemma *Suc-diff-1* [*simp*]: *0 < n ==> Suc (n - 1) = n*
⟨*proof*⟩

lemma *nat-add-left-cancel-le* [*simp*]: *(k + m ≤ k + n) = (m≤(n::nat))*
⟨*proof*⟩

lemma *nat-add-left-cancel-less* [*simp*]: *(k + m < k + n) = (m<(n::nat))*
⟨*proof*⟩

lemma *add-gr-0* [*iff*]: *!!m::nat. (m + n > 0) = (m>0 | n>0)*
⟨*proof*⟩

strict, in 1st argument

lemma *add-less-mono1*: *i < j ==> i + k < j + (k::nat)*
⟨*proof*⟩

strict, in both arguments

lemma *add-less-mono*: *[|i < j; k < l|] ==> i + k < j + (l::nat)*
⟨*proof*⟩

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

lemma *less-imp-Suc-add*: *m < n ==> (∃k. n = Suc (m + k))*
⟨*proof*⟩

strict, in 1st argument; proof is by induction on *k > 0*

lemma *mult-less-mono2*: *(i::nat) < j ==> 0<k ==> k * i < k * j*
⟨*proof*⟩

The naturals form an ordered *comm-semiring-1-cancel*

instance *nat* :: *linordered-semidom*
⟨*proof*⟩

instance *nat* :: *no-zero-divisors*
⟨*proof*⟩

lemma *nat-mult-1*: *(1::nat) * n = n*
⟨*proof*⟩

lemma *nat-mult-1-right*: *n * (1::nat) = n*
⟨*proof*⟩

15.4.7 Additional theorems about *op* ≤

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder* ⟨*proof*⟩

lemma *Least-Suc*:
[| *P n*; ~ *P 0* |] ==> (*LEAST n*. *P n*) = *Suc* (*LEAST m*. *P*(*Suc m*))
⟨*proof*⟩

lemma *Least-Suc2*:
[|*P n*; *Q m*; ~*P 0*; !*k*. *P* (*Suc k*) = *Q k*|] ==> *Least P* = *Suc* (*Least Q*)
⟨*proof*⟩

lemma *ex-least-nat-le*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃ *k*≤*n*. (∀ *i*<*k*. ¬*P i*) & *P*(*k*)
⟨*proof*⟩

lemma *ex-least-nat-less*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃ *k*<*n*. (∀ *i*≤*k*. ¬*P i*) & *P*(*k*+*1*)
⟨*proof*⟩

lemma *nat-less-induct*:
assumes !!*n*. ∀ *m*::*nat*. *m* < *n* −−> *P m* ==> *P n* **shows** *P n*
⟨*proof*⟩

lemma *measure-induct-rule* [*case-names less*]:
fixes *f* :: *'a* ⇒ *nat*
assumes *step*: ⋀*x*. (⋀*y*. *f y* < *f x* ⟹ *P y*) ⟹ *P x*
shows *P a*
⟨*proof*⟩

old style induction rules:

lemma *measure-induct*:
fixes *f* :: *'a* ⇒ *nat*
shows (⋀*x*. ∀ *y*. *f y* < *f x* ⟶ *P y* ⟹ *P x*) ⟹ *P a*
⟨*proof*⟩

lemma *full-nat-induct*:
assumes *step*: (!!*n*. (*ALL m*. *Suc m* <= *n* −−> *P m*) ==> *P n*)
shows *P n*
⟨*proof*⟩

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*⟩

The method of infinite descent, frequently used in number theory. Provided

by Roelof Oosterhuis. $P(n)$ is true for all $n \in \mathbb{N}$ if

- case "0": given $n = 0$ prove $P(n)$,
- case "smaller": given $n > 0$ and $\neg P(n)$ prove there exists a smaller integer m such that $\neg P(m)$.

A compact version without explicit base case:

lemma *infinite-descent*:
$[\![\; !!n{::}nat.\ \neg\ P\ n \Longrightarrow\ \exists\, m{<}n.\ \neg\ \ P\ m \;]\!] \Longrightarrow\ \ P\ n$
⟨*proof*⟩

lemma *infinite-descent0*[*case-names 0 smaller*]:
$[\![\; P\ 0;\ !!n.\ n{>}0 \Longrightarrow \neg\ P\ n \Longrightarrow (\exists\, m{::}nat.\ m < n \wedge \neg P\ m) \;]\!] \Longrightarrow P\ n$
⟨*proof*⟩

Infinite descent using a mapping to $\mathbb{N}$: $P(x)$ is true for all $x \in D$ if there exists a $V : D \to \mathbb{N}$ and

- case "0": given $V(x) = 0$ prove $P(x)$,
- case "smaller": given $V(x) > 0$ and $\neg P(x)$ prove there exists a $y \in D$ such that $V(y) < V(x)$ and $\neg P(y)$.

NB: the proof also shows how to use the previous lemma.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
assumes *A0*: $!!x.\ V\ x = (0{::}nat) \Longrightarrow P\ x$
and *A1*: $!!x.\ V\ x > 0 \Longrightarrow \neg P\ x \Longrightarrow (\exists\, y.\ V\ y < V\ x \wedge \neg P\ y)$
shows $P\ x$
⟨*proof*⟩

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*⟩

A [clumsy] way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
$[\![\; !!i\ j{::}nat.\ i < j \Longrightarrow f\ i < f\ j;\ i \leq j \;]\!] \Longrightarrow f\ i \leq ((f\ j){::}nat)$
⟨*proof*⟩

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j$ ==> $i + k \leq j + (k{::}nat)$
⟨*proof*⟩

non-strict, in both arguments

lemma *add-le-mono*: $[\!|\ i \leq j;\ \ k \leq l\ |\!]$ ==> $i + k \leq j + (l{::}nat)$

⟨*proof*⟩

lemma *le-add2*: $n \leq ((m + n)::nat)$
⟨*proof*⟩

lemma *le-add1*: $n \leq ((n + m)::nat)$
⟨*proof*⟩

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
⟨*proof*⟩

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
⟨*proof*⟩

lemma *less-iff-Suc-add*: $(m < n) = (\exists k.\ n = Suc\ (m + k))$
⟨*proof*⟩

lemma *trans-le-add1*: $(i::nat) \leq j \Longrightarrow i \leq j + m$
⟨*proof*⟩

lemma *trans-le-add2*: $(i::nat) \leq j \Longrightarrow i \leq m + j$
⟨*proof*⟩

lemma *trans-less-add1*: $(i::nat) < j \Longrightarrow i < j + m$
⟨*proof*⟩

lemma *trans-less-add2*: $(i::nat) < j \Longrightarrow i < m + j$
⟨*proof*⟩

lemma *add-lessD1*: $i + j < (k::nat) \Longrightarrow i < k$
⟨*proof*⟩

lemma *not-add-less1* [*iff*]: $\sim (i + j < (i::nat))$
⟨*proof*⟩

lemma *not-add-less2* [*iff*]: $\sim (j + i < (i::nat))$
⟨*proof*⟩

lemma *add-leD1*: $m + k \leq n \Longrightarrow m \leq (n::nat)$
⟨*proof*⟩

lemma *add-leD2*: $m + k \leq n \Longrightarrow k \leq (n::nat)$
⟨*proof*⟩

lemma *add-leE*: $(m::nat) + k \leq n \Longrightarrow (m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

needs $!!k$ for *add-ac* to work

lemma *less-add-eq-less*: $!!k::nat.\ k < l \Longrightarrow m + l = k + n \Longrightarrow m < n$

⟨*proof*⟩

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*: ~ $m < n \Longrightarrow n + (m - n) = (m::nat)$
⟨*proof*⟩

lemma *le-add-diff-inverse* [*simp*]: $n \leq m \Longrightarrow n + (m - n) = (m::nat)$
⟨*proof*⟩

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m \Longrightarrow (m - n) + n = (m::nat)$
⟨*proof*⟩

lemma *Suc-diff-le*: $n \leq m \Longrightarrow Suc\ m - n = Suc\ (m - n)$
⟨*proof*⟩

lemma *diff-less-Suc*: $m - n < Suc\ m$
⟨*proof*⟩

lemma *diff-le-self* [*simp*]: $m - n \leq (m::nat)$
⟨*proof*⟩

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k.\ n = m + k)$
⟨*proof*⟩

lemma *less-imp-diff-less*: $(j::nat) < k \Longrightarrow j - n < k$
⟨*proof*⟩

lemma *diff-Suc-less* [*simp*]: $0<n \Longrightarrow n - Suc\ i < n$
⟨*proof*⟩

lemma *diff-add-assoc*: $k \leq (j::nat) \Longrightarrow (i + j) - k = i + (j - k)$
⟨*proof*⟩

lemma *diff-add-assoc2*: $k \leq (j::nat) \Longrightarrow (j + i) - k = (j - k) + i$
⟨*proof*⟩

lemma *le-imp-diff-is-add*: $i \leq (j::nat) \Longrightarrow (j - i = k) = (j = k + i)$
⟨*proof*⟩

lemma *diff-is-0-eq* [*simp*]: $((m::nat) - n = 0) = (m \leq n)$
⟨*proof*⟩

lemma *diff-is-0-eq'* [*simp*]: $m \leq n \Longrightarrow (m::nat) - n = 0$
⟨*proof*⟩

lemma *zero-less-diff* [*simp*]: $(0 < n - (m::nat)) = (m < n)$
⟨*proof*⟩

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k::nat.\ 0 < k\ \&\ i + k = j$
⟨*proof*⟩

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*:
fixes $n\ m :: nat$
shows $n - m + m = max\ n\ m$
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

15.4.9 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \le (j::nat) \Longrightarrow i * k \le j * k$
⟨*proof*⟩

lemma *mult-le-mono2*: $i \le (j::nat) \Longrightarrow k * i \le k * j$
⟨*proof*⟩

$\le$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \le (j::nat) \Longrightarrow k \le l \Longrightarrow i * k \le j * l$
⟨*proof*⟩

lemma *mult-less-mono1*: $(i::nat) < j \Longrightarrow 0 < k \Longrightarrow i * k < j * k$
⟨*proof*⟩

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $(0 < (m::nat) * n) = (0 < m\ \&\ 0 < n)$
⟨*proof*⟩

lemma *one-le-mult-iff* [*simp*]: $(Suc\ 0 \le m * n) = (Suc\ 0 \le m\ \&\ Suc\ 0 \le n)$
⟨*proof*⟩

lemma *mult-less-cancel2* [*simp*]: $((m::nat) * k < n * k) = (0 < k\ \&\ m < n)$
⟨*proof*⟩

lemma *mult-less-cancel1* [*simp*]: $(k * (m::nat) < k * n) = (0 < k\ \&\ m < n)$

⟨*proof*⟩

lemma *mult-le-cancel1* [*simp*]: $(k * (m::nat) \leq k * n) = (0 < k \longrightarrow m \leq n)$
⟨*proof*⟩

lemma *mult-le-cancel2* [*simp*]: $((m::nat) * k \leq n * k) = (0 < k \longrightarrow m \leq n)$
⟨*proof*⟩

lemma *Suc-mult-less-cancel1*: $(Suc\ k * m < Suc\ k * n) = (m < n)$
⟨*proof*⟩

lemma *Suc-mult-le-cancel1*: $(Suc\ k * m \leq Suc\ k * n) = (m \leq n)$
⟨*proof*⟩

lemma *le-square*: $m \leq m * (m::nat)$
⟨*proof*⟩

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
⟨*proof*⟩

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n \Longrightarrow n = 1 \mid m = 0$
⟨*proof*⟩

the lattice order on *nat*

instantiation *nat* :: *distrib-lattice*
begin

definition
$(inf :: nat \Rightarrow nat \Rightarrow nat) = min$

definition
$(sup :: nat \Rightarrow nat \Rightarrow nat) = max$

instance ⟨*proof*⟩

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* :: $nat \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

abbreviation *compower* :: $('a \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a \Rightarrow 'b$ (**infixr** ^^ *80*) **where**
$f\ \hat{}\hat{}\ n \equiv compow\ n\ f$

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*
⟨*proof*⟩

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

lemma *funpow-add*:
f ^^ (*m* + *n*) = *f* ^^ *m* ∘ *f* ^^ *n*
⟨*proof*⟩

lemma *funpow-swap1*:
f ((*f* ^^ *n*) *x*) = (*f* ^^ *n*) (*f x*)
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *of-nat-add* [*simp*]: *of-nat* (*m* + *n*) = *of-nat m* + *of-nat n*
⟨*proof*⟩

lemma *of-nat-mult*: *of-nat* (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
⟨*proof*⟩

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* (λ*i*. *i* + *1*) *n 0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *of-nat-eq-0-iff* [*simp*, *no-atp*]: *of-nat m* = *0* ⟷ *m* = *0*
⟨*proof*⟩

lemma *inj-of-nat*: *inj of-nat*
⟨*proof*⟩

end

context *linordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0* ≤ *of-nat m*
⟨*proof*⟩

lemma *less-imp-of-nat-less*: *m* < *n* ⟹ *of-nat m* < *of-nat n*
⟨*proof*⟩

lemma *of-nat-less-imp-less*: *of-nat m* < *of-nat n* ⟹ *m* < *n*
⟨*proof*⟩

lemma *of-nat-less-iff* [*simp*]: *of-nat m* < *of-nat n* ⟷ *m* < *n*
⟨*proof*⟩

lemma *of-nat-le-iff* [*simp*]: *of-nat m* ≤ *of-nat n* ⟷ *m* ≤ *n*
⟨*proof*⟩

Every *linordered-semidom* has characteristic zero.

subclass *semiring-char-0*
⟨*proof*⟩

Special cases where either operand is zero

lemma *of-nat-0-le-iff* [*simp*]: *0* ≤ *of-nat n*
⟨*proof*⟩

lemma *of-nat-le-0-iff* [*simp*, *no-atp*]: *of-nat m* ≤ *0* ⟷ *m* = *0*
⟨*proof*⟩

lemma *of-nat-0-less-iff* [*simp*]: *0* < *of-nat n* ⟷ *0* < *n*
⟨*proof*⟩

lemma *of-nat-less-0-iff* [*simp*]: ¬ *of-nat m* < *0*
⟨*proof*⟩

end

context *ring-1*
begin

lemma *of-nat-diff*: *n* ≤ *m* ⟹ *of-nat* (*m* − *n*) = *of-nat m* − *of-nat n*
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *abs-of-nat* [*simp*]: |*of-nat n*| = *of-nat n*
⟨*proof*⟩

end

lemma *of-nat-id* [*simp*]: *of-nat n* = *n*
⟨*proof*⟩

lemma *of-nat-eq-id* [*simp*]: *of-nat* = *id*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *Nats-0* [*simp*]: *0* ∈ ℕ
⟨*proof*⟩

lemma *Nats-1* [*simp*]: *1* ∈ ℕ
⟨*proof*⟩

lemma *Nats-add* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* + *b* ∈ ℕ
⟨*proof*⟩

lemma *Nats-mult* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* ∗ *b* ∈ ℕ
⟨*proof*⟩

lemma *Nats-cases* [*cases set*: *Nats*]:
assumes *x* ∈ ℕ
obtains (*of-nat*) *n* **where** *x* = *of-nat n*
⟨*proof*⟩

lemma *Nats-induct* [*case-names of-nat*, *induct set*: *Nats*]:
x ∈ ℕ ⟹ (⋀*n*. *P* (*of-nat n*)) ⟹ *P x*
⟨*proof*⟩

end

15.8 Further Arithmetic Facts Concerning the Natural Numbers

lemma *subst-equals*:
assumes *1*: *t* = *s* **and** *2*: *u* = *t*
shows *u* = *s*
⟨*proof*⟩

⟨*ML*⟩

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*⟩

lemma *lift-Suc-mono-less*:
  assumes *mono*: !!n. $f\ n < f(Suc\ n)$ **and** $n < n'$
  shows $f\ n < f\ n'$
⟨*proof*⟩

lemma *lift-Suc-mono-less-iff*:
  $(!!n.\ f\ n < f(Suc\ n)) \Longrightarrow f(n) < f(m) \longleftrightarrow n<m$
⟨*proof*⟩

end

lemma *mono-iff-le-Suc*: $mono\ f = (\forall\, n.\ f\ n \leq f\ (Suc\ n))$
⟨*proof*⟩

lemma *mono-nat-linear-lb*:
  $(!!m\ n{::}nat.\ m<n \Longrightarrow f\ m < f\ n) \Longrightarrow f(m)+k \leq f(m+k)$
⟨*proof*⟩

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: $[|\ a < (b{::}nat);\ c \leq a\ |] ==> a-c < b-c$
⟨*proof*⟩

lemma *less-diff-conv*: $(i < j-k) = (i+k < (j{::}nat))$
⟨*proof*⟩

lemma *le-diff-conv*: $(j-k \leq (i{::}nat)) = (j \leq i+k)$
⟨*proof*⟩

lemma *le-diff-conv2*: $k \leq j ==> (i \leq j-k) = (i+k \leq (j{::}nat))$
⟨*proof*⟩

lemma *diff-diff-cancel* [*simp*]: $i \leq (n{::}nat) ==> n - (n - i) = i$
⟨*proof*⟩

lemma *le-add-diff*: $k \leq (n{::}nat) ==> m \leq n + m - k$
⟨*proof*⟩

lemma *diff-less*[*simp*]: !!$m{::}nat$. $[|\ 0<n;\ 0<m\ |] ==> m - n < m$
⟨*proof*⟩

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $((m-k) - (n-k)) = (m-n)$
⟨*proof*⟩

hide-fact (**open**) *diff-diff-eq*

lemma *eq-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k = n-k) = (m=n)$
⟨*proof*⟩

lemma *less-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k < n-k) = (m<n)$
⟨*proof*⟩

lemma *le-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k \leq n-k) = (m \leq n)$
⟨*proof*⟩

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat)$ ==> $(m-l) \leq (n-l)$
⟨*proof*⟩

lemma *diff-le-mono2*: $m \leq (n::nat)$ ==> $(l-n) \leq (l-m)$
⟨*proof*⟩

lemma *diff-less-mono2*: [| $m < (n::nat)$; $m<l$ |] ==> $(l-n) < (l-m)$
⟨*proof*⟩

lemma *diffs0-imp-equal*: !!$m::nat$. [| $m-n = 0$; $n-m = 0$ |] ==> $m=n$
⟨*proof*⟩

lemma *min-diff*: $min\ (m - (i::nat))\ (n - i) = min\ m\ n - i$
⟨*proof*⟩

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*⟩

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j$ --> $i - (j - k) = i + (k::nat) - j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j$ ==> $m - Suc\ (j - k) = m + k - Suc\ j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j$ ==> $Suc\ (j - k) - m = Suc\ j - (k + m)$
⟨*proof*⟩

Lemmas for ex/Factorization

lemma *one-less-mult*: [| $Suc\ 0 < n$; $Suc\ 0 < m$ |] ==> $Suc\ 0 < m*n$
⟨*proof*⟩

lemma *n-less-m-mult-n*: [| *Suc 0* < *n*; *Suc 0* < *m* |] ==> *n*<*m*∗*n*
⟨*proof*⟩

lemma *n-less-n-mult-m*: [| *Suc 0* < *n*; *Suc 0* < *m* |] ==> *n*<*n*∗*m*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *zero-induct-lemma*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P* (*k* − *i*)
⟨*proof*⟩

lemma *zero-induct*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P 0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dvd-1-iff-1* [*simp*]: (*m dvd Suc 0*) = (*m* = *Suc 0*)
⟨*proof*⟩

lemma *nat-dvd-1-iff-1* [*simp*]: *m dvd* (*1*::*nat*) ⟷ *m* = *1*
⟨*proof*⟩

lemma *dvd-antisym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
⟨*proof*⟩

op dvd is a partial order

interpretation *dvd*: *order op dvd* $\lambda n\ m :: nat.\ n\ dvd\ m \wedge \neg\ m\ dvd\ n$
⟨*proof*⟩

lemma *dvd-diff-nat*[*simp*]: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m*−*n* :: *nat*)
⟨*proof*⟩

lemma *dvd-diffD*: [| *k dvd m*−*n*; *k dvd n*; *n*≤*m* |] ==> *k dvd* (*m*::*nat*)
⟨*proof*⟩

lemma *dvd-diffD1*: [| *k dvd m*−*n*; *k dvd m*; *n*≤*m* |] ==> *k dvd* (*n*::*nat*)
⟨*proof*⟩

lemma *dvd-reduce*: (*k dvd n* + *k*) = (*k dvd* (*n*::*nat*))
⟨*proof*⟩

lemma *dvd-mult-cancel*: !!*k*::*nat*. [| *k*∗*m dvd k*∗*n*; *0*<*k* |] ==> *m dvd n*
⟨*proof*⟩

lemma *dvd-mult-cancel1*: *0*<*m* ==> (*m*∗*n dvd m*) = (*n* = (*1*::*nat*))
⟨*proof*⟩

lemma *dvd-mult-cancel2*: *0*<*m* ==> (*n*∗*m dvd m*) = (*n* = (*1*::*nat*))
⟨*proof*⟩

lemma *dvd-imp-le*: [| *k dvd n*; *0* < *n* |] ==> *k* ≤ (*n*::*nat*)
⟨*proof*⟩

lemma *nat-dvd-not-less*:
fixes *m n* :: *nat*
shows *0* < *m* ⟹ *m* < *n* ⟹ ¬ *n dvd m*
⟨*proof*⟩

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*)
⟨*proof*⟩

Datatypes will be represented by sets of type *node*

types 'a *item* = ('a, *unit*) *node set*
('a, 'b) *dtree* = ('a, 'b) *node set*

consts
Push :: [('b + *nat*), *nat* => ('b + *nat*)] => (*nat* => ('b + *nat*))

Push-Node :: [('b + *nat*), ('a, 'b) *node*] => ('a, 'b) *node*
ndepth :: ('a, 'b) *node* => *nat*

Atom :: ('a + *nat*) => ('a, 'b) *dtree*
Leaf :: 'a => ('a, 'b) *dtree*
Numb :: *nat* => ('a, 'b) *dtree*
Scons :: [('a, 'b) *dtree*, ('a, 'b) *dtree*] => ('a, 'b) *dtree*
In0 :: ('a, 'b) *dtree* => ('a, 'b) *dtree*
In1 :: ('a, 'b) *dtree* => ('a, 'b) *dtree*
Lim :: ('b => ('a, 'b) *dtree*) => ('a, 'b) *dtree*

ntrunc :: [*nat*, ('a, 'b) *dtree*] => ('a, 'b) *dtree*

uprod :: [('a, 'b) *dtree set*, ('a, 'b) *dtree set*]=> ('a, 'b) *dtree set*
usum :: [('a, 'b) *dtree set*, ('a, 'b) *dtree set*]=> ('a, 'b) *dtree set*

Split :: [[('a, 'b) *dtree*, ('a, 'b) *dtree*]=>'c, ('a, 'b) *dtree*] => 'c
Case :: [[('a, 'b) *dtree*]=>'c, [('a, 'b) *dtree*]=>'c, ('a, 'b) *dtree*] => 'c

dprod :: [(('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*, (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*]
=> (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*
dsum :: [(('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*, (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*]
=> (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*

defs

Push-Node-def: *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

Push-def: *Push* == (%*b h*. *nat-case b h*)

Atom-def: *Atom* == (%*x*. {*Abs-Node*((%*k*. *Inr 0*, *x*))})
Scons-def: *Scons M N* == (*Push-Node* (*Inr 1*) ‘ *M*) *Un* (*Push-Node* (*Inr* (*Suc 1*)) ‘ *N*)

Leaf-def: *Leaf* == *Atom o Inl*
Numb-def: *Numb* == *Atom o Inr*

In0-def: *In0*(*M*) == *Scons* (*Numb 0*) *M*
In1-def: *In1*(*M*) == *Scons* (*Numb 1*) *M*

Lim-def: *Lim f* == *Union* {*z*. *? x*. *z* = *Push-Node* (*Inl x*) ‘ (*f x*)}

ndepth-def: *ndepth*(*n*) == (%(*f*,*x*). *LEAST k*. *f k* = *Inr 0*) (*Rep-Node n*)
ntrunc-def: *ntrunc k N* == {*n*. *n*:*N* & *ndepth*(*n*)<*k*}

uprod-def: *uprod A B* == *UN x*:*A*. *UN y*:*B*. { *Scons x y* }
usum-def: *usum A B* == *In0*‘*A Un In1*‘*B*

Split-def: *Split c M* == *THE u*. *EX x y*. *M* = *Scons x y* & *u* = *c x y*

Case-def: *Case c d M* == *THE u*. (*EX x* . *M* = *In0*(*x*) & *u* = *c*(*x*))
| (*EX y* . *M* = *In1*(*y*) & *u* = *d*(*y*))

dprod-def: *dprod r s* == *UN* (*x*,*x*′):*r*. *UN* (*y*,*y*′):*s*. {(*Scons x y*, *Scons x*′ *y*′)}

dsum-def: *dsum r s* == (*UN* (*x*,*x*′):*r*. {(*In0*(*x*),*In0*(*x*′))}) *Un*
(*UN* (*y*,*y*′):*s*. {(*In1*(*y*),*In1*(*y*′))})

lemma *apfst-convE*:
[| $q = apfst\ f\ p$; !!$x\ y$. [| $p = (x,y)$; $q = (f(x),y)$ |] ==> R
|] ==> R
⟨*proof*⟩

lemma *Push-inject1*: $Push\ i\ f = Push\ j\ g$ ==> $i=j$
⟨*proof*⟩

lemma *Push-inject2*: $Push\ i\ f = Push\ j\ g$ ==> $f=g$
⟨*proof*⟩

lemma *Push-inject*:
[| $Push\ i\ f = Push\ j\ g$; [| $i=j$; $f=g$ |] ==> P |] ==> P
⟨*proof*⟩

lemma *Push-neq-K0*: $Push\ (Inr\ (Suc\ k))\ f = (\%z.\ Inr\ 0)$ ==> P
⟨*proof*⟩

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*, *standard*]

lemma *Node-K0-I*: $(\%k.\ Inr\ 0,\ a) : Node$
⟨*proof*⟩

lemma *Node-Push-I*: p: *Node* ==> $apfst\ (Push\ i)\ p : Node$
⟨*proof*⟩

16.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: $Scons\ M\ N \neq Atom(a)$
⟨*proof*⟩

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*, *standard*]

lemma *inj-Atom*: $inj(Atom)$
⟨*proof*⟩
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*, *standard*]

lemma *Atom-Atom-eq* [*iff*]: $(Atom(a)=Atom(b)) = (a=b)$
⟨*proof*⟩

lemma *inj-Leaf*: *inj*(*Leaf*)
⟨*proof*⟩

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*, *standard*]

lemma *inj-Numb*: *inj*(*Numb*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M*<=*M'*
⟨*proof*⟩

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M' N'* ==> *N*<=*N'*
⟨*proof*⟩

lemma *Scons-inject1*: *Scons M N* = *Scons M' N'* ==> *M*=*M'*
⟨*proof*⟩

lemma *Scons-inject2*: *Scons M N* = *Scons M' N'* ==> *N*=*N'*
⟨*proof*⟩

lemma *Scons-inject*:
[| *Scons M N* = *Scons M' N'*; [| *M*=*M'*; *N*=*N'* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N* = *Scons M' N'*) = (*M*=*M'* & *N*=*N'*)
⟨*proof*⟩

lemma *Scons-not-Leaf* [*iff*]: *Scons M N* ≠ *Leaf*(*a*)
⟨*proof*⟩

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*, *standard*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*, *standard*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*, *standard*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr 0*, *x*)) = *0*
⟨*proof*⟩

lemma *ndepth-Push-Node-aux*:
nat-case (*Inr* (*Suc i*)) *f k* = *Inr 0* −−> *Suc*(*LEAST x*. *f x* = *Inr 0*) <= *k*
⟨*proof*⟩

lemma *ndepth-Push-Node*:
ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
⟨*proof*⟩

lemma *ntrunc-0* [*simp*]: *ntrunc 0 M* = {}
⟨*proof*⟩

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
⟨*proof*⟩

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
⟨*proof*⟩

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
⟨*proof*⟩

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)

⟨*proof*⟩

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

16.3 Set Constructions

lemma *uprodI* [*intro*!]: [| *M*:*A*; *N*:*B* |] ==> *Scons M N* : *uprod A B*
⟨*proof*⟩

lemma *uprodE* [*elim*!]:
[| *c* : *uprod A B*;
!!*x y*. [| *x*:*A*; *y*:*B*; *c* = *Scons x y* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *uprodE2*: [| *Scons M N* : *uprod A B*; [| *M*:*A*; *N*:*B* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *usum-In0I* [*intro*]: *M*:*A* ==> *In0*(*M*) : *usum A B*
⟨*proof*⟩

lemma *usum-In1I* [*intro*]: *N*:*B* ==> *In1*(*N*) : *usum A B*
⟨*proof*⟩

lemma *usumE* [*elim*!]:
[| *u* : *usum A B*;
!!*x*. [| *x*:*A*; *u*=*In0*(*x*) |] ==> *P*;

!!y. [| y:B; u=*In1*(y) |] ==> P
|] ==> P
⟨*proof*⟩

lemma *In0-not-In1* [*iff*]: *In0*(M) ≠ *In1*(N)
⟨*proof*⟩

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*, *standard*]

lemma *In0-inject*: *In0*(M) = *In0*(N) ==> M=N
⟨*proof*⟩

lemma *In1-inject*: *In1*(M) = *In1*(N) ==> M=N
⟨*proof*⟩

lemma *In0-eq* [*iff*]: (*In0* M = *In0* N) = (M=N)
⟨*proof*⟩

lemma *In1-eq* [*iff*]: (*In1* M = *In1* N) = (M=N)
⟨*proof*⟩

lemma *inj-In0*: *inj In0*
⟨*proof*⟩

lemma *inj-In1*: *inj In1*
⟨*proof*⟩

lemma *Lim-inject*: *Lim* f = *Lim* g ==> f = g
⟨*proof*⟩

lemma *ntrunc-subsetI*: *ntrunc* k M <= M
⟨*proof*⟩

lemma *ntrunc-subsetD*: (!!k. *ntrunc* k M <= N) ==> M<=N
⟨*proof*⟩

lemma *ntrunc-equality*: (!!k. *ntrunc* k M = *ntrunc* k N) ==> M=N
⟨*proof*⟩

lemma *ntrunc-o-equality*:
[| !!k. (*ntrunc*(k) *o h1*) = (*ntrunc*(k) *o h2*) |] ==> *h1=h2*
⟨*proof*⟩

lemma *uprod-mono*: [| A<=A'; B<=B' |] ==> *uprod A B* <= *uprod A' B'*
⟨*proof*⟩

lemma *usum-mono*: [| A<=A'; B<=B' |] ==> *usum A B* <= *usum A' B'*
⟨*proof*⟩

lemma *Scons-mono*: [| M<=M'; N<=N' |] ==> *Scons M N* <= *Scons M' N'*
⟨*proof*⟩

lemma *In0-mono*: M<=N ==> *In0*(M) <= *In0*(N)
⟨*proof*⟩

lemma *In1-mono*: M<=N ==> *In1*(M) <= *In1*(N)
⟨*proof*⟩

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
⟨*proof*⟩

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = c(M)
⟨*proof*⟩

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = d(N)
⟨*proof*⟩

lemma *ntrunc-UN1*: *ntrunc k* (*UN x*. f(x)) = (*UN x*. *ntrunc k* (*f x*))
⟨*proof*⟩

lemma *Scons-UN1-x*: *Scons* (*UN x*. *f x*) *M* = (*UN x*. *Scons* (*f x*) *M*)
⟨*proof*⟩

lemma *Scons-UN1-y*: *Scons M* (*UN x*. *f x*) = (*UN x*. *Scons M* (*f x*))
⟨*proof*⟩

lemma *In0-UN1*: *In0*(*UN x*. f(x)) = (*UN x*. *In0*(f(x)))
⟨*proof*⟩

lemma *In1-UN1*: $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
⟨*proof*⟩

lemma *dprodI* [*intro*!]:
$[|\ (M,M'):r;\ \ (N,N'):s\ |] ==> (Scons\ M\ N,\ Scons\ M'\ N') : dprod\ r\ s$
⟨*proof*⟩

lemma *dprodE* [*elim*!]:
$[|\ c : dprod\ r\ s;$
$!!x\ y\ x'\ y'.\ [|\ (x,x') : r;\ \ (y,y') : s;$
$c = (Scons\ x\ y,\ Scons\ x'\ y')\ |] ==> P$
$|] ==> P$
⟨*proof*⟩

lemma *dsum-In0I* [*intro*]: $(M,M'):r ==> (In0(M),\ In0(M')) : dsum\ r\ s$
⟨*proof*⟩

lemma *dsum-In1I* [*intro*]: $(N,N'):s ==> (In1(N),\ In1(N')) : dsum\ r\ s$
⟨*proof*⟩

lemma *dsumE* [*elim*!]:
$[|\ w : dsum\ r\ s;$
$!!x\ x'.\ [|\ (x,x') : r;\ \ w = (In0(x),\ In0(x'))\ |] ==> P;$
$!!y\ y'.\ [|\ (y,y') : s;\ \ w = (In1(y),\ In1(y'))\ |] ==> P$
$|] ==> P$
⟨*proof*⟩

lemma *dprod-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dprod\ r\ s <= dprod\ r'\ s'$
⟨*proof*⟩

lemma *dsum-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dsum\ r\ s <= dsum\ r'\ s'$
⟨*proof*⟩

lemma *dprod-Sigma*: $(dprod\ (A <*> B)\ (C <*> D)) <= (uprod\ A\ C) <*> (uprod\ B\ D)$

⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *dsum-Sigma*: *(dsum (A <*> B) (C <*> D)) <= (usum A C) <*> (usum B D)*
⟨*proof*⟩

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*

⟨*ML*⟩

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 $O(log(n)\hat{}2)$ 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* $\circ$ *repr isom*

definition
iso-tuple-snd :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow$ $'a \Rightarrow 'c$ **where**
iso-tuple-snd isom = *snd* $\circ$ *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* $\circ$ *apfst f* $\circ$ *repr isom*

definition
iso-tuple-snd-update ::
$('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow$ $('c \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'a)$ **where**
iso-tuple-snd-update isom f = *abst isom* $\circ$ *apsnd f* $\circ$ *repr isom*

definition
iso-tuple-cons ::
$('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow$ $'b \Rightarrow 'c \Rightarrow 'a$ **where**
iso-tuple-cons isom = *curry* (*abst isom*)

17.3 Logical infrastructure for records

definition
iso-tuple-surjective-proof-assist :: $'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b) \Rightarrow$ *bool* **where**
iso-tuple-surjective-proof-assist x y f $\longleftrightarrow$ *f x* = *y*

definition
iso-tuple-update-accessor-cong-assist ::
$(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow$ *bool* **where**
iso-tuple-update-accessor-cong-assist upd acc $\longleftrightarrow$
$(\forall f\ v.\ upd\ (\lambda x.\ f\ (acc\ v))\ v = upd\ f\ v) \wedge (\forall v.\ upd\ id\ v = v)$

definition
iso-tuple-update-accessor-eq-assist ::
$(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow$ *bool*
where
iso-tuple-update-accessor-eq-assist upd acc v f v' x $\longleftrightarrow$
upd f v = *v'* $\wedge$ *acc v* = *x* $\wedge$ *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*: $\bigwedge v.\ v' = v \Longrightarrow f\ v = f'\ v$

shows *upd f r = upd f' r'*
⟨*proof*⟩

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'
⟨*proof*⟩

lemma *iso-tuple-update-accessor-cong-assist-id*:
iso-tuple-update-accessor-cong-assist upd acc ⟹ upd id = id
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *update-accessor-cong-assist-idI*:
iso-tuple-update-accessor-cong-assist id id
⟨*proof*⟩

lemma *update-accessor-cong-assist-triv*:
iso-tuple-update-accessor-cong-assist upd acc ⟹
iso-tuple-update-accessor-cong-assist upd acc
⟨*proof*⟩

lemma *update-accessor-accessor-eqE*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹ acc v = x
⟨*proof*⟩

lemma *update-accessor-updator-eqE*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹ upd f v = v'
⟨*proof*⟩

lemma *iso-tuple-update-accessor-eq-assist-idI*:
v' = f v ⟹ iso-tuple-update-accessor-eq-assist id id v f v' v
⟨*proof*⟩

lemma *iso-tuple-update-accessor-eq-assist-triv*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹
iso-tuple-update-accessor-eq-assist upd acc v f v' x

⟨*proof*⟩

lemma *iso-tuple-update-accessor-cong-from-eq*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹
iso-tuple-update-accessor-cong-assist upd acc
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assistI*:
f x = *y* ⟹ *iso-tuple-surjective-proof-assist x y f*
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assist-idE*:
iso-tuple-surjective-proof-assist x y id ⟹ *x* = *y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *abst-inj*: *abst isom x* = *abst isom y* ⟷ *x* = *y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *iso-tuple-cons-conj-eqI*:
a = *c* ∧ *b* = *d* ∧ *P* ⟷ *Q* ⟹
iso-tuple-cons isom a b = *iso-tuple-cons isom c d* ∧ *P* ⟷ *Q*
⟨*proof*⟩

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*: ⋀*x y*. *repr x* = *repr y* ⟷ *x* = *y*
and *abst-inv*: ⋀*z*. *repr* (*abst z*) = *z*
and *v*: *v* ≡ *Tuple-Isomorphism repr abst*

shows *isomorphic-tuple v*
⟨*proof*⟩

definition
tuple-iso-tuple ≡ *Tuple-Isomorphism id id*

lemma *tuple-iso-tuple*:
isomorphic-tuple tuple-iso-tuple
⟨*proof*⟩

lemma *refl-conj-eq*: $Q = R \Longrightarrow P \wedge Q \longleftrightarrow P \wedge R$
⟨*proof*⟩

lemma *iso-tuple-UNIV-I*: $x \in UNIV \equiv True$
⟨*proof*⟩

lemma *iso-tuple-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
⟨*proof*⟩

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
⟨*proof*⟩

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
⟨*proof*⟩

lemma *o-eq-dest-lhs*: $a\ o\ b = c \Longrightarrow a\ (b\ v) = c\ v$
⟨*proof*⟩

lemma *o-eq-id-dest*: $a\ o\ b = id\ o\ c \Longrightarrow a\ (b\ v) = c\ v$
⟨*proof*⟩

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* ((*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

⟨*ML*⟩

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* ⟨*proof*⟩

lemma *power-one* [*simp*]:
1 ^ n = 1
⟨*proof*⟩

lemma *power-one-right* [*simp*]:
a ^ 1 = a
⟨*proof*⟩

lemma *power-commutes*:
*a ^ n * a = a * a ^ n*
⟨*proof*⟩

lemma *power-Suc2*:
*a ^ Suc n = a ^ n * a*
⟨*proof*⟩

lemma *power-add*:
*a ^ (m + n) = a ^ m * a ^ n*
⟨*proof*⟩

lemma *power-mult*:
*a ^ (m * n) = (a ^ m) ^ n*
⟨*proof*⟩

end

context *comm-monoid-mult*
begin

lemma *power-mult-distrib*:
*(a * b) ^ n = (a ^ n) * (b ^ n)*
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *of-nat-power*:
of-nat (m ^ n) = of-nat m ^ n
⟨*proof*⟩

end

context *comm-semiring-1*
begin

The divides relation

lemma *le-imp-power-dvd*:
  assumes $m \leq n$ **shows** $a \mathbin{\hat{}} m \; dvd \; a \mathbin{\hat{}} n$
⟨*proof*⟩

lemma *power-le-dvd*:
  $a \mathbin{\hat{}} n \; dvd \; b \Longrightarrow m \leq n \Longrightarrow a \mathbin{\hat{}} m \; dvd \; b$
  ⟨*proof*⟩

lemma *dvd-power-same*:
  $x \; dvd \; y \Longrightarrow x \mathbin{\hat{}} n \; dvd \; y \mathbin{\hat{}} n$
  ⟨*proof*⟩

lemma *dvd-power-le*:
  $x \; dvd \; y \Longrightarrow m \geq n \Longrightarrow x \mathbin{\hat{}} n \; dvd \; y \mathbin{\hat{}} m$
  ⟨*proof*⟩

lemma *dvd-power* [*simp*]:
  assumes $n > (0::nat) \vee x = 1$
  shows $x \; dvd \; (x \mathbin{\hat{}} n)$
⟨*proof*⟩

end

context *ring-1*
begin

lemma *power-minus*:
  $(-\, a) \mathbin{\hat{}} n = (-\, 1) \mathbin{\hat{}} n * a \mathbin{\hat{}} n$
⟨*proof*⟩

end

context *linordered-semidom*
begin

lemma *zero-less-power* [*simp*]:
  $0 < a \Longrightarrow 0 < a \mathbin{\hat{}} n$
  ⟨*proof*⟩

lemma *zero-le-power* [*simp*]:
  $0 \leq a \Longrightarrow 0 \leq a \mathbin{\hat{}} n$
  ⟨*proof*⟩

lemma *one-le-power*[*simp*]:
  $1 \leq a \Longrightarrow 1 \leq a \mathbin{\hat{}} n$

⟨*proof*⟩

lemma *power-gt1-lemma*:
assumes *gt1*: $1 < a$
shows $1 < a * a \hat{} n$
⟨*proof*⟩

lemma *power-gt1*:
$1 < a \Longrightarrow 1 < a \hat{} Suc\ n$
⟨*proof*⟩

lemma *one-less-power* [*simp*]:
$1 < a \Longrightarrow 0 < n \Longrightarrow 1 < a \hat{} n$
⟨*proof*⟩

lemma *power-le-imp-le-exp*:
assumes *gt1*: $1 < a$
shows $a \hat{} m \leq a \hat{} n \Longrightarrow m \leq n$
⟨*proof*⟩

Surely we can strengthen this? It holds for $0<a<1$ too.

lemma *power-inject-exp* [*simp*]:
$1 < a \Longrightarrow a \hat{} m = a \hat{} n \longleftrightarrow m = n$
⟨*proof*⟩

Can relax the first premise to $(0::'a) < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*:
$1 < a \Longrightarrow a \hat{} m < a \hat{} n \Longrightarrow m < n$
⟨*proof*⟩

lemma *power-mono*:
$a \leq b \Longrightarrow 0 \leq a \Longrightarrow a \hat{} n \leq b \hat{} n$
⟨*proof*⟩

lemma *power-strict-mono* [*rule-format*]:
$a < b \Longrightarrow 0 \leq a \Longrightarrow 0 < n \longrightarrow a \hat{} n < b \hat{} n$
⟨*proof*⟩

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:
$0 < a \Longrightarrow a < 1 \Longrightarrow a * a \hat{} n < a \hat{} n$
⟨*proof*⟩

lemma *power-strict-decreasing* [*rule-format*]:
$n < N \Longrightarrow 0 < a \Longrightarrow a < 1 \longrightarrow a \hat{} N < a \hat{} n$
⟨*proof*⟩

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing* [*rule-format*]:

$n \leq N \Longrightarrow 0 \leq a \Longrightarrow a \leq 1 \longrightarrow a \hat{} N \leq a \hat{} n$
⟨*proof*⟩

lemma *power-Suc-less-one*:
$0 < a \Longrightarrow a < 1 \Longrightarrow a \hat{} Suc\ n < 1$
⟨*proof*⟩

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*⟩

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
$1 < a \Longrightarrow a \hat{} n < a * a \hat{} n$
⟨*proof*⟩

lemma *power-strict-increasing* [*rule-format*]:
$n < N \Longrightarrow 1 < a \longrightarrow a \hat{} n < a \hat{} N$
⟨*proof*⟩

lemma *power-increasing-iff* [*simp*]:
$1 < b \Longrightarrow b \hat{} x \leq b \hat{} y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *power-strict-increasing-iff* [*simp*]:
$1 < b \Longrightarrow b \hat{} x < b \hat{} y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *power-le-imp-le-base*:
assumes *le*: $a \hat{} Suc\ n \leq b \hat{} Suc\ n$
and *ynonneg*: $0 \leq b$
shows $a \leq b$
⟨*proof*⟩

lemma *power-less-imp-less-base*:
assumes *less*: $a \hat{} n < b \hat{} n$
assumes *nonneg*: $0 \leq b$
shows $a < b$
⟨*proof*⟩

lemma *power-inject-base*:
$a \hat{} Suc\ n = b \hat{} Suc\ n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a = b$
⟨*proof*⟩

lemma *power-eq-imp-eq-base*:
$a \hat{} n = b \hat{} n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < n \Longrightarrow a = b$
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *power-abs*:
$abs\ (a\ \hat{}\ n) = abs\ a\ \hat{}\ n$
⟨*proof*⟩

lemma *abs-power-minus* [*simp*]:
$abs\ ((-a)\ \hat{}\ n) = abs\ (a\ \hat{}\ n)$
⟨*proof*⟩

lemma *zero-less-power-abs-iff* [*simp*, *no-atp*]:
$0 < abs\ a\ \hat{}\ n \longleftrightarrow a \neq 0 \vee n = 0$
⟨*proof*⟩

lemma *zero-le-power-abs* [*simp*]:
$0 \leq abs\ a\ \hat{}\ n$
⟨*proof*⟩

end

context *ring-1-no-zero-divisors*
begin

lemma *field-power-not-zero*:
$a \neq 0 \Longrightarrow a\ \hat{}\ n \neq 0$
⟨*proof*⟩

end

context *division-ring*
begin

FIXME reorient or rename to *nonzero-inverse-power*

lemma *nonzero-power-inverse*:
$a \neq 0 \Longrightarrow inverse\ (a\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
⟨*proof*⟩

end

context *field*
begin

lemma *nonzero-power-divide*:
$b \neq 0 \Longrightarrow (a\ /\ b)\ \hat{}\ n = a\ \hat{}\ n\ /\ b\ \hat{}\ n$
⟨*proof*⟩

end

lemma *power-0-Suc* [*simp*]:
(*0*::'*a*::{*power*, *semiring-0*}) ^ *Suc n* = *0*
⟨*proof*⟩

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*}))
⟨*proof*⟩

lemma *power-eq-0-iff* [*simp*]:
a ^ *n* = *0* ⟷
a = (*0*::'*a*::{*mult-zero*,*zero-neq-one*,*no-zero-divisors*,*power*}) ∧ *n* ≠ *0*
⟨*proof*⟩

lemma (**in** *field*) *power-diff*:
assumes *nz*: *a* ≠ *0*
shows *n* ≤ *m* ⟹ *a* ^ (*m* − *n*) = *a* ^ *m* / *a* ^ *n*
⟨*proof*⟩

Perhaps these should be simprules.

lemma *power-inverse*:
fixes *a* :: '*a*::*division-ring-inverse-zero*
shows *inverse* (*a* ^ *n*) = *inverse a* ^ *n*
⟨*proof*⟩

lemma *power-one-over*:
1 / (*a*::'*a*::{*field-inverse-zero*, *power*}) ^ *n* = (*1* / *a*) ^ *n*
⟨*proof*⟩

lemma *power-divide*:
(*a* / *b*) ^ *n* = (*a*::'*a*::*field-inverse-zero*) ^ *n* / *b* ^ *n*
⟨*proof*⟩

18.2 Exponentiation for the Natural Numbers

lemma *nat-one-le-power* [*simp*]:
Suc 0 ≤ *i* ⟹ *Suc 0* ≤ *i* ^ *n*
⟨*proof*⟩

lemma *nat-zero-less-power-iff* [*simp*]:
x ^ *n* > *0* ⟷ *x* > (*0*::*nat*) ∨ *n* = *0*
⟨*proof*⟩

lemma *nat-power-eq-Suc-0-iff* [*simp*]:
x ^ *m* = *Suc 0* ⟷ *m* = *0* ∨ *x* = *Suc 0*
⟨*proof*⟩

lemma *power-Suc-0* [*simp*]:
Suc 0 ^ n = Suc 0
⟨*proof*⟩

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*⟩

lemma *power-dvd-imp-le*:
i ^ m dvd i ^ n ⟹ (1::nat) < i ⟹ m ≤ n
⟨*proof*⟩

18.3 Code generator tweak

lemma *power-power-power* [*code*, *code-unfold*, *code-inline del*]:
*power = power.power (1::'a::{power}) (op *)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *not-Some-eq* [*iff*]: *(ALL y. x ~= Some y) = (x = None)*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *UNIV-option-conv*: *UNIV = insert None* (*range Some*)
⟨*proof*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

lemma *elem-set* [*iff*]: (*x : set xo*) = (*xo = Some x*)
⟨*proof*⟩

lemma *set-empty-eq* [*simp*]: (*set xo* = {}) = (*xo = None*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *option-map-Some* [*simp*, *code*]: *map f* (*Some x*) = *Some* (*f x*)
⟨*proof*⟩

lemma *option-map-is-None* [*iff*]:
(*map f opt = None*) = (*opt = None*)
⟨*proof*⟩

lemma *option-map-eq-Some* [*iff*]:
(*map f xo* = *Some y*) = (*EX z*. *xo* = *Some z* & *f z* = *y*)
⟨*proof*⟩

lemma *option-map-comp*:
map f (*map g opt*) = *map* (*f o g*) *opt*
⟨*proof*⟩

lemma *option-map-o-sum-case* [*simp*]:
map f o sum-case g h = *sum-case* (*map f o g*) (*map f o h*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-none-none*:
is-none x ⟷ *x* = *None*
⟨*proof*⟩

lemma [*code-unfold*]:
eq-class.eq x None ⟷ *is-none x*
⟨*proof*⟩

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*⟩

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
finite F ==>
P {} ==> (!!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)) ==>
P F
— Discharging *x* ∉ *F* entails extra work.
⟨*proof*⟩

lemma *finite-ne-induct*[*case-names singleton insert*, *consumes 2*]:
assumes *fin*: *finite F* **shows** *F* ≠ {} ⟹
⟦ ⋀*x*. *P*{*x*};
⋀*x F*. ⟦ *finite F*; *F* ≠ {}; *x* ∉ *F*; *P F* ⟧ ⟹ *P* (*insert x F*) ⟧
⟹ *P F*
⟨*proof*⟩

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*⟩

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*⟩

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}
⟨*proof*⟩

lemma *nat-seg-image-imp-finite*:
$!!f\ A.\ A = f$ ‘ {i::*nat*. i<n} $\Longrightarrow$ *finite A*
⟨*proof*⟩

lemma *finite-conv-nat-seg-image*:
finite A = (∃ (n::*nat*) f. $A = f$ ‘ {i::*nat*. i<n})
⟨*proof*⟩

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*⟩

lemma *finite-Collect-less-nat*[*iff*]: *finite*{n::*nat*. n<k}
⟨*proof*⟩

Finiteness and set theoretic constructions

lemma *finite-UnI*: *finite F* ==> *finite G* ==> *finite* (*F Un G*)
⟨*proof*⟩

lemma *finite-subset*: $A \subseteq B$ ==> *finite B* ==> *finite A*
— Every subset of a finite set is finite.
⟨*proof*⟩

lemma *rev-finite-subset*: *finite B* ==> $A \subseteq B$ ==> *finite A*
⟨*proof*⟩

lemma *finite-Un* [*iff*]: *finite* (*F Un G*) = (*finite F* & *finite G*)
⟨*proof*⟩

lemma *finite-Collect-disjI*[*simp*]:

finite{*x*. *P x* | *Q x*} = (*finite*{*x*. *P x*} & *finite*{*x*. *Q x*})
⟨*proof*⟩

lemma *finite-Int* [*simp*, *intro*]: *finite F* | *finite G* ==> *finite* (*F Int G*)
— The converse obviously fails.
⟨*proof*⟩

lemma *finite-Collect-conjI* [*simp*, *intro*]:
finite{*x*. *P x*} | *finite*{*x*. *Q x*} ==> *finite*{*x*. *P x* & *Q x*}
— The converse obviously fails.
⟨*proof*⟩

lemma *finite-Collect-le-nat*[*iff*]: *finite*{*n*::*nat*. *n*<=*k*}
⟨*proof*⟩

lemma *finite-insert* [*simp*]: *finite* (*insert a A*) = *finite A*
⟨*proof*⟩

lemma *finite-Union*[*simp*, *intro*]:
⟦ *finite A*; !!*M*. *M* ∈ *A* ⟹ *finite M* ⟧ ⟹ *finite*(⋃ *A*)
⟨*proof*⟩

lemma *finite-Inter*[*intro*]: *EX A*:*M*. *finite*(*A*) ⟹ *finite*(*Inter M*)
⟨*proof*⟩

lemma *finite-INT*[*intro*]: *EX x*:*I*. *finite*(*A x*) ⟹ *finite*(*INT x*:*I*. *A x*)
⟨*proof*⟩

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*⟩

lemma *finite-Diff* [*simp*]: *finite A* ==> *finite* (*A* − *B*)
⟨*proof*⟩

lemma *finite-Diff2* [*simp*]:
assumes *finite B* **shows** *finite* (*A* − *B*) = *finite A*
⟨*proof*⟩

lemma *finite-compl*[*simp*]:
finite(*A*::'*a set*) ⟹ *finite*(−*A*) = *finite*(*UNIV*::'*a set*)
⟨*proof*⟩

lemma *finite-Collect-not*[*simp*]:
finite{*x*::'*a*. *P x*} ⟹ *finite*{*x*. ~*P x*} = *finite*(*UNIV*::'*a set*)
⟨*proof*⟩

lemma *finite-Diff-insert* [*iff*]: *finite* (*A* − *insert a B*) = *finite* (*A* − *B*)
⟨*proof*⟩

Image and Inverse Image over Finite Sets

lemma *finite-imageI*[*simp*]: *finite F* ==> *finite* (*h* ‘ *F*)
— The image of a finite set is finite.
⟨*proof*⟩

lemma *finite-image-set* [*simp*]:
finite {*x*. *P x*} ⟹ *finite* { *f x* | *x*. *P x* }
⟨*proof*⟩

lemma *finite-surj*: *finite A* ==> *B* <= *f* ‘ *A* ==> *finite B*
⟨*proof*⟩

lemma *finite-range-imageI*:
finite (*range g*) ==> *finite* (*range* (%*x*. *f* (*g x*)))
⟨*proof*⟩

lemma *finite-imageD*: *finite* (*f*‘*A*) ==> *inj-on f A* ==> *finite A*
⟨*proof*⟩

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.
⟨*proof*⟩

lemma *finite-vimageI*: [|*finite F*; *inj h*|] ==> *finite* (*h* −‘ *F*)
— The inverse image of a finite set under an injective function is finite.
⟨*proof*⟩

lemma *finite-vimageD*:
assumes *fin*: *finite* (*h* −‘ *F*) **and** *surj*: *surj h*
shows *finite F*
⟨*proof*⟩

lemma *finite-vimage-iff*: *bij h* ⟹ *finite* (*h* −‘ *F*) ⟷ *finite F*
⟨*proof*⟩

The finite UNION of finite sets

lemma *finite-UN-I*: *finite A* ==> (!!*a*. *a*:*A* ==> *finite* (*B a*)) ==> *finite* (*UN a*:*A*. *B a*)
⟨*proof*⟩

Strengthen RHS to (∀ *x*∈*A*. *finite* (*B x*)) ∧ *finite* {*x* ∈ *A*. *B x* ≠ {}}?
We'd need to prove *finite C* ⟹ ∀ *A B*. *UNION A B* ⊆ *C* ⟶ *finite* {*x* ∈ *A*. *B x* ≠ {}} by induction.

lemma *finite-UN* [*simp*]:
finite A ==> finite (UNION A B) = (ALL x:A. finite (B x))
⟨*proof*⟩

lemma *finite-Collect-bex*[*simp*]: *finite A ⟹*
finite{x. EX y:A. Q x y} = (ALL y:A. finite{x. Q x y})
⟨*proof*⟩

lemma *finite-Collect-bounded-ex*[*simp*]: *finite{y. P y} ⟹*
finite{x. EX y. P y & Q x y} = (ALL y. P y ⟶ finite{x. Q x y})
⟨*proof*⟩

lemma *finite-Plus*: [| *finite A*; *finite B* |] ==> *finite (A <+> B)*
⟨*proof*⟩

lemma *finite-PlusD*:
fixes *A* :: *'a set* **and** *B* :: *'b set*
assumes *fin*: *finite (A <+> B)*
shows *finite A finite B*
⟨*proof*⟩

lemma *finite-Plus-iff*[*simp*]: *finite (A <+> B) ⟷ finite A ∧ finite B*
⟨*proof*⟩

lemma *finite-Plus-UNIV-iff*[*simp*]:
finite (UNIV :: ('a + 'b) set) =
(finite (UNIV :: 'a set) & finite (UNIV :: 'b set))
⟨*proof*⟩

Sigma of finite sets

lemma *finite-SigmaI* [*simp*]:
finite A ==> (!!a. a:A ==> finite (B a)) ==> finite (SIGMA a:A. B a)
⟨*proof*⟩

lemma *finite-cartesian-product*: [| *finite A*; *finite B* |] ==>
finite (A <> B)*
⟨*proof*⟩

lemma *finite-Prod-UNIV*:
*finite (UNIV::'a set) ==> finite (UNIV::'b set) ==> finite (UNIV::('a * 'b) set)*
⟨*proof*⟩

lemma *finite-cartesian-productD1*:
[| *finite (A <*> B)*; *B ≠ {}* |] ==> *finite A*
⟨*proof*⟩

lemma *finite-cartesian-productD2*:

[| *finite* (*A* <*> *B*); *A* ≠ {} |] ==> *finite B*
⟨*proof*⟩

The powerset of a finite set

lemma *finite-Pow-iff* [*iff*]: *finite* (*Pow A*) = *finite A*
⟨*proof*⟩

lemma *finite-Collect-subsets*[*simp*,*intro*]: *finite A* ⟹ *finite*{*B*. *B* ⊆ *A*}
⟨*proof*⟩

lemma *finite-UnionD*: *finite*(⋃ *A*) ⟹ *finite A*
⟨*proof*⟩

lemma *finite-subset-image*:
assumes *finite B*
shows *B* ⊆ *f* ‘ *A* ⟹ ∃ *C*⊆*A*. *finite C* ∧ *B* = *f* ‘ *C*
⟨*proof*⟩

20.2 Class *finite*

class *finite* =
assumes *finite-UNIV*: *finite* (*UNIV* :: *'a set*)
begin

lemma *finite* [*simp*]: *finite* (*A* :: *'a set*)
⟨*proof*⟩

end

lemma *UNIV-unit* [*no-atp*]:
UNIV = {()} ⟨*proof*⟩

instance *unit* :: *finite* ⟨*proof*⟩

lemma *UNIV-bool* [*no-atp*]:
UNIV = {*False*, *True*} ⟨*proof*⟩

instance *bool* :: *finite* ⟨*proof*⟩

instance ∗ :: (*finite*, *finite*) *finite* ⟨*proof*⟩

lemma *finite-option-UNIV* [*simp*]:
finite (*UNIV* :: *'a option set*) = *finite* (*UNIV* :: *'a set*)
⟨*proof*⟩

instance *option* :: (*finite*) *finite* ⟨*proof*⟩

lemma *inj-graph*: *inj* (%*f*. {(*x*, *y*). *y* = *f x*})
⟨*proof*⟩

instance *fun* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

instance + :: (*finite*, *finite*) *finite* ⟨*proof*⟩

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*
⟨*proof*⟩

end

inductive *fold-graph* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a *set* ⇒ 'b ⇒ *bool*
for *f* :: 'a ⇒ 'b ⇒ 'b **and** *z* :: 'b **where**
emptyI [*intro*]: *fold-graph f z* {} *z* |
insertI [*intro*]: *x* ∉ *A* ⟹ *fold-graph f z A y*
⟹ *fold-graph f z* (*insert x A*) (*f x y*)

inductive-cases *empty-fold-graphE* [*elim*!]: *fold-graph f z* {} *x*

definition *fold* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a *set* ⇒ '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* ⟹ *x* ∈ *A* ⟹ *fold-graph f z A* (*f x y*)
⟨*proof*⟩

lemma *fold-graph-imp-finite*: *fold-graph f z A x* ⟹ *finite A*
⟨*proof*⟩

lemma *finite-imp-fold-graph*: *finite A* ⟹ ∃ *x*. *fold-graph f z A x*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *fold-graph-determ*:
fold-graph f z A x $\Longrightarrow$ *fold-graph f z A y* $\Longrightarrow$ $y = x$
⟨*proof*⟩

lemma *fold-equality*:
fold-graph f z A y $\Longrightarrow$ *fold f z A* $= y$
⟨*proof*⟩

lemma *fold-graph-fold*: *finite A* $\Longrightarrow$ *fold-graph f z A* (*fold f z A*)
⟨*proof*⟩

The base case for *fold*:

lemma (**in** −) *fold-empty* [*simp*]: *fold f z* $\{\} = z$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fold-fun-comm*:
finite A $\Longrightarrow$ *f x* (*fold f z A*) $=$ *fold f* (*f x z*) *A*
⟨*proof*⟩

lemma *fold-insert2*:
finite A $\Longrightarrow$ $x \notin A \Longrightarrow$ *fold f z* (*insert x A*) $=$ *fold f* (*f x z*) *A*
⟨*proof*⟩

lemma *fold-rec*:
assumes *finite A* **and** $x \in A$
shows *fold f z A* $=$ *f x* (*fold f z* $(A - \{x\})$)
⟨*proof*⟩

lemma *fold-insert-remove*:
assumes *finite A*
shows *fold f z* (*insert x A*) $=$ *f x* (*fold f z* $(A - \{x\})$)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fold-insert-idem*:
  assumes *fin*: *finite A*
  shows *fold f z* (*insert x A*) = *f x* (*fold f z A*)
⟨*proof*⟩

declare *fold-insert*[*simp del*] *fold-insert-idem*[*simp*]

lemma *fold-insert-idem2*:
  finite A ⟹ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
⟨*proof*⟩

end

20.3.2 Expressing set operations via *fold*

lemma (**in** *fun-left-comm*) *fun-left-comm-apply*:
  fun-left-comm (λ*x*. *f* (*g x*))
⟨*proof*⟩

lemma (**in** *fun-left-comm-idem*) *fun-left-comm-idem-apply*:
  fun-left-comm-idem (λ*x*. *f* (*g x*))
  ⟨*proof*⟩

lemma *fun-left-comm-idem-insert*:
  fun-left-comm-idem insert
⟨*proof*⟩

lemma *fun-left-comm-idem-remove*:
  fun-left-comm-idem (λ*x A*. *A* − {*x*})
⟨*proof*⟩

lemma (**in** *semilattice-inf*) *fun-left-comm-idem-inf*:
  fun-left-comm-idem inf
⟨*proof*⟩

lemma (**in** *semilattice-sup*) *fun-left-comm-idem-sup*:

fun-left-comm-idem sup
⟨*proof*⟩

lemma *union-fold-insert*:
assumes *finite A*
shows $A \cup B = fold\ insert\ B\ A$
⟨*proof*⟩

lemma *minus-fold-remove*:
assumes *finite A*
shows $B - A = fold\ (\lambda x\ A.\ A - \{x\})\ B\ A$
⟨*proof*⟩

context *complete-lattice*
begin

lemma *inf-Inf-fold-inf*:
assumes *finite A*
shows $inf\ B\ (Inf\ A) = fold\ inf\ B\ A$
⟨*proof*⟩

lemma *sup-Sup-fold-sup*:
assumes *finite A*
shows $sup\ B\ (Sup\ A) = fold\ sup\ B\ A$
⟨*proof*⟩

lemma *Inf-fold-inf*:
assumes *finite A*
shows $Inf\ A = fold\ inf\ top\ A$
⟨*proof*⟩

lemma *Sup-fold-sup*:
assumes *finite A*
shows $Sup\ A = fold\ sup\ bot\ A$
⟨*proof*⟩

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*⟩

lemma *sup-SUPR-fold-sup*:
assumes *finite A*
shows $sup\ B\ (SUPR\ A\ f) = fold\ (\lambda A.\ sup\ (f\ A))\ B\ A$ (**is** *?sup* = *?fold*)
⟨*proof*⟩

lemma *INFI-fold-inf*:
assumes *finite A*
shows $INFI\ A\ f = fold\ (\lambda A.\ inf\ (f\ A))\ top\ A$

⟨*proof*⟩

lemma *SUPR-fold-sup*:
assumes *finite A*
shows *SUPR A f* = *fold* (λ*A*. *sup* (*f A*)) *bot A*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *fold-image-cong*:
finite A ⟹
(!!*x*. *x*:*A* ==> *g x* = *h x*) ==> *fold-image times g z A* = *fold-image times h z A*
⟨*proof*⟩

end

context *comm-monoid-mult*
begin

lemma *fold-image-1*:
finite S ⟹ (∀ *x*∈*S*. *f x* = *1*) ⟹ *fold-image op* ∗ *f 1 S* = *1*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *fold-image-Un-one*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: ∀ *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*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

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*
⟨*proof*⟩

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)*
⟨*proof*⟩

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*⟩

lemma *fold-image-eq-general-inverses*:
assumes *fS*: *finite S*
and *kh*: $\bigwedge y.\ y \in T \Longrightarrow k\ y \in S \wedge h\ (k\ y) = y$
and *hk*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x\ \wedge g\ (h\ x) = f\ x$
shows *fold-image* $(op\ *)\ f\ e\ S =$ *fold-image* $(op\ *)\ g\ e\ T$

⟨*proof*⟩

end

20.5 A fold functional for non-empty sets

Does not require start value.

inductive
fold1Set :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow bool$
for f :: $'a \Rightarrow 'a \Rightarrow 'a$
where
fold1Set-insertI [*intro*]:
$[\![$ *fold-graph f a A x*; $a \notin A\ ]\!] \Longrightarrow$ *fold1Set f* (*insert a A*) *x*

definition *fold1* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a$ **where**
fold1 f A == *THE x. fold1Set f A x*

lemma *fold1Set-nonempty*:
fold1Set f A x $\Longrightarrow A \neq \{\}$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fold1-singleton* [*simp*]: *fold1 f* {*a*} = *a*
⟨*proof*⟩

lemma *finite-nonempty-imp-fold1Set*:
$[\![$ *finite A*; $A \neq \{\}\ ]\!] \Longrightarrow$ *EX x. fold1Set f A x*
⟨*proof*⟩

First, some lemmas about *fold-graph*.

context *ab-semigroup-mult*
begin

lemma *fun-left-comm*: *fun-left-comm*(*op* *)
⟨*proof*⟩

lemma *fold-graph-insert-swap*:
assumes *fold*: *fold-graph times* (*b*::'*a*) *A y* **and** $b \notin A$
shows *fold-graph times z* (*insert b A*) (*z* ∗ *y*)
⟨*proof*⟩

lemma *fold-graph-permute-diff*:
assumes *fold*: *fold-graph times b A x*
shows !!*a*. ⟦*a* ∈ *A*; *b* ∉ *A*⟧ ⟹ *fold-graph times a* (*insert b* (*A*−{*a*})) *x*
⟨*proof*⟩

lemma *fold1-eq-fold*:
assumes *finite A a* ∉ *A* **shows** *fold1 times* (*insert a A*) = *fold times a A*
⟨*proof*⟩

lemma *nonempty-iff*: (*A* ≠ {}) = (∃ *x B*. *A* = *insert x B* & *x* ∉ *B*)
⟨*proof*⟩

lemma *fold1-insert*:
assumes *nonempty*: *A* ≠ {} **and** *A*: *finite A x* ∉ *A*
shows *fold1 times* (*insert x A*) = *x* ∗ *fold1 times A*
⟨*proof*⟩

end

context *ab-semigroup-idem-mult*
begin

lemma *fun-left-comm-idem*: *fun-left-comm-idem*(*op* ∗)
⟨*proof*⟩

lemma *fold1-insert-idem* [*simp*]:
assumes *nonempty*: *A* ≠ {} **and** *A*: *finite A*
shows *fold1 times* (*insert x A*) = *x* ∗ *fold1 times A*
⟨*proof*⟩

lemma *hom-fold1-commute*:
assumes *hom*: !!*x y*. *h* (*x* ∗ *y*) = *h x* ∗ *h y*
and *N*: *finite N N* ≠ {} **shows** *h* (*fold1 times N*) = *fold1 times* (*h* ‘ *N*)
⟨*proof*⟩

lemma *fold1-eq-fold-idem*:
assumes *finite A*
shows *fold1 times* (*insert a A*) = *fold times a A*
⟨*proof*⟩

end

Now the recursion rules for definitions:

lemma *fold1-singleton-def*: *g* = *fold1 f* ⟹ *g* {*a*} = *a*

⟨*proof*⟩

lemma (**in** *ab-semigroup-mult*) *fold1-insert-def*:
⟦ *g* = *fold1 times*; *finite A*; *x* ∉ *A*; *A* ≠ {} ⟧ ⟹ *g* (*insert x A*) = *x* ∗ *g A*
⟨*proof*⟩

lemma (**in** *ab-semigroup-idem-mult*) *fold1-insert-idem-def*:
⟦ *g* = *fold1 times*; *finite A*; *A* ≠ {} ⟧ ⟹ *g* (*insert x A*) = *x* ∗ *g A*
⟨*proof*⟩

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* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹ *A Int B* = {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
⟨*proof*⟩

lemma *fold1-in*:
assumes *A*: *finite* (*A*) *A* ≠ {} **and** *elem*: ⋀*x y*. *x* ∗ *y* ∈ {*x*,*y*}
shows *fold1 times A* ∈ *A*
⟨*proof*⟩

end

lemma (**in** *ab-semigroup-idem-mult*) *fold1-Un2*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
⟨*proof*⟩

20.6 Locales as mini-packages for fold operations

20.6.1 The natural case

locale *folding* =
fixes *f* :: *'a* ⇒ *'b* ⇒ *'b*
fixes *F* :: *'a set* ⇒ *'b* ⇒ *'b*
assumes *commute-comp*: *f y* ∘ *f x* = *f x* ∘ *f y*
assumes *eq-fold*: *finite A* ⟹ *F A s* = *fold f s A*

begin

lemma *empty* [*simp*]:
$F\ \{\} = id$
⟨*proof*⟩

lemma *insert* [*simp*]:
assumes *finite* A **and** $x \notin A$
shows $F\ (insert\ x\ A) = F\ A \circ f\ x$
⟨*proof*⟩

lemma *remove*:
assumes *finite* A **and** $x \in A$
shows $F\ A = F\ (A - \{x\}) \circ f\ x$
⟨*proof*⟩

lemma *insert-remove*:
assumes *finite* A
shows $F\ (insert\ x\ A) = F\ (A - \{x\}) \circ f\ x$
⟨*proof*⟩

lemma *commute-left-comp*:
$f\ y \circ (f\ x \circ g) = f\ x \circ (f\ y \circ g)$
⟨*proof*⟩

lemma *commute-comp'*:
assumes *finite* A
shows $f\ x \circ F\ A = F\ A \circ f\ x$
⟨*proof*⟩

lemma *commute-left-comp'*:
assumes *finite* A
shows $f\ x \circ (F\ A \circ g) = F\ A \circ (f\ x \circ g)$
⟨*proof*⟩

lemma *commute-comp''*:
assumes *finite* A **and** *finite* B
shows $F\ B \circ F\ A = F\ A \circ F\ B$
⟨*proof*⟩

lemma *commute-left-comp''*:
assumes *finite* A **and** *finite* B
shows $F\ B \circ (F\ A \circ g) = F\ A \circ (F\ B \circ g)$
⟨*proof*⟩

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 $F\ (A \cup B) \circ F\ (A \cap B) = F\ A \circ F\ B$
⟨*proof*⟩

lemma *union*:
assumes *finite A* **and** *finite B*
and $A \cap B = \{\}$
shows $F\ (A \cup B) = F\ A \circ F\ B$
⟨*proof*⟩

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*:
$f\ x \circ (f\ x \circ g) = f\ x \circ g$
⟨*proof*⟩

lemma *in-comp-idem*:
assumes *finite A* **and** $x \in A$
shows $F\ A \circ f\ x = F\ A$
⟨*proof*⟩

lemma *subset-comp-idem*:
assumes *finite A* **and** $B \subseteq A$
shows $F\ A \circ F\ B = F\ A$
⟨*proof*⟩

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
assumes *finite A*
shows $F\ (insert\ x\ A) = F\ A \circ f\ x$
⟨*proof*⟩

lemma *union-idem*:
assumes *finite A* **and** *finite B*
shows $F\ (A \cup B) = F\ A \circ F\ B$
⟨*proof*⟩

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* $A \Longrightarrow F\ A = fold\text{-}image\ f\ g\ 1\ A$
begin

lemma *empty* [*simp*]:
  $F\ \{\} = 1$
  ⟨*proof*⟩

lemma *insert* [*simp*]:
  assumes *finite* A **and** $x \notin A$
  shows $F\ (insert\ x\ A) = g\ x * F\ A$
⟨*proof*⟩

lemma *remove*:
  assumes *finite* A **and** $x \in A$
  shows $F\ A = g\ x * F\ (A - \{x\})$
⟨*proof*⟩

lemma *insert-remove*:
  assumes *finite* A
  shows $F\ (insert\ x\ A) = g\ x * F\ (A - \{x\})$
  ⟨*proof*⟩

lemma *neutral*:
  assumes *finite* A **and** $\forall x{\in}A.\ g\ x = 1$
  shows $F\ A = 1$
  ⟨*proof*⟩

lemma *union-inter*:
  assumes *finite* A **and** *finite* B
  shows $F\ (A \cup B) * F\ (A \cap B) = F\ A * F\ B$
⟨*proof*⟩

corollary *union-inter-neutral*:
  assumes *finite* A **and** *finite* B
  and *I0*: $\forall x \in A{\cap}B.\ g\ x = 1$
  shows $F\ (A \cup B) = F\ A * F\ B$
  ⟨*proof*⟩

corollary *union-disjoint*:
  assumes *finite* A **and** *finite* B
  assumes $A \cap B = \{\}$
  shows $F\ (A \cup B) = F\ A * F\ B$
  ⟨*proof*⟩

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*⟩

context *folding-image*
begin

lemma *reindex*:
  assumes *finite A* **and** *inj-on h A*
  shows $F\ g\ (h\ `\ A) = F\ (g \circ h)\ A$
  ⟨*proof*⟩

lemma *cong*:
  assumes *finite A* **and** $\bigwedge x.\ x \in A \Longrightarrow g\ x = h\ x$
  shows $F\ g\ A = F\ h\ A$
⟨*proof*⟩

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 $F\ g\ (UNION\ I\ A) = F\ (F\ g \circ A)\ I$
⟨*proof*⟩

lemma *distrib*:
  assumes *finite A*
  shows $F\ (\lambda x.\ g\ x * h\ x)\ A = F\ g\ A * F\ h\ A$
  ⟨*proof*⟩

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 $R\ (F\ h\ S)\ (F\ g\ S)$
  ⟨*proof*⟩

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*⟩

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 $F\ j\ S = F\ g\ T$

⟨*proof*⟩

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*⟩

context *folding-image-simple-idem*
begin

lemma *in-idem*:
assumes *finite* A **and** $x \in A$
shows $g\ x * F\ A = F\ A$
⟨*proof*⟩

lemma *subset-idem*:
assumes *finite* A **and** $B \subseteq A$
shows $F\ B * F\ A = F\ A$
⟨*proof*⟩

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
assumes *finite* A
shows $F\ (\mathit{insert}\ x\ A) = g\ x * F\ A$
⟨*proof*⟩

lemma *union-idem*:
assumes *finite* A **and** *finite* B
shows $F\ (A \cup B) = F\ A * F\ B$
⟨*proof*⟩

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*⟩

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*]:
$F\ \{x\} = x$
⟨*proof*⟩

lemma *eq-fold'*:
assumes *finite* A **and** $x \notin A$
shows $F\ (insert\ x\ A) = fold\ (op\ *)\ x\ A$
⟨*proof*⟩

lemma *insert* [*simp*]:
assumes *finite* A **and** $x \notin A$ **and** $A \neq \{\}$
shows $F\ (insert\ x\ A) = x * F\ A$
⟨*proof*⟩
thm *fold.commute-comp'* [*of* B b, *simplified expand-fun-eq*, *simplified*]
⟨*proof*⟩

lemma *remove*:
assumes *finite* A **and** $x \in A$
shows $F\ A = (if\ A - \{x\} = \{\}\ then\ x\ else\ x * F\ (A - \{x\}))$
⟨*proof*⟩

lemma *insert-remove*:
assumes *finite* A
shows $F\ (insert\ x\ A) = (if\ A - \{x\} = \{\}\ then\ x\ else\ x * F\ (A - \{x\}))$
⟨*proof*⟩

lemma *union-disjoint*:
assumes *finite* A $A \neq \{\}$ **and** *finite* B $B \neq \{\}$ **and** $A \cap B = \{\}$
shows $F\ (A \cup B) = F\ A * F\ B$
⟨*proof*⟩

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*⟩

lemma *closed*:
assumes *finite* A $A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x * y \in \{x, y\}$
shows $F\ A \in A$
⟨*proof*⟩

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*⟩

context *folding-one-idem*
begin

lemma *in-idem*:
  assumes *finite* A **and** $x \in A$
  shows $x * F\ A = F\ A$
⟨*proof*⟩

lemma *subset-idem*:
  assumes *finite* A $B \neq \{\}$ **and** $B \subseteq A$
  shows $F\ B * F\ A = F\ A$
⟨*proof*⟩

lemma *eq-fold-idem'*:
  assumes *finite* A
  shows $F\ (insert\ a\ A) = fold\ (op\ *)\ a\ A$
⟨*proof*⟩

lemma *insert-idem* [*simp*]:
  assumes *finite* A **and** $A \neq \{\}$
  shows $F\ (insert\ x\ A) = x * F\ A$
⟨*proof*⟩

lemma *union-idem*:
  assumes *finite* A $A \neq \{\}$ **and** *finite* B $B \neq \{\}$
  shows $F\ (A \cup B) = F\ A * F\ B$
⟨*proof*⟩

lemma *hom-commute*:
  assumes *hom*: $\bigwedge x\ y.\ h\ (x * y) = h\ x * h\ y$
  and N: *finite* N $N \neq \{\}$ **shows** $h\ (F\ N) = F\ (h\ `\ N)$
⟨*proof*⟩

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. $A = \{f\ i \mid i.\ i < n\}$. 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*⟩

lemma *card-infinite* [*simp*]:
¬ *finite A* ⟹ *card A* = *0*
⟨*proof*⟩

lemma *card-empty*:
card {} = *0*
⟨*proof*⟩

lemma *card-insert-disjoint*:
finite A ==> *x* ∉ *A* ==> *card* (*insert x A*) = *Suc* (*card A*)
⟨*proof*⟩

lemma *card-insert-if*:
finite A ==> *card* (*insert x A*) = (*if x* ∈ *A then card A else Suc* (*card A*))
⟨*proof*⟩

lemma *card-ge-0-finite*:
card A > *0* ⟹ *finite A*
⟨*proof*⟩

lemma *card-0-eq* [*simp*, *no-atp*]:
finite A ⟹ *card A* = *0* ⟷ *A* = {}
⟨*proof*⟩

lemma *finite-UNIV-card-ge-0*:
finite (*UNIV* :: *'a set*) ⟹ *card* (*UNIV* :: *'a set*) > *0*
⟨*proof*⟩

lemma *card-eq-0-iff*:
card A = *0* ⟷ *A* = {} ∨ ¬ *finite A*
⟨*proof*⟩

lemma *card-gt-0-iff*:
0 < *card A* ⟷ *A* ≠ {} ∧ *finite A*
⟨*proof*⟩

lemma *card-Suc-Diff1*: *finite A* ==> *x*: *A* ==> *Suc* (*card* (*A* − {*x*})) = *card A*
⟨*proof*⟩

lemma *card-Diff-singleton*:
finite A ==> *x*: *A* ==> *card* (*A* − {*x*}) = *card A* − *1*
⟨*proof*⟩

lemma *card-Diff-singleton-if*:
finite A ==> card (A−{x}) = (if x : A then card A − 1 else card A)
⟨*proof*⟩

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*⟩

lemma *card-insert*: *finite A ==> card (insert x A) = Suc (card (A − {x}))*
⟨*proof*⟩

lemma *card-insert-le*: *finite A ==> card A <= card (insert x A)*
⟨*proof*⟩

lemma *card-mono*:
assumes *finite B* **and** *A* ⊆ *B*
shows *card A* ≤ *card B*
⟨*proof*⟩

lemma *card-seteq*: *finite B ==>* (!!*A*. *A <= B ==> card B <= card A ==> A = B*)
⟨*proof*⟩

lemma *psubset-card-mono*: *finite B ==> A < B ==> card A < card B*
⟨*proof*⟩

lemma *card-Un-Int*: *finite A ==> finite B*
==> card A + card B = card (A Un B) + card (A Int B)
⟨*proof*⟩

lemma *card-Un-disjoint*: *finite A ==> finite B*
==> A Int B = {} ==> card (A Un B) = card A + card B
⟨*proof*⟩

lemma *card-Diff-subset*:
assumes *finite B* **and** *B* ⊆ *A*
shows *card (A − B) = card A − card B*
⟨*proof*⟩

lemma *card-Diff-subset-Int*:
assumes *AB*: *finite* (*A* ∩ *B*) **shows** *card (A − B) = card A − card* (*A* ∩ *B*)
⟨*proof*⟩

lemma *card-Diff1-less*: *finite A ==> x*: *A ==> card (A − {x}) < card A*
⟨*proof*⟩

lemma *card-Diff2-less*:
finite A ==> x: *A ==> y*: *A ==> card (A − {x} − {y}) < card A*

⟨*proof*⟩

lemma *card-Diff1-le*: *finite A* ==> *card* $(A - \{x\})$ <= *card A*
⟨*proof*⟩

lemma *card-psubset*: *finite B* ==> $A \subseteq B$ ==> *card A* < *card B* ==> $A < B$
⟨*proof*⟩

lemma *insert-partition*:
$[\![x \notin F; \forall c1 \in insert\ x\ F.\ \forall c2 \in insert\ x\ F.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}]\!]$
$\Longrightarrow x \cap \bigcup F = \{\}$
⟨*proof*⟩

lemma *finite-psubset-induct*[*consumes 1*, *case-names psubset*]:
assumes *fin*: *finite A*
and *major*: $\bigwedge A.$ *finite A* $\Longrightarrow (\bigwedge B.\ B \subset A \Longrightarrow P\ B) \Longrightarrow P\ A$
shows *P A*
⟨*proof*⟩

main cardinality theorem

lemma *card-partition* [*rule-format*]:
finite C ==>
finite $(\bigcup C)$ -->
$(\forall c \in C.$ *card c* = *k*$)$ -->
$(\forall c1 \in C.\ \forall c2 \in C.\ c1 \neq c2$ --> $c1 \cap c2 = \{\}) -->$
k * *card*(*C*) = *card* $(\bigcup C)$
⟨*proof*⟩

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*⟩

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*⟩

lemma *card-Suc-eq*:
(*card A* = *Suc k*) =
($\exists b\ B.$ *A* = *insert b B* & $b \notin B$ & *card B* = *k* & (*k*=*0* $\longrightarrow$ *B*={}))
⟨*proof*⟩

lemma *finite-fun-UNIVD2*:
assumes *fin*: *finite* (*UNIV* :: (*'a* ⇒ *'b*) *set*)
shows *finite* (*UNIV* :: *'b set*)
⟨*proof*⟩

lemma *card-UNIV-unit*: *card* (*UNIV* :: *unit set*) = *1*
⟨*proof*⟩

20.7.1 Cardinality of image

lemma *card-image-le*: *finite A* ==> *card* (*f* ‘ *A*) <= *card A*
⟨*proof*⟩

lemma *card-image*:
assumes *inj-on f A*
shows *card* (*f* ‘ *A*) = *card A*
⟨*proof*⟩

lemma *bij-betw-same-card*: *bij-betw f A B* ⟹ *card A* = *card B*
⟨*proof*⟩

lemma *endo-inj-surj*: *finite A* ==> *f* ‘ *A* ⊆ *A* ==> *inj-on f A* ==> *f* ‘ *A* = *A*
⟨*proof*⟩

lemma *eq-card-imp-inj-on*:
[| *finite A*; *card*(*f* ‘ *A*) = *card A* |] ==> *inj-on f A*
⟨*proof*⟩

lemma *inj-on-iff-eq-card*:
finite A ==> *inj-on f A* = (*card*(*f* ‘ *A*) = *card A*)
⟨*proof*⟩

lemma *card-inj-on-le*:
[|*inj-on f A*; *f* ‘ *A* ⊆ *B*; *finite B* |] ==> *card A* ≤ *card B*
⟨*proof*⟩

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*
⟨*proof*⟩

20.7.2 Cardinality of sums

lemma *card-Plus*:
assumes *finite A* **and** *finite B*
shows *card* (*A* <+> *B*) = *card A* + *card B*
⟨*proof*⟩

lemma *card-Plus-conv-if*:
card (*A* <+> *B*) = (*if finite A* ∧ *finite B then card A* + *card B else 0*)
⟨*proof*⟩

20.7.3 Cardinality of the Powerset

lemma *card-Pow*: *finite A ==> card (Pow A) = Suc (Suc 0) ^ card A*
⟨*proof*⟩

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)
⟨*proof*⟩

20.7.4 Relating injectivity and surjectivity

lemma *finite-surj-inj*: *finite(A) ⟹ A <= f'A ⟹ inj-on f A*
⟨*proof*⟩

lemma *finite-UNIV-surj-inj*: **fixes** *f* :: *'a ⇒ 'a*
shows *finite(UNIV:: 'a set) ⟹ surj f ⟹ inj f*
⟨*proof*⟩

lemma *finite-UNIV-inj-surj*: **fixes** *f* :: *'a ⇒ 'a*
shows *finite(UNIV:: 'a set) ⟹ inj f ⟹ surj f*
⟨*proof*⟩

corollary *infinite-UNIV-nat*[*iff*]: *~finite(UNIV::nat set)*
⟨*proof*⟩

lemma *infinite-UNIV-char-0*[*no-atp*]:
¬ finite (UNIV::'a::semiring-char-0 set)
⟨*proof*⟩

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 == $\bigcup x{\in}A$. {(*x*,*x*)}

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

definition
Range :: ('*a* * '*b*) *set* => '*b set* **where**
Range r == *Domain*(*r*^−*1*)

definition
Field :: ('*a* * '*a*) *set* => '*a set* **where**
Field r == *Domain r* ∪ *Range r*

definition
refl-on :: ['*a set*, ('*a* * '*a*) *set*] => *bool* **where** — reflexivity over a set
refl-on A r == *r* ⊆ *A* × *A* & (*ALL x*: *A*. (*x*,*x*) : *r*)

abbreviation
refl :: ('*a* * '*a*) *set* => *bool* **where** — reflexivity over a type
refl == *refl-on UNIV*

definition
sym :: ('*a* * '*a*) *set* => *bool* **where** — symmetry predicate
sym r == *ALL x y*. (*x*,*y*): *r* −−> (*y*,*x*): *r*

definition
antisym :: ('*a* * '*a*) *set* => *bool* **where** — antisymmetry predicate
antisym r == *ALL x y*. (*x*,*y*):*r* −−> (*y*,*x*):*r* −−> *x*=*y*

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

definition
irrefl :: $('a * 'a)$ *set* => *bool* **where**
irrefl r $\equiv \forall x.\ (x,x) \notin r$

definition
total-on :: *'a set* => $('a * 'a)$ *set* => *bool* **where**
total-on A r $\equiv \forall x{\in}A. \forall y{\in}A.\ x{\neq}y \longrightarrow (x,y){\in}r \vee (y,x){\in}r$

abbreviation *total* $\equiv$ *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*]: $(a,\ a) : Id$
⟨*proof*⟩

lemma *IdE* [*elim*!]: $p : Id$ ==> $(!!x.\ p = (x,\ x)$ ==> $P)$ ==> P
⟨*proof*⟩

lemma *pair-in-Id-conv* [*iff*]: $((a,\ b) : Id) = (a = b)$
⟨*proof*⟩

lemma *refl-Id*: *refl Id*
⟨*proof*⟩

lemma *antisym-Id*: *antisym Id*
— A strange result, since *Id* is also symmetric.
⟨*proof*⟩

lemma *sym-Id*: *sym Id*
⟨*proof*⟩

lemma *trans-Id*: *trans Id*
⟨*proof*⟩

21.3 Diagonal: identity over a set

lemma *Id-on-empty* [*simp*]: *Id-on* $\{\} = \{\}$

⟨*proof*⟩

lemma *Id-on-eqI*: $a = b \Longrightarrow a : A \Longrightarrow (a, b) : \mathit{Id\text{-}on}\ A$
⟨*proof*⟩

lemma *Id-onI* [*intro*!,*no-atp*]: $a : A \Longrightarrow (a, a) : \mathit{Id\text{-}on}\ A$
⟨*proof*⟩

lemma *Id-onE* [*elim*!]:
$c : \mathit{Id\text{-}on}\ A \Longrightarrow (!!x.\ x : A \Longrightarrow c = (x, x) \Longrightarrow P) \Longrightarrow P$
— The general elimination rule.
⟨*proof*⟩

lemma *Id-on-iff*: $((x, y) : \mathit{Id\text{-}on}\ A) = (x = y\ \&\ x : A)$
⟨*proof*⟩

lemma *Id-on-subset-Times*: $\mathit{Id\text{-}on}\ A \subseteq A \times A$
⟨*proof*⟩

21.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a, b) : r \Longrightarrow (b, c) : s \Longrightarrow (a, c) : r\ O\ s$
⟨*proof*⟩

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s \Longrightarrow$
$(!!x\ y\ z.\ xz = (x, z) \Longrightarrow (x, y) : r \Longrightarrow (y, z) : s \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-compEpair*:
$(a, c) : r\ O\ s \Longrightarrow (!!y.\ (a, y) : r \Longrightarrow (y, c) : s \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *R-O-Id* [*simp*]: $R\ O\ \mathit{Id} = R$
⟨*proof*⟩

lemma *Id-O-R* [*simp*]: $\mathit{Id}\ O\ R = R$
⟨*proof*⟩

lemma *rel-comp-empty1*[*simp*]: $\{\}\ O\ R = \{\}$
⟨*proof*⟩

lemma *rel-comp-empty2*[*simp*]: $R\ O\ \{\} = \{\}$
⟨*proof*⟩

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
⟨*proof*⟩

lemma *trans-O-subset*: $\mathit{trans}\ r \Longrightarrow r\ O\ r \subseteq r$

⟨*proof*⟩

lemma *rel-comp-mono*: $r' \subseteq r \Longrightarrow s' \subseteq s \Longrightarrow (r'\ O\ s') \subseteq (r\ O\ s)$
⟨*proof*⟩

lemma *rel-comp-subset-Sigma*:
$r \subseteq A \times B \Longrightarrow s \subseteq B \times C \Longrightarrow (r\ O\ s) \subseteq A \times C$
⟨*proof*⟩

lemma *rel-comp-distrib*[*simp*]: $R\ O\ (S \cup T) = (R\ O\ S) \cup (R\ O\ T)$
⟨*proof*⟩

lemma *rel-comp-distrib2*[*simp*]: $(S \cup T)\ O\ R = (S\ O\ R) \cup (T\ O\ R)$
⟨*proof*⟩

lemma *rel-comp-UNION-distrib*: $s\ O\ UNION\ I\ r = UNION\ I\ (\%i.\ s\ O\ r\ i)$
⟨*proof*⟩

lemma *rel-comp-UNION-distrib2*: $UNION\ I\ r\ O\ s = UNION\ I\ (\%i.\ r\ i\ O\ s)$
⟨*proof*⟩

21.5 Reflexivity

lemma *refl-onI*: $r \subseteq A \times A \Longrightarrow (!!x.\ x : A \Longrightarrow (x, x) : r) \Longrightarrow refl\text{-}on\ A\ r$
⟨*proof*⟩

lemma *refl-onD*: $refl\text{-}on\ A\ r \Longrightarrow a : A \Longrightarrow (a, a) : r$
⟨*proof*⟩

lemma *refl-onD1*: $refl\text{-}on\ A\ r \Longrightarrow (x, y) : r \Longrightarrow x : A$
⟨*proof*⟩

lemma *refl-onD2*: $refl\text{-}on\ A\ r \Longrightarrow (x, y) : r \Longrightarrow y : A$
⟨*proof*⟩

lemma *refl-on-Int*: $refl\text{-}on\ A\ r \Longrightarrow refl\text{-}on\ B\ s \Longrightarrow refl\text{-}on\ (A \cap B)\ (r \cap s)$
⟨*proof*⟩

lemma *refl-on-Un*: $refl\text{-}on\ A\ r \Longrightarrow refl\text{-}on\ B\ s \Longrightarrow refl\text{-}on\ (A \cup B)\ (r \cup s)$
⟨*proof*⟩

lemma *refl-on-INTER*:
$ALL\ x{:}S.\ refl\text{-}on\ (A\ x)\ (r\ x) \Longrightarrow refl\text{-}on\ (INTER\ S\ A)\ (INTER\ S\ r)$
⟨*proof*⟩

lemma *refl-on-UNION*:
$ALL\ x{:}S.\ refl\text{-}on\ (A\ x)\ (r\ x) \Longrightarrow refl\text{-}on\ (UNION\ S\ A)\ (UNION\ S\ r)$
⟨*proof*⟩

lemma *refl-on-empty*[*simp*]: *refl-on* {} {}
⟨*proof*⟩

lemma *refl-on-Id-on*: *refl-on A* (*Id-on A*)
⟨*proof*⟩

21.6 Antisymmetry

lemma *antisymI*:
(!!*x y*. (*x*, *y*) : *r* ==> (*y*, *x*) : *r* ==> *x*=*y*) ==> *antisym r*
⟨*proof*⟩

lemma *antisymD*: *antisym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r* ==> *a* = *b*
⟨*proof*⟩

lemma *antisym-subset*: *r* ⊆ *s* ==> *antisym s* ==> *antisym r*
⟨*proof*⟩

lemma *antisym-empty* [*simp*]: *antisym* {}
⟨*proof*⟩

lemma *antisym-Id-on* [*simp*]: *antisym* (*Id-on A*)
⟨*proof*⟩

21.7 Symmetry

lemma *symI*: (!!*a b*. (*a*, *b*) : *r* ==> (*b*, *a*) : *r*) ==> *sym r*
⟨*proof*⟩

lemma *symD*: *sym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r*
⟨*proof*⟩

lemma *sym-Int*: *sym r* ==> *sym s* ==> *sym* (*r* ∩ *s*)
⟨*proof*⟩

lemma *sym-Un*: *sym r* ==> *sym s* ==> *sym* (*r* ∪ *s*)
⟨*proof*⟩

lemma *sym-INTER*: *ALL x*:*S*. *sym* (*r x*) ==> *sym* (*INTER S r*)
⟨*proof*⟩

lemma *sym-UNION*: *ALL x*:*S*. *sym* (*r x*) ==> *sym* (*UNION S r*)
⟨*proof*⟩

lemma *sym-Id-on* [*simp*]: *sym* (*Id-on A*)
⟨*proof*⟩

21.8 Transitivity

lemma *transI*:

(!!$x\ y\ z.\ (x,\ y) : r$ ==> $(y,\ z) : r$ ==> $(x,\ z) : r$) ==> *trans* r
⟨*proof*⟩

lemma *transD*: *trans* r ==> $(a,\ b) : r$ ==> $(b,\ c) : r$ ==> $(a,\ c) : r$
⟨*proof*⟩

lemma *trans-Int*: *trans* r ==> *trans* s ==> *trans* $(r \cap s)$
⟨*proof*⟩

lemma *trans-INTER*: *ALL* x:S. *trans* $(r\ x)$ ==> *trans* $(INTER\ S\ r)$
⟨*proof*⟩

lemma *trans-Id-on* [*simp*]: *trans* $(Id\text{-}on\ A)$
⟨*proof*⟩

lemma *trans-diff-Id*: *trans* $r \Longrightarrow$ *antisym* $r \Longrightarrow$ *trans* $(r-Id)$
⟨*proof*⟩

21.9 Irreflexivity

lemma *irrefl-diff-Id*[*simp*]: *irrefl*$(r-Id)$
⟨*proof*⟩

21.10 Totality

lemma *total-on-empty*[*simp*]: *total-on* $\{\}$ r
⟨*proof*⟩

lemma *total-on-diff-Id*[*simp*]: *total-on* A $(r-Id)$ = *total-on* A r
⟨*proof*⟩

21.11 Converse

lemma *converse-iff* [*iff*]: $((a,b): r\hat{}-1) = ((b,a) : r)$
⟨*proof*⟩

lemma *converseI*[*sym*]: $(a,\ b) : r$ ==> $(b,\ a) : r\hat{}-1$
⟨*proof*⟩

lemma *converseD*[*sym*]: $(a,b) : r\hat{}-1$ ==> $(b,\ a) : r$
⟨*proof*⟩

lemma *converseE* [*elim*!]:
$yx : r\hat{}-1$ ==> (!!$x\ y.\ yx = (y,\ x)$ ==> $(x,\ y) : r$ ==> P) ==> P
— More general than *converseD*, as it "splits" the member of the relation.
⟨*proof*⟩

lemma *converse-converse* [*simp*]: $(r\hat{}-1)\hat{}-1 = r$
⟨*proof*⟩

lemma *converse-rel-comp*: $(r\ O\ s)\hat{}-1 = s\hat{}-1\ O\ r\hat{}-1$
⟨*proof*⟩

lemma *converse-Int*: $(r \cap s)\hat{}-1 = r\hat{}-1 \cap s\hat{}-1$
⟨*proof*⟩

lemma *converse-Un*: $(r \cup s)\hat{}-1 = r\hat{}-1 \cup s\hat{}-1$
⟨*proof*⟩

lemma *converse-INTER*: $(INTER\ S\ r)\hat{}-1 = (INT\ x{:}S.\ (r\ x)\hat{}-1)$
⟨*proof*⟩

lemma *converse-UNION*: $(UNION\ S\ r)\hat{}-1 = (UN\ x{:}S.\ (r\ x)\hat{}-1)$
⟨*proof*⟩

lemma *converse-Id* [*simp*]: $Id\hat{}-1 = Id$
⟨*proof*⟩

lemma *converse-Id-on* [*simp*]: $(Id\text{-}on\ A)\hat{}-1 = Id\text{-}on\ A$
⟨*proof*⟩

lemma *refl-on-converse* [*simp*]: $refl\text{-}on\ A\ (converse\ r) = refl\text{-}on\ A\ r$
⟨*proof*⟩

lemma *sym-converse* [*simp*]: $sym\ (converse\ r) = sym\ r$
⟨*proof*⟩

lemma *antisym-converse* [*simp*]: $antisym\ (converse\ r) = antisym\ r$
⟨*proof*⟩

lemma *trans-converse* [*simp*]: $trans\ (converse\ r) = trans\ r$
⟨*proof*⟩

lemma *sym-conv-converse-eq*: $sym\ r = (r\hat{}-1 = r)$
⟨*proof*⟩

lemma *sym-Un-converse*: $sym\ (r \cup r\hat{}-1)$
⟨*proof*⟩

lemma *sym-Int-converse*: $sym\ (r \cap r\hat{}-1)$
⟨*proof*⟩

lemma *total-on-converse*[*simp*]: $total\text{-}on\ A\ (r\hat{}-1) = total\text{-}on\ A\ r$
⟨*proof*⟩

21.12 Domain

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

lemma *Domain-iff*: $(a : Domain\ r) = (EX\ y.\ (a,\ y) : r)$
⟨*proof*⟩

lemma *DomainI* [*intro*]: $(a,\ b) : r \Longrightarrow a : Domain\ r$
⟨*proof*⟩

lemma *DomainE* [*elim*!]:
$a : Domain\ r \Longrightarrow (!!y.\ (a,\ y) : r \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Domain-empty* [*simp*]: $Domain\ \{\} = \{\}$
⟨*proof*⟩

lemma *Domain-empty-iff*: $Domain\ r = \{\} \longleftrightarrow r = \{\}$
⟨*proof*⟩

lemma *Domain-insert*: $Domain\ (insert\ (a,\ b)\ r) = insert\ a\ (Domain\ r)$
⟨*proof*⟩

lemma *Domain-Id* [*simp*]: $Domain\ Id = UNIV$
⟨*proof*⟩

lemma *Domain-Id-on* [*simp*]: $Domain\ (Id\text{-}on\ A) = A$
⟨*proof*⟩

lemma *Domain-Un-eq*: $Domain(A \cup B) = Domain(A) \cup Domain(B)$
⟨*proof*⟩

lemma *Domain-Int-subset*: $Domain(A \cap B) \subseteq Domain(A) \cap Domain(B)$
⟨*proof*⟩

lemma *Domain-Diff-subset*: $Domain(A) - Domain(B) \subseteq Domain(A - B)$
⟨*proof*⟩

lemma *Domain-Union*: $Domain\ (Union\ S) = (\bigcup A{\in}S.\ Domain\ A)$
⟨*proof*⟩

lemma *Domain-converse*[*simp*]: $Domain(r\hat{}-1) = Range\ r$
⟨*proof*⟩

lemma *Domain-mono*: $r \subseteq s \Longrightarrow Domain\ r \subseteq Domain\ s$
⟨*proof*⟩

lemma *fst-eq-Domain*: $fst\ `\ R = Domain\ R$
⟨*proof*⟩

lemma *Domain-dprod* [*simp*]: $Domain\ (dprod\ r\ s) = uprod\ (Domain\ r)\ (Domain\ s)$
⟨*proof*⟩

lemma *Domain-dsum* [*simp*]: *Domain* (*dsum r s*) = *usum* (*Domain r*) (*Domain s*)
⟨*proof*⟩

21.13 Range

lemma *Range-iff*: (*a* : *Range r*) = (*EX y*. (*y*, *a*) : *r*)
⟨*proof*⟩

lemma *RangeI* [*intro*]: (*a*, *b*) : *r* ==> *b* : *Range r*
⟨*proof*⟩

lemma *RangeE* [*elim*!]: *b* : *Range r* ==> (!!*x*. (*x*, *b*) : *r* ==> *P*) ==> *P*
⟨*proof*⟩

lemma *Range-empty* [*simp*]: *Range* {} = {}
⟨*proof*⟩

lemma *Range-empty-iff*: *Range r* = {} ⟷ *r* = {}
⟨*proof*⟩

lemma *Range-insert*: *Range* (*insert* (*a*, *b*) *r*) = *insert b* (*Range r*)
⟨*proof*⟩

lemma *Range-Id* [*simp*]: *Range Id* = *UNIV*
⟨*proof*⟩

lemma *Range-Id-on* [*simp*]: *Range* (*Id-on A*) = *A*
⟨*proof*⟩

lemma *Range-Un-eq*: *Range*(*A* ∪ *B*) = *Range*(*A*) ∪ *Range*(*B*)
⟨*proof*⟩

lemma *Range-Int-subset*: *Range*(*A* ∩ *B*) ⊆ *Range*(*A*) ∩ *Range*(*B*)
⟨*proof*⟩

lemma *Range-Diff-subset*: *Range*(*A*) − *Range*(*B*) ⊆ *Range*(*A* − *B*)
⟨*proof*⟩

lemma *Range-Union*: *Range* (*Union S*) = (⋃ *A*∈*S*. *Range A*)
⟨*proof*⟩

lemma *Range-converse*[*simp*]: *Range*(*r*^−*1*) = *Domain r*
⟨*proof*⟩

lemma *snd-eq-Range*: *snd* ‘ *R* = *Range R*
⟨*proof*⟩

21.14 Field

lemma *mono-Field*: $r \subseteq s \Longrightarrow Field\ r \subseteq Field\ s$
⟨*proof*⟩

lemma *Field-empty*[*simp*]: $Field\ \{\} = \{\}$
⟨*proof*⟩

lemma *Field-insert*[*simp*]: $Field\ (insert\ (a,b)\ r) = \{a,b\} \cup Field\ r$
⟨*proof*⟩

lemma *Field-Un*[*simp*]: $Field\ (r \cup s) = Field\ r \cup Field\ s$
⟨*proof*⟩

lemma *Field-Union*[*simp*]: $Field\ (\bigcup R) = \bigcup (Field\ `\ R)$
⟨*proof*⟩

lemma *Field-converse*[*simp*]: $Field(r\hat{}-1) = Field\ r$
⟨*proof*⟩

21.15 Image of a set under a relation

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

lemma *Image-iff*: $(b : r``A) = (EX\ x{:}A.\ (x,\ b) : r)$
⟨*proof*⟩

lemma *Image-singleton*: $r``\{a\} = \{b.\ (a,\ b) : r\}$
⟨*proof*⟩

lemma *Image-singleton-iff* [*iff*]: $(b : r``\{a\}) = ((a,\ b) : r)$
⟨*proof*⟩

lemma *ImageI* [*intro*,*no-atp*]: $(a,\ b) : r ==> a : A ==> b : r``A$
⟨*proof*⟩

lemma *ImageE* [*elim*!]:
$b : r\ ``\ A ==> (!!x.\ (x,\ b) : r ==> x : A ==> P) ==> P$
⟨*proof*⟩

lemma *rev-ImageI*: $a : A ==> (a,\ b) : r ==> b : r\ ``\ A$
— This version's more effective when we already have the required a
⟨*proof*⟩

lemma *Image-empty* [*simp*]: $R``\{\} = \{\}$
⟨*proof*⟩

lemma *Image-Id* [*simp*]: $Id\ ``\ A = A$
⟨*proof*⟩

lemma *Image-Id-on* [*simp*]: *Id-on A `` B = A ∩ B*
⟨*proof*⟩

lemma *Image-Int-subset*: *R `` (A ∩ B) ⊆ R `` A ∩ R `` B*
⟨*proof*⟩

lemma *Image-Int-eq*:
single-valued (converse R) ==> R `` (A ∩ B) = R `` A ∩ R `` B
⟨*proof*⟩

lemma *Image-Un*: *R `` (A ∪ B) = R `` A ∪ R `` B*
⟨*proof*⟩

lemma *Un-Image*: *(R ∪ S) `` A = R `` A ∪ S `` A*
⟨*proof*⟩

lemma *Image-subset*: *r ⊆ A × B ==> r``C ⊆ B*
⟨*proof*⟩

lemma *Image-eq-UN*: *r``B = (⋃y∈ B. r``{y})*
— NOT suitable for rewriting
⟨*proof*⟩

lemma *Image-mono*: *r' ⊆ r ==> A' ⊆ A ==> (r' `` A') ⊆ (r `` A)*
⟨*proof*⟩

lemma *Image-UN*: *(r `` (UNION A B)) = (⋃x∈A. r `` (B x))*
⟨*proof*⟩

lemma *Image-INT-subset*: *(r `` INTER A B) ⊆ (⋂x∈A. r `` (B x))*
⟨*proof*⟩

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
[|single-valued (r⁻¹); A≠{}|] ==> r `` INTER A B = (⋂x∈A. r `` B x)
⟨*proof*⟩

lemma *Image-subset-eq*: *(r``A ⊆ B) = (A ⊆ - ((r^-1) `` (-B)))*
⟨*proof*⟩

21.16 Single valued relations

lemma *single-valuedI*:
ALL x y. (x,y):r --> (ALL z. (x,z):r --> y=z) ==> single-valued r
⟨*proof*⟩

lemma *single-valuedD*:
single-valued r ==> (x, y) : r ==> (x, z) : r ==> y = z
⟨*proof*⟩

lemma *single-valued-rel-comp*:
single-valued r ==> single-valued s ==> single-valued (r O s)
⟨*proof*⟩

lemma *single-valued-subset*:
r ⊆ s ==> single-valued s ==> single-valued r
⟨*proof*⟩

lemma *single-valued-Id* [*simp*]: *single-valued Id*
⟨*proof*⟩

lemma *single-valued-Id-on* [*simp*]: *single-valued (Id-on A)*
⟨*proof*⟩

21.17 Graphs given by *Collect*

lemma *Domain-Collect-split* [*simp*]: *Domain{(x,y). P x y} = {x. EX y. P x y}*
⟨*proof*⟩

lemma *Range-Collect-split* [*simp*]: *Range{(x,y). P x y} = {y. EX x. P x y}*
⟨*proof*⟩

lemma *Image-Collect-split* [*simp*]: *{(x,y). P x y} `` A = {y. EX x:A. P x y}*
⟨*proof*⟩

21.18 Inverse image

lemma *sym-inv-image*: *sym r ==> sym (inv-image r f)*
⟨*proof*⟩

lemma *trans-inv-image*: *trans r ==> trans (inv-image r f)*
⟨*proof*⟩

lemma *in-inv-image*[*simp*]: *((x,y) : inv-image r f) = ((f x, f y) : r)*
⟨*proof*⟩

lemma *converse-inv-image*[*simp*]: *(inv-image R f)^-1 = inv-image (R^-1) f*
⟨*proof*⟩

21.19 Finiteness

lemma *finite-converse* [*iff*]: *finite (r^-1) = finite r*
⟨*proof*⟩

lemma *finite-Domain*: *finite r ==> finite (Domain r)*
⟨*proof*⟩

lemma *finite-Range*: *finite r ==> finite (Range r)*
⟨*proof*⟩

lemma *finite-Field*: *finite r* ==> *finite* (*Field r*)
— A finite relation has a finite field (= *domain* ∪ *range*.
⟨*proof*⟩

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*: (⋀*x y*. (*x*, *y*) ∈ *r* ⟹ (*x*, *y*) ∈ *s*) ⟹ *r* ⊆ *s*
⟨*proof*⟩

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*: ⋀*x*. *P x* ⟹ *Q x*
shows *P* ≤ *Q*
⟨*proof*⟩

lemma *predicate1D* [*Pure.dest?*, *dest?*]:
P ≤ *Q* ⟹ *P x* ⟹ *Q x*
⟨*proof*⟩

lemma *rev-predicate1D*:

$P\ x$ ==> P <= Q ==> $Q\ x$
⟨*proof*⟩

lemma *predicate2I* [*Pure.intro*!, *intro*!]:
assumes *PQ*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y$
shows $P \leq Q$
⟨*proof*⟩

lemma *predicate2D* [*Pure.dest*, *dest*]:
$P \leq Q \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
⟨*proof*⟩

lemma *rev-predicate2D*:
$P\ x\ y$ ==> P <= Q ==> $Q\ x\ y$
⟨*proof*⟩

22.1.1 Equality

lemma *pred-equals-eq*: $((\lambda x.\ x \in R) = (\lambda x.\ x \in S)) = (R = S)$
⟨*proof*⟩

lemma *pred-equals-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) = (\lambda x\ y.\ (x,\ y) \in S)) = (R = S)$
⟨*proof*⟩

22.1.2 Order relation

lemma *pred-subset-eq*: $((\lambda x.\ x \in R) <= (\lambda x.\ x \in S)) = (R <= S)$
⟨*proof*⟩

lemma *pred-subset-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) <= (\lambda x\ y.\ (x,\ y) \in S)) = (R <= S)$
⟨*proof*⟩

22.1.3 Top and bottom elements

lemma *top1I* [*intro*!]: *top x*
⟨*proof*⟩

lemma *top2I* [*intro*!]: *top x y*
⟨*proof*⟩

lemma *bot1E* [*elim*!]: *bot x* $\Longrightarrow P$
⟨*proof*⟩

lemma *bot2E* [*elim*!]: *bot x y* $\Longrightarrow P$
⟨*proof*⟩

lemma *bot-empty-eq*: *bot* $= (\lambda x.\ x \in \{\})$
⟨*proof*⟩

lemma *bot-empty-eq2*: $bot = (\lambda x\ y.\ (x,\ y) \in \{\})$
⟨*proof*⟩

22.1.4 Binary union

lemma *sup1I1* [*elim?*]: $A\ x \Longrightarrow sup\ A\ B\ x$
⟨*proof*⟩

lemma *sup2I1* [*elim?*]: $A\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
⟨*proof*⟩

lemma *sup1I2* [*elim?*]: $B\ x \Longrightarrow sup\ A\ B\ x$
⟨*proof*⟩

lemma *sup2I2* [*elim?*]: $B\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
⟨*proof*⟩

lemma *sup1E* [*elim!*]: $sup\ A\ B\ x ==> (A\ x ==> P) ==> (B\ x ==> P) ==> P$
⟨*proof*⟩

lemma *sup2E* [*elim!*]: $sup\ A\ B\ x\ y ==> (A\ x\ y ==> P) ==> (B\ x\ y ==> P) ==> P$
⟨*proof*⟩

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI* [*intro!*]: $(\sim B\ x ==> A\ x) ==> sup\ A\ B\ x$
⟨*proof*⟩

lemma *sup2CI* [*intro!*]: $(\sim B\ x\ y ==> A\ x\ y) ==> sup\ A\ B\ x\ y$
⟨*proof*⟩

lemma *sup-Un-eq*: $sup\ (\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
⟨*proof*⟩

lemma *sup-Un-eq2* [*pred-set-conv*]: $sup\ (\lambda x\ y.\ (x,\ y) \in R)\ (\lambda x\ y.\ (x,\ y) \in S) = (\lambda x\ y.\ (x,\ y) \in R \cup S)$
⟨*proof*⟩

22.1.5 Binary intersection

lemma *inf1I* [*intro!*]: $A\ x ==> B\ x ==> inf\ A\ B\ x$
⟨*proof*⟩

lemma *inf2I* [*intro!*]: $A\ x\ y ==> B\ x\ y ==> inf\ A\ B\ x\ y$
⟨*proof*⟩

lemma *inf1E* [*elim!*]: $inf\ A\ B\ x ==> (A\ x ==> B\ x ==> P) ==> P$

⟨*proof*⟩

lemma *inf2E* [*elim*!]: *inf A B x y* ==> (*A x y* ==> *B x y* ==> *P*) ==> *P*
⟨*proof*⟩

lemma *inf1D1*: *inf A B x* ==> *A x*
⟨*proof*⟩

lemma *inf2D1*: *inf A B x y* ==> *A x y*
⟨*proof*⟩

lemma *inf1D2*: *inf A B x* ==> *B x*
⟨*proof*⟩

lemma *inf2D2*: *inf A B x y* ==> *B x y*
⟨*proof*⟩

lemma *inf-Int-eq*: *inf* (λ*x*. *x* ∈ *R*) (λ*x*. *x* ∈ *S*) = (λ*x*. *x* ∈ *R* ∩ *S*)
⟨*proof*⟩

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*)
⟨*proof*⟩

22.1.6 Unions of families

lemma *SUP1-iff*: (*SUP x*:*A*. *B x*) *b* = (*EX x*:*A*. *B x b*)
⟨*proof*⟩

lemma *SUP2-iff*: (*SUP x*:*A*. *B x*) *b c* = (*EX x*:*A*. *B x b c*)
⟨*proof*⟩

lemma *SUP1-I* [*intro*]: *a* : *A* ==> *B a b* ==> (*SUP x*:*A*. *B x*) *b*
⟨*proof*⟩

lemma *SUP2-I* [*intro*]: *a* : *A* ==> *B a b c* ==> (*SUP x*:*A*. *B x*) *b c*
⟨*proof*⟩

lemma *SUP1-E* [*elim*!]: (*SUP x*:*A*. *B x*) *b* ==> (!!*x*. *x* : *A* ==> *B x b* ==> *R*) ==> *R*
⟨*proof*⟩

lemma *SUP2-E* [*elim*!]: (*SUP x*:*A*. *B x*) *b c* ==> (!!*x*. *x* : *A* ==> *B x b c* ==> *R*) ==> *R*
⟨*proof*⟩

lemma *SUP-UN-eq*: (*SUP i*. (λ*x*. *x* ∈ *r i*)) = (λ*x*. *x* ∈ (*UN i*. *r i*))
⟨*proof*⟩

lemma *SUP-UN-eq2*: $(SUP\ i.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (UN\ i.\ r\ i))$
⟨*proof*⟩

22.1.7 Intersections of families

lemma *INF1-iff*: $(INF\ x{:}A.\ B\ x)\ b = (ALL\ x{:}A.\ B\ x\ b)$
⟨*proof*⟩

lemma *INF2-iff*: $(INF\ x{:}A.\ B\ x)\ b\ c = (ALL\ x{:}A.\ B\ x\ b\ c)$
⟨*proof*⟩

lemma *INF1-I* [*intro*!]: $(!!x.\ x : A ==> B\ x\ b) ==> (INF\ x{:}A.\ B\ x)\ b$
⟨*proof*⟩

lemma *INF2-I* [*intro*!]: $(!!x.\ x : A ==> B\ x\ b\ c) ==> (INF\ x{:}A.\ B\ x)\ b\ c$
⟨*proof*⟩

lemma *INF1-D* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b ==> a : A ==> B\ a\ b$
⟨*proof*⟩

lemma *INF2-D* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b\ c ==> a : A ==> B\ a\ b\ c$
⟨*proof*⟩

lemma *INF1-E* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b ==> (B\ a\ b ==> R) ==> (a\ ~{:}\ A ==> R) ==> R$
⟨*proof*⟩

lemma *INF2-E* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b\ c ==> (B\ a\ b\ c ==> R) ==> (a\ ~{:}\ A ==> R) ==> R$
⟨*proof*⟩

lemma *INF-INT-eq*: $(INF\ i.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (INT\ i.\ r\ i))$
⟨*proof*⟩

lemma *INF-INT-eq2*: $(INF\ i.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (INT\ i.\ r\ i))$
⟨*proof*⟩

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*]:
$((\lambda x\ y.\ (x,\ y) \in r)\ OO\ (\lambda x\ y.\ (x,\ y) \in s)) = (\lambda x\ y.\ (x,\ y) \in r\ O\ s)$
⟨*proof*⟩

22.2.2 Converse

inductive
conversep :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b \Rightarrow 'a \Rightarrow bool$
((-ˆ−−1) [1000] 1000)
for $r :: 'a \Rightarrow 'b \Rightarrow bool$
where
conversepI: $r\ a\ b \Longrightarrow r\hat{}{-}{-}1\ b\ a$

notation (*xsymbols*)
conversep ((-$^{-1-1}$) [1000] 1000)

lemma *conversepD*:
assumes *ab*: $r\hat{}{-}{-}1\ a\ b$
shows $r\ b\ a$ ⟨*proof*⟩

lemma *conversep-iff* [*iff*]: $r\hat{}{-}{-}1\ a\ b = r\ b\ a$
⟨*proof*⟩

lemma *conversep-converse-eq* [*pred-set-conv*]:
$(\lambda x\ y.\ (x,\ y) \in r)\hat{}{-}{-}1 = (\lambda x\ y.\ (x,\ y) \in r\hat{}{-}1)$
⟨*proof*⟩

lemma *conversep-conversep* [*simp*]: $(r\hat{}{-}{-}1)\hat{}{-}{-}1 = r$
⟨*proof*⟩

lemma *converse-pred-comp*: $(r\ OO\ s)\hat{}{-}{-}1 = s\hat{}{-}{-}1\ OO\ r\hat{}{-}{-}1$
⟨*proof*⟩

lemma *converse-meet*: $(inf\ r\ s)\hat{}{-}{-}1 = inf\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
⟨*proof*⟩

lemma *converse-join*: $(sup\ r\ s)\hat{}{-}{-}1 = sup\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
⟨*proof*⟩

lemma *conversep-noteq* [*simp*]: $(op\ \tilde{}{=})\hat{}{-}{-}1 = op\ \tilde{}{=}$
⟨*proof*⟩

lemma *conversep-eq* [*simp*]: $(op\ =)\hat{}{-}{-}1 = op\ =$
⟨*proof*⟩

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* ==> *DomainP r a*

inductive-cases *DomainPE* [*elim*!]: *DomainP r a*

lemma *DomainP-Domain-eq* [*pred-set-conv*]: *DomainP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Domain r*)
⟨*proof*⟩

22.2.4 Range

inductive
RangeP :: ('*a* => '*b* => *bool*) => '*b* => *bool*
for *r* :: '*a* => '*b* => *bool*
where
RangePI [*intro*]: *r a b* ==> *RangeP r b*

inductive-cases *RangePE* [*elim*!]: *RangeP r b*

lemma *RangeP-Range-eq* [*pred-set-conv*]: *RangeP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Range r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *in-inv-imagep* [*simp*]: *inv-imagep r f x y* = *r* (*f x*) (*f y*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *eval-inject*: *eval x = eval y ⟷ x = y*
⟨*proof*⟩

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*]: $\bigsqcap A = Pred\ (INFI\ A\ eval)$

definition
[*code del*]: $\bigsqcup A = Pred\ (SUPR\ A\ eval)$

definition
$- P = Pred\ (-\ eval\ P)$

definition
$P - Q = Pred\ (eval\ P - eval\ Q)$

instance ⟨*proof*⟩

end

lemma *bind-bind*:
$(P \gg= Q) \gg= R = P \gg= (\lambda x.\ Q\ x \gg= R)$
⟨*proof*⟩

lemma *bind-single*:
$P \gg= single = P$
⟨*proof*⟩

lemma *single-bind*:
$single\ x \gg= P = P\ x$
⟨*proof*⟩

lemma *bottom-bind*:
$\bot \gg= P = \bot$
⟨*proof*⟩

lemma *sup-bind*:
$(P \sqcup Q) \gg= R = P \gg= R \sqcup Q \gg= R$
⟨*proof*⟩

lemma *Sup-bind*: $(\bigsqcup A \gg= f) = \bigsqcup ((\lambda x.\ x \gg= f)\ `\ A)$
⟨*proof*⟩

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*⟩

lemma *singleI*: $eval\ (single\ x)\ x$
⟨*proof*⟩

lemma *singleI-unit*: $eval\ (single\ ())\ x$

⟨*proof*⟩

lemma *singleE*: *eval* (*single x*) *y* ⟹ (*y* = *x* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *singleE'*: *eval* (*single x*) *y* ⟹ (*x* = *y* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *bindI*: *eval P x* ⟹ *eval* (*Q x*) *y* ⟹ *eval* (*P* ⤜ *Q*) *y*
⟨*proof*⟩

lemma *bindE*: *eval* (*R* ⤜ *Q*) *y* ⟹ (⋀*x*. *eval R x* ⟹ *eval* (*Q x*) *y* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *botE*: *eval* ⊥ *x* ⟹ *P*
⟨*proof*⟩

lemma *supI1*: *eval A x* ⟹ *eval* (*A* ⊔ *B*) *x*
⟨*proof*⟩

lemma *supI2*: *eval B x* ⟹ *eval* (*A* ⊔ *B*) *x*
⟨*proof*⟩

lemma *supE*: *eval* (*A* ⊔ *B*) *x* ⟹ (*eval A x* ⟹ *P*) ⟹ (*eval B x* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *single-not-bot* [*simp*]:
single x ≠ ⊥
⟨*proof*⟩

lemma *not-bot*:
assumes *A* ≠ ⊥
obtains *x* **where** *eval A x*
⟨*proof*⟩

22.3.2 Emptiness check and definite choice

definition *is-empty* :: *'a pred* ⇒ *bool* **where**
is-empty A ⟷ *A* = ⊥

lemma *is-empty-bot*:
is-empty ⊥
⟨*proof*⟩

lemma *not-is-empty-single*:
¬ *is-empty* (*single x*)
⟨*proof*⟩

lemma *is-empty-sup*:
is-empty $(A \sqcup B) \longleftrightarrow$ *is-empty* $A \wedge$ *is-empty* B
⟨*proof*⟩

definition *singleton* :: $(unit \Rightarrow 'a) \Rightarrow 'a\ pred \Rightarrow 'a$ **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* $A = x$
⟨*proof*⟩

lemma *eval-singletonI*:
$\exists!x.$ *eval* $A\ x \Longrightarrow$ *eval* A (*singleton dfault* A)
⟨*proof*⟩

lemma *single-singleton*:
$\exists!x.$ *eval* $A\ x \Longrightarrow$ *single* (*singleton dfault* A) $= A$
⟨*proof*⟩

lemma *singleton-undefinedI*:
$\neg$ ($\exists!x.$ *eval* $A\ x$) $\Longrightarrow$ *singleton dfault* A = *dfault* ()
⟨*proof*⟩

lemma *singleton-bot*:
singleton dfault $\bot$ = *dfault* ()
⟨*proof*⟩

lemma *singleton-single*:
singleton dfault (*single* x) $= x$
⟨*proof*⟩

lemma *singleton-sup-single-single*:
singleton dfault (*single* $x \sqcup$ *single* y) = (*if* $x = y$ *then* x *else dfault* ())
⟨*proof*⟩

lemma *singleton-sup-aux*:
singleton dfault $(A \sqcup B)$ = (*if* $A = \bot$ *then singleton dfault* B
else if $B = \bot$ *then singleton dfault* A
else singleton dfault
(*single* (*singleton dfault* A) $\sqcup$ *single* (*singleton dfault* B)))
⟨*proof*⟩

lemma *singleton-sup*:
singleton dfault $(A \sqcup B)$ = (*if* $A = \bot$ *then singleton dfault* B
else if $B = \bot$ *then singleton dfault* A
else if singleton dfault A = *singleton dfault* B *then singleton dfault* A *else dfault* ())
⟨*proof*⟩

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*) ()
⟨*proof*⟩

lemma *if-predE*: *eval* (*if-pred b*) *x* ⟹ (*b* ⟹ *x* = () ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *not-predI*: ¬ *P* ⟹ *eval* (*not-pred* (*Pred* (λ*u*. *P*))) ()
⟨*proof*⟩

lemma *not-predI′*: ¬ *eval P* () ⟹ *eval* (*not-pred P*) ()
⟨*proof*⟩

lemma *not-predE*: *eval* (*not-pred* (*Pred* (λ*u*. *P*))) *x* ⟹ (¬ *P* ⟹ *thesis*) ⟹ *thesis*
⟨*proof*⟩

lemma *not-predE′*: *eval* (*not-pred P*) *x* ⟹ (¬ *eval P x* ⟹ *thesis*) ⟹ *thesis*
⟨*proof*⟩
lemma *f* () = *False* ∨ *f* () = *True*
⟨*proof*⟩

lemma *closure-of-bool-cases*:
assumes (*f* :: *unit* ⇒ *bool*) = (%*u*. *False*) ⟹ *P f*
assumes *f* = (%*u*. *True*) ⟹ *P f*
shows *P f*
⟨*proof*⟩

lemma *unit-pred-cases*:
assumes *P* ⊥
assumes *P* (*single* ())
shows *P Q*
⟨*proof*⟩

lemma *holds-if-pred*:
holds (*if-pred b*) = *b*
⟨*proof*⟩

lemma *if-pred-holds*:
if-pred (*holds P*) = *P*

⟨*proof*⟩

lemma *is-empty-holds*:
is-empty P ⟷ ¬ *holds P*
⟨*proof*⟩

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*⟩

lemma *eval-code* [*code*]: *eval* (*Seq f*) = *member* (*f* ())
⟨*proof*⟩

lemma *single-code* [*code*]:
single x = *Seq* (λ*u*. *Insert x* ⊥)
⟨*proof*⟩

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*⟩

lemma *bind-code* [*code*]:
Seq g ⤜ *f* = *Seq* (λ*u*. *apply f* (*g* ()))
⟨*proof*⟩

lemma *bot-set-code* [*code*]:
$\bot = Seq\ (\lambda u.\ Empty)$
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *contained-less-eq*:
contained xq Q ⟷ *pred-of-seq xq* ≤ *Q*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eq-pred-code* [*code*]:
fixes *P Q* :: *'a pred*
shows *eq-class.eq P Q* ⟷ *P* ≤ *Q* ∧ *Q* ≤ *P*
⟨*proof*⟩

lemma [*code*]:
pred-case f P = *f* (*eval P*)
⟨*proof*⟩

lemma [*code*]:
pred-rec f P = *f* (*eval P*)
⟨*proof*⟩

inductive *eq* :: *'a* ⇒ *'a* ⇒ *bool* **where** *eq x x*

lemma *eq-is-eq*: *eq x y* ≡ (*x* = *y*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *is-empty-code* [*code*]:
is-empty (*Seq f*) ⟷ *null* (*f* ())
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ())
⟨*proof*⟩

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*
⟨*proof*⟩

code-abort *not-unique*

code-reflect *Predicate*
  datatypes *pred* = *Seq* **and** *seq* = *Empty* | *Insert* | *Join*
  functions *map*

⟨*ML*⟩

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
  *refl*cl :: (*'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**
  *refl*cl ((-=) [*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**
  *refl*cl ((-=) [*1000*] *999*)

23.1 Reflexive closure

lemma *refl-reflcl*[*simp*]: *refl*($r\hat{}=$)
⟨*proof*⟩

lemma *antisym-reflcl*[*simp*]: *antisym*($r\hat{}=$) = *antisym r*
⟨*proof*⟩

lemma *trans-reflclI*[*simp*]: *trans r* ⟹ *trans*($r\hat{}=$)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-into-rtrancl* [*intro*]: !!*p*. *p* ∈ *r* ==> *p* ∈ $r\hat{}*$
— *rtrancl* of *r* contains *r*
⟨*proof*⟩

lemma *r-into-rtranclp* [*intro*]: *r x y* ==> $r\hat{}**$ *x y*
— *rtrancl* of *r* contains *r*
⟨*proof*⟩

lemma *rtranclp-mono*: *r* ≤ *s* ==> $r\hat{}** \leq s\hat{}**$
— monotonicity of *rtrancl*
⟨*proof*⟩

lemmas *rtrancl-mono* = *rtranclp-mono* [*to-set*]

theorem *rtranclp-induct* [*consumes 1*, *case-names base step*, *induct set*: *rtranclp*]:
assumes *a*: $r\hat{}**$ *a b*
and *cases*: *P a* !!*y z*. [| $r\hat{}**$ *a y*; *r y z*; *P y* |] ==> *P z*
shows *P b* ⟨*proof*⟩

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* ($r\hat{}*$)
⟨*proof*⟩

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* $(r\hat{}*)$
⟨*proof*⟩

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

lemma *rtranclp-trans*:
assumes *xy*: $r\hat{}** x\ y$
and *yz*: $r\hat{}** y\ z$
shows $r\hat{}** x\ z$ ⟨*proof*⟩

lemma *rtranclE* [*cases set*: *rtrancl*]:
assumes *major*: $(a::'a,\ b) : r\hat{}*$
obtains
(*base*) $a = b$
| (*step*) y **where** $(a,\ y) : r\hat{}*$ **and** $(y,\ b) : r$
— elimination of *rtrancl* – by induction on a special formula
⟨*proof*⟩

lemma *rtrancl-Int-subset*: $[|\ Id \subseteq s;\ (r\hat{}* \cap s)\ O\ r \subseteq s|] ==> r\hat{}* \subseteq s$
⟨*proof*⟩

lemma *converse-rtranclp-into-rtranclp*:
$r\ a\ b \Longrightarrow r^{**}\ b\ c \Longrightarrow r^{**}\ a\ c$
⟨*proof*⟩

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [*to-set*]

More r^* equations and inclusions.

lemma *rtranclp-idemp* [*simp*]: $(r\hat{}**)\hat{}** = r\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [*to-set*]

lemma *rtrancl-idemp-self-comp* [*simp*]: $R\hat{}*\ O\ R\hat{}* = R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-subset-rtrancl*: $r \subseteq s\hat{}* ==> r\hat{}* \subseteq s\hat{}*$
⟨*proof*⟩

lemma *rtranclp-subset*: $R \leq S ==> S \leq R\hat{}** ==> S\hat{}** = R\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(sup\ (R\hat{}**)\ (S\hat{}**))\hat{}** = (sup\ R\ S)\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflcl* [*simp*]: $(R\hat{}==)\hat{}** = R\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)\hat{}* = r\hat{}*$
⟨*proof*⟩

lemma *rtranclp-r-diff-Id*: $(inf\ r\ op\ \tilde{}=)\hat{}** = r\hat{}**$
⟨*proof*⟩

theorem *rtranclp-converseD*:
assumes *r*: $(r\hat{}--1)\hat{}**\ x\ y$
shows $r\hat{}**\ y\ x$
⟨*proof*⟩

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

theorem *rtranclp-converseI*:
assumes $r\hat{}**\ y\ x$
shows $(r\hat{}--1)\hat{}**\ x\ y$
⟨*proof*⟩

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

lemma *rtrancl-converse*: $(r\hat{}-1)\hat{}* = (r\hat{}*)\hat{}-1$
⟨*proof*⟩

lemma *sym-rtrancl*: *sym r* ==> *sym* $(r\hat{}*)$
⟨*proof*⟩

theorem *converse-rtranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: $r\hat{}**\ a\ b$
and *cases*: $P\ b$!!$y\ z$. [| $r\ y\ z$; $r\hat{}**\ z\ b$; $P\ z$ |] ==> $P\ y$
shows $P\ a$
⟨*proof*⟩

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*: $r\hat{}**\ x\ z$

and *cases*: $x{=}z \Longrightarrow P$
$!!y.\ [|\ r\ x\ y;\ r\hat{}{*}{*}\ y\ z\ |] \Longrightarrow P$
shows P
⟨*proof*⟩

lemmas *converse-rtranclE* = *converse-rtranclpE* [*to-set*]

lemmas *converse-rtranclpE2* = *converse-rtranclpE* [*of* - (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemmas *converse-rtranclE2* = *converse-rtranclE* [*of* (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemma *r-comp-rtrancl-eq*: $r\ O\ r\hat{}{*} = r\hat{}{*}\ O\ r$
⟨*proof*⟩

lemma *rtrancl-unfold*: $r\hat{}{*} = Id\ Un\ r\hat{}{*}\ O\ r$
⟨*proof*⟩

lemma *rtrancl-Un-separatorE*:
$(a,b) : (P \cup Q)\hat{}{*} \Longrightarrow \forall x\ y.\ (a,x) : P\hat{}{*} \longrightarrow (x,y) : Q \longrightarrow x{=}y \Longrightarrow (a,b) : P\hat{}{*}$
⟨*proof*⟩

lemma *rtrancl-Un-separator-converseE*:
$(a,b) : (P \cup Q)\hat{}{*} \Longrightarrow \forall x\ y.\ (x,b) : P\hat{}{*} \longrightarrow (y,x) : Q \longrightarrow y{=}x \Longrightarrow (a,b) : P\hat{}{*}$
⟨*proof*⟩

lemma *Image-closed-trancl*:
assumes $r\ ``\ X \subseteq X$ **shows** $r^*\ ``\ X = X$
⟨*proof*⟩

23.3 Transitive closure

lemma *trancl-mono*: $!!p.\ p \in r\hat{}{+} \Longrightarrow r \subseteq s \Longrightarrow p \in s\hat{}{+}$
⟨*proof*⟩

lemma *r-into-trancl'*: $!!p.\ p : r \Longrightarrow p : r\hat{}{+}$
⟨*proof*⟩

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: $r\hat{}{+}{+}\ a\ b \Longrightarrow r\hat{}{*}{*}\ a\ b$
⟨*proof*⟩

lemmas *trancl-into-rtrancl* = *tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*: **assumes** *r*: $r\hat{}{*}{*}\ a\ b$
shows $!!c.\ r\ b\ c \Longrightarrow r\hat{}{+}{+}\ a\ c$ ⟨*proof*⟩

lemmas *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*: $[|\ r\ a\ b;\ r\hat{}{*}{*}\ b\ c\ |] \Longrightarrow r\hat{}{+}{+}\ a\ c$

— intro rule from r and *rtrancl*
⟨*proof*⟩

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*: $r\hat{}{+}{+}\ a\ b$
and *cases*: $!!y.\ r\ a\ y \Longrightarrow P\ y$
$!!y\ z.\ r\hat{}{+}{+}\ a\ y \Longrightarrow r\ y\ z \Longrightarrow P\ y \Longrightarrow P\ z$
shows $P\ b$ ⟨*proof*⟩

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*: $r\hat{}{+}{+}\ x\ y$
and *cases*: $!!x\ y.\ r\ x\ y \Longrightarrow P\ x\ y$
$!!x\ y\ z.\ [|\ r\hat{}{+}{+}\ x\ y;\ P\ x\ y;\ r\hat{}{+}{+}\ y\ z;\ P\ y\ z\ |] \Longrightarrow P\ x\ z$
shows $P\ x\ y$
— Another induction rule for trancl, incorporating transitivity
⟨*proof*⟩

lemmas *trancl-trans-induct* = *tranclp-trans-induct* [*to-set*]

lemma *tranclE* [*cases set*: *trancl*]:
assumes $(a,\ b) : r\hat{}{+}$
obtains
$(base)\ (a,\ b) : r$
| $(step)\ c$ **where** $(a,\ c) : r\hat{}{+}$ **and** $(c,\ b) : r$
⟨*proof*⟩

lemma *trancl-Int-subset*: $[|\ r \subseteq s;\ (r\hat{}{+} \cap s)\ O\ r \subseteq s|] \Longrightarrow r\hat{}{+} \subseteq s$
⟨*proof*⟩

lemma *trancl-unfold*: $r\hat{}{+} = r\ Un\ r\hat{}{+}\ O\ r$
⟨*proof*⟩

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* $(r\hat{}{+})$
⟨*proof*⟩

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

lemma *tranclp-trans*:
assumes *xy*: $r\hat{}{+}{+}\ x\ y$
and *yz*: $r\hat{}{+}{+}\ y\ z$
shows $r\hat{}{+}{+}\ x\ z$ ⟨*proof*⟩

lemma *trancl-id* [*simp*]: *trans* $r \Longrightarrow r\hat{}{+} = r$
⟨*proof*⟩

lemma *rtranclp-tranclp-tranclp*:
assumes $r\hat{}{*}{*}\ x\ y$
shows $!!z.\ r\hat{}{+}{+}\ y\ z \Longrightarrow r\hat{}{+}{+}\ x\ z$ ⟨*proof*⟩

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: $r\ a\ b \Longrightarrow r\hat{}{+}{+}\ b\ c \Longrightarrow r\hat{}{+}{+}\ a\ c$
⟨*proof*⟩

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-insert*:
$(insert\ (y,\ x)\ r)\hat{}{+} = r\hat{}{+} \cup \{(a,\ b).\ (a,\ y) \in r\hat{}{*} \wedge (x,\ b) \in r\hat{}{*}\}$
— primitive recursion for *trancl* over finite relations
⟨*proof*⟩

lemma *tranclp-converseI*: $(r\hat{}{+}{+})\hat{}{-}{-}1\ x\ y \Longrightarrow (r\hat{}{-}{-}1)\hat{}{+}{+}\ x\ y$
⟨*proof*⟩

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*: $(r\hat{}{-}{-}1)\hat{}{+}{+}\ x\ y \Longrightarrow (r\hat{}{+}{+})\hat{}{-}{-}1\ x\ y$
⟨*proof*⟩

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

lemma *tranclp-converse*: $(r\hat{}{-}{-}1)\hat{}{+}{+} = (r\hat{}{+}{+})\hat{}{-}{-}1$
⟨*proof*⟩

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: *sym* $r \Longrightarrow$ *sym* $(r\hat{}{+})$
⟨*proof*⟩

lemma *converse-tranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: $r\hat{}{+}{+}\ a\ b$
and *cases*: $!!y.\ r\ y\ b \Longrightarrow P(y)$
$!!y\ z.[|\ r\ y\ z;\ \ r\hat{}{+}{+}\ z\ b;\ \ P(z)\ |] \Longrightarrow P(y)$
shows $P\ a$

⟨*proof*⟩

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

lemma *tranclpD*: $R\hat{}{+}{+}\ x\ y \Longrightarrow EX\ z.\ R\ x\ z \wedge R\hat{}{*}{*}\ z\ y$
⟨*proof*⟩

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *converse-tranclpE*:
assumes *major*: *tranclp r x z*
assumes *base*: $r\ x\ z \Longrightarrow P$
assumes *step*: $\bigwedge y.\ [\!|\ r\ x\ y;\ tranclp\ r\ y\ z\ |\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemmas *converse-tranclE* = *converse-tranclpE* [*to-set*]

lemma *tranclD2*:
$(x,\ y) \in R^+ \Longrightarrow \exists z.\ (x,\ z) \in R^* \wedge (z,\ y) \in R$
⟨*proof*⟩

lemma *irrefl-tranclI*: $r\hat{}{-}1 \cap r\hat{}{*} = \{\} \Longrightarrow (x,\ x) \notin r\hat{}{+}$
⟨*proof*⟩

lemma *irrefl-trancl-rD*: $!!X.\ ALL\ x.\ (x,\ x) \notin r\hat{}{+} \Longrightarrow (x,\ y) \in r \Longrightarrow x \neq y$
⟨*proof*⟩

lemma *trancl-subset-Sigma-aux*:
$(a,\ b) \in r\hat{}{*} \Longrightarrow r \subseteq A \times A \Longrightarrow a = b \vee a \in A$
⟨*proof*⟩

lemma *trancl-subset-Sigma*: $r \subseteq A \times A \Longrightarrow r\hat{}{+} \subseteq A \times A$
⟨*proof*⟩

lemma *reflcl-tranclp* [*simp*]: $(r\hat{}{+}{+})\hat{}{=}{=} = r\hat{}{*}{*}$
⟨*proof*⟩

lemmas *reflcl-trancl* [*simp*] = *reflcl-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r\hat{}{=})\hat{}{+} = r\hat{}{*}$
⟨*proof*⟩

lemma *trancl-empty* [*simp*]: $\{\}\hat{}{+} = \{\}$
⟨*proof*⟩

lemma *rtrancl-empty* [*simp*]: $\{\}\hat{}{*} = Id$
⟨*proof*⟩

lemma *rtranclpD*: $R\hat{}** a\ b ==> a = b \vee a \neq b \wedge R\hat{}++ a\ b$
⟨*proof*⟩

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*:
$(x,y) \in R^* = (x{=}y \vee x{\neq}y \wedge (x,y) \in R^+)$
⟨*proof*⟩

lemma *trancl-unfold-right*: $r\hat{}+ = r\hat{}* \ O\ r$
⟨*proof*⟩

lemma *trancl-unfold-left*: $r\hat{}+ = r\ O\ r\hat{}*$
⟨*proof*⟩

Simplifying nested closures

lemma *rtrancl-trancl-absorb*[*simp*]: $(R\hat{}*)\hat{}+ = R\hat{}*$
⟨*proof*⟩

lemma *trancl-rtrancl-absorb*[*simp*]: $(R\hat{}+)\hat{}* = R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-reflcl-absorb*[*simp*]: $(R\hat{}*)\hat{}= = R\hat{}*$
⟨*proof*⟩

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* $(R\hat{}*) = UNIV$
⟨*proof*⟩

lemma *Range-rtrancl* [*simp*]: *Range* $(R\hat{}*) = UNIV$
⟨*proof*⟩

lemma *rtrancl-Un-subset*: $(R\hat{}* \cup S\hat{}*) \subseteq (R\ Un\ S)\hat{}*$
⟨*proof*⟩

lemma *in-rtrancl-UnI*: $x \in R\hat{}* \vee x \in S\hat{}* ==> x \in (R \cup S)\hat{}*$
⟨*proof*⟩

lemma *trancl-domain* [*simp*]: *Domain* $(r\hat{}+) =$ *Domain* r
⟨*proof*⟩

lemma *trancl-range* [*simp*]: *Range* $(r\hat{}+) =$ *Range* r
⟨*proof*⟩

lemma *Not-Domain-rtrancl*:
$x \sim$: *Domain* $R ==> ((x,\ y) : R\hat{}*) = (x = y)$
⟨*proof*⟩

lemma *trancl-subset-Field2*: $r\hat{}+ <= Field\ r \times Field\ r$

⟨*proof*⟩

lemma *finite-trancl*: *finite* (*r*^+) = *finite r*
⟨*proof*⟩

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
⟦ *single-valued r*; (*x*,*y*) ∈ *r*^*; (*x*,*z*) ∈ *r*^* ⟧
⟹ (*y*,*z*) ∈ *r*^* ∨ (*z*,*y*) ∈ *r*^*
⟨*proof*⟩

lemma *r-r-into-trancl*: (*a*, *b*) ∈ *R* ==> (*b*, *c*) ∈ *R* ==> (*a*, *c*) ∈ *R*^+
⟨*proof*⟩

lemma *trancl-into-trancl* [*rule-format*]:
(*a*, *b*) ∈ *r*^+ ==> (*b*, *c*) ∈ *r* --> (*a*,*c*) ∈ *r*^+
⟨*proof*⟩

lemma *tranclp-rtranclp-tranclp*:
r^++ *a b* ==> *r*^** *b c* ==> *r*^++ *a c*
⟨*proof*⟩

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 ^^ *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 $R :: ('a \times 'a)\ set$
shows $R\ \hat{}\hat{}\ 1 = R$
⟨*proof*⟩

lemma *rel-pow-0-I*:
$(x,\ x) \in R\ \hat{}\hat{}\ 0$
⟨*proof*⟩

lemma *rel-pow-Suc-I*:
$(x,\ y) \in\ R\ \hat{}\hat{}\ n \Longrightarrow (y,\ z) \in R \Longrightarrow (x,\ z) \in R\ \hat{}\hat{}\ Suc\ n$
⟨*proof*⟩

lemma *rel-pow-Suc-I2*:
$(x,\ y) \in R \Longrightarrow (y,\ z) \in R\ \hat{}\hat{}\ n \Longrightarrow (x,\ z) \in R\ \hat{}\hat{}\ Suc\ n$
⟨*proof*⟩

lemma *rel-pow-0-E*:
$(x,\ y) \in R\ \hat{}\hat{}\ 0 \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-Suc-E*:
$(x,\ z) \in R\ \hat{}\hat{}\ Suc\ n \Longrightarrow (\bigwedge y.\ (x,\ y) \in R\ \hat{}\hat{}\ n \Longrightarrow (y,\ z) \in R \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-E*:
$(x,\ z) \in\ R\ \hat{}\hat{}\ n \Longrightarrow\ (n = 0 \Longrightarrow x = z \Longrightarrow P)$
$\Longrightarrow (\bigwedge y\ m.\ n = Suc\ m \Longrightarrow (x,\ y) \in\ R\ \hat{}\hat{}\ m \Longrightarrow (y,\ z) \in R \Longrightarrow P)$
$\Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-Suc-D2*:
$(x,\ z) \in R\ \hat{}\hat{}\ Suc\ n \Longrightarrow (\exists\, y.\ (x,\ y) \in R \wedge (y,\ z) \in R\ \hat{}\hat{}\ n)$
⟨*proof*⟩

lemma *rel-pow-Suc-E2*:
$(x,\ z) \in R\ \hat{}\hat{}\ Suc\ n \Longrightarrow (\bigwedge y.\ (x,\ y) \in R \Longrightarrow (y,\ z) \in R\ \hat{}\hat{}\ n \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-Suc-D2'*:
$\forall x\ y\ z.\ (x,\ y) \in R\ \hat{}\hat{}\ n \wedge (y,\ z) \in R \longrightarrow (\exists\, w.\ (x,\ w) \in R \wedge (w,\ z) \in R\ \hat{}\hat{}\ n)$
⟨*proof*⟩

lemma *rel-pow-E2*:
$(x,\ z) \in R\ \hat{}\hat{}\ n \Longrightarrow\ (n = 0 \Longrightarrow x = z \Longrightarrow P)$
$\Longrightarrow (\bigwedge y\ m.\ n = Suc\ m \Longrightarrow (x,\ y) \in R \Longrightarrow (y,\ z) \in R\ \hat{}\hat{}\ m \Longrightarrow P)$
$\Longrightarrow P$

⟨*proof*⟩

lemma *rel-pow-add*: $R \mathbin{\hat{}\hat{}} (m+n) = R \mathbin{\hat{}\hat{}} m \; O \; R \mathbin{\hat{}\hat{}} n$
⟨*proof*⟩

lemma *rel-pow-commute*: $R \; O \; R \mathbin{\hat{}\hat{}} n = R \mathbin{\hat{}\hat{}} n \; O \; R$
⟨*proof*⟩

lemma *rtrancl-imp-UN-rel-pow*:
assumes $p \in R\hat{}*$
shows $p \in (\bigcup n.\ R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *rel-pow-imp-rtrancl*:
assumes $p \in R \mathbin{\hat{}\hat{}} n$
shows $p \in R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-is-UN-rel-pow*:
$R\hat{}* = (\bigcup n.\ R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *rtrancl-power*:
$p \in R\hat{}* \longleftrightarrow (\exists n.\ p \in R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *trancl-power*:
$p \in R\hat{}+ \longleftrightarrow (\exists n > 0.\ p \in R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *rtrancl-imp-rel-pow*:
$p \in R\hat{}* \Longrightarrow \exists n.\ p \in R \mathbin{\hat{}\hat{}} n$
⟨*proof*⟩

lemma *single-valued-rel-pow*:
fixes $R :: ('a * 'a)\ set$
shows *single-valued* $R \Longrightarrow$ *single-valued* $(R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

23.5 Setup of transitivity reasoner

⟨*ML*⟩

Optional methods.

⟨*ML*⟩

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*
⟨*proof*⟩

lemma *wfUNIVI*:
(!!*P x*. (*ALL x*. (*ALL y*. (*y*,*x*) : *r* --> *P*(*y*)) --> *P*(*x*)) ==> *P*(*x*)) ==> *wf*(*r*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wf-induct*:
[| *wf*(*r*);
!!*x*.[| *ALL y*. (*y*,*x*): *r* --> *P*(*y*) |] ==> *P*(*x*)
|] ==> *P*(*a*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wf-asym*:
assumes *wf r* $(a, x) \in r$
obtains $(x, a) \notin r$
⟨*proof*⟩

lemma *wf-not-refl* [*simp*]: *wf r* ==> $(a, a) \sim: r$
⟨*proof*⟩

lemma *wf-irrefl*: **assumes** *wf r* **obtains** $(a, a) \notin r$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *wellorder*) *wf*:
wf $\{(x, y).\ x < y\}$
⟨*proof*⟩

24.2 Basic Results

Point-free characterization of well-foundedness

lemma *wfE-pf*:
assumes *wf*: *wf R*
assumes *a*: $A \subseteq R$ `` A
shows $A = \{\}$
⟨*proof*⟩

lemma *wfI-pf*:
assumes *a*: $\bigwedge A.\ A \subseteq R$ `` $A \Longrightarrow A = \{\}$
shows *wf R*
⟨*proof*⟩

Minimal-element characterization of well-foundedness

lemma *wfE-min*:
assumes *wf*: *wf R* **and** *Q*: $x \in Q$
obtains *z* **where** $z \in Q$ $\bigwedge y.\ (y, z) \in R \Longrightarrow y \notin Q$
⟨*proof*⟩

lemma *wfI-min*:
assumes *a*: $\bigwedge x\ Q.\ x \in Q \Longrightarrow \exists\, z{\in}Q.\ \forall\, y.\ (y, z) \in R \longrightarrow y \notin Q$
shows *wf R*
⟨*proof*⟩

lemma *wf-eq-minimal*: *wf r* = $(\forall\, Q\ x.\ x{\in}Q$ −−> $(\exists\, z{\in}Q.\ \forall\, y.\ (y,z){\in}r$ −−> $y{\notin}Q))$
⟨*proof*⟩

lemmas *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

Well-foundedness of transitive closure

lemma *wf-trancl*:
assumes *wf r*
shows *wf* (*r*^+)
⟨*proof*⟩

lemmas *wfP-trancl* = *wf-trancl* [*to-pred*]

lemma *wf-converse-trancl*: *wf* (*r*^−1) ==> *wf* ((*r*^+)^−1)
⟨*proof*⟩

Well-foundedness of subsets

lemma *wf-subset*: [| *wf*(*r*); *p*<=*r* |] ==> *wf*(*p*)
⟨*proof*⟩

lemmas *wfP-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

lemma *wf-empty* [*iff*]: *wf* {}
⟨*proof*⟩

lemma *wfP-empty* [*iff*]:
wfP (λ*x y*. *False*)
⟨*proof*⟩

lemma *wf-Int1*: *wf r* ==> *wf* (*r Int r'*)
⟨*proof*⟩

lemma *wf-Int2*: *wf r* ==> *wf* (*r' Int r*)
⟨*proof*⟩

Exponentiation

lemma *wf-exp*:
assumes *wf* (*R* ^^ *n*)
shows *wf R*
⟨*proof*⟩

Well-foundedness of insert

lemma *wf-insert* [*iff*]: *wf*(*insert* (*y*,*x*) *r*) = (*wf*(*r*) & (*x*,*y*) ~: *r*^*)
⟨*proof*⟩

Well-foundedness of image

lemma *wf-prod-fun-image*: [| *wf r*; *inj f* |] ==> *wf*(*prod-fun f f* ‘ *r*)
⟨*proof*⟩

24.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
assumes *wf R wf S*
assumes $R\ O\ S \subseteq R$
shows *wf* $(R \cup S)$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *wfP-SUP*:
$\forall i.\ wfP\ (r\ i) \Longrightarrow \forall i\ j.\ r\ i \neq r\ j \longrightarrow inf\ (DomainP\ (r\ i))\ (RangeP\ (r\ j)) = bot$
$\Longrightarrow wfP\ (SUPR\ UNIV\ r)$
⟨*proof*⟩

lemma *wf-Union*:
[| *ALL r:R. wf r*;
ALL r:R. ALL s:R. r ~= s --> Domain r Int Range s = {}
|] ==> *wf*(*Union R*)
⟨*proof*⟩

lemma *wf-Un*:
[| *wf r*; *wf s*; *Domain r Int Range s* = {} |] ==> *wf*(*r Un s*)
⟨*proof*⟩

lemma *wf-union-merge*:
$wf\ (R \cup S) = wf\ (R\ O\ R \cup S\ O\ R \cup S)$ (**is** *wf ?A* = *wf ?B*)
⟨*proof*⟩

lemma *wf-comp-self*: $wf\ R = wf\ (R\ O\ R)$ — special case
⟨*proof*⟩

24.4 Acyclic relations

definition *acyclic* :: $('a * 'a)\ set \Rightarrow bool$ **where**
acyclic r == !*x*. (*x*,*x*) ~: *r*^+

abbreviation *acyclicP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
acyclicP r == *acyclic* {(*x*, *y*). *r x y*}

lemma *acyclicI*: *ALL x*. (*x*, *x*) ~: *r*^+ ==> *acyclic r*
⟨*proof*⟩

lemma *wf-acyclic*: *wf r* ==> *acyclic r*
⟨*proof*⟩

lemmas *wfP-acyclicP* = *wf-acyclic* [*to-pred*]

lemma *acyclic-insert* [*iff*]:
  acyclic(*insert* (*y*,*x*) *r*) = (*acyclic r* & (*x*,*y*) ~: *r*^*)
⟨*proof*⟩

lemma *acyclic-converse* [*iff*]: *acyclic*(*r*^−1) = *acyclic r*
⟨*proof*⟩

lemmas *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

lemma *acyclic-impl-antisym-rtrancl*: *acyclic r* ==> *antisym*(*r*^*)
⟨*proof*⟩

lemma *acyclic-subset*: [| *acyclic s*; *r* <= *s* |] ==> *acyclic r*
⟨*proof*⟩

Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf* [*rule-format*]: *finite r* ==> *acyclic r* −−> *wf r*
⟨*proof*⟩

lemma *finite-acyclic-wf-converse*: [|*finite r*; *acyclic r*|] ==> *wf* (*r*^−1)
⟨*proof*⟩

lemma *wf-iff-acyclic-if-finite*: *finite r* ==> *wf r* = *acyclic r*
⟨*proof*⟩

24.5 *nat* is well-founded

lemma *less-nat-rel*: *op* < = (λ*m n*. *n* = *Suc m*)^++
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *pred-nat-trancl-eq-le*:
  (*m*, *n*) ∈ *pred-nat*^* ⟷ *m* ≤ *n*
  ⟨*proof*⟩

lemma *wf-pred-nat*: *wf pred-nat*
⟨*proof*⟩

lemma *wf-less-than* [*iff*]: *wf less-than*
⟨*proof*⟩

lemma *trans-less-than* [*iff*]: *trans less-than*
⟨*proof*⟩

lemma *less-than-iff* [*iff*]: ((*x*,*y*): *less-than*) = (*x*<*y*)
⟨*proof*⟩

lemma *wf-less*: *wf* {(*x*, *y*::*nat*). *x* < *y*}
⟨*proof*⟩

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*
⟨*proof*⟩

theorems *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp r b* ==> *r a b* ==> *accp r a*
⟨*proof*⟩

lemma *not-accp-down*:
assumes *na*: ¬ *accp R x*

obtains z **where** $R\ z\ x$ **and** $\neg\ accp\ R\ z$
⟨*proof*⟩

lemma *accp-downwards-aux*: $r^{**}\ b\ a \Longrightarrow accp\ r\ a \longrightarrow accp\ r\ b$
⟨*proof*⟩

theorem *accp-downwards*: $accp\ r\ a \Longrightarrow r^{**}\ b\ a \Longrightarrow accp\ r\ b$
⟨*proof*⟩

theorem *accp-wfPI*: $\forall x.\ accp\ r\ x \Longrightarrow wfP\ r$
⟨*proof*⟩

theorem *accp-wfPD*: $wfP\ r \Longrightarrow accp\ r\ x$
⟨*proof*⟩

theorem *wfP-accp-iff*: $wfP\ r = (\forall x.\ accp\ r\ x)$
⟨*proof*⟩

Smaller relations have bigger accessible parts:

lemma *accp-subset*:
assumes *sub*: $R1 \leq R2$
shows $accp\ R2 \leq accp\ R1$
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *in-lex-prod*[*simp*]:
(((*a*,*b*),(*a*',*b*')): *r* <**lex**> *s*) = ((*a*,*a*'): *r* ∨ (*a* = *a*' ∧ (*b*, *b*') : *s*))
⟨*proof*⟩

op <**lex**> preserves transitivity

lemma *trans-lex-prod* [*intro*!]:
[| *trans R1*; *trans R2* |] ==> *trans* (*R1* <**lex**> *R2*)
⟨*proof*⟩

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* $\Longrightarrow$ *wf* (*f* <∗*mlex*∗> *R*)
⟨*proof*⟩

lemma *mlex-less*: *f x* < *f y* $\Longrightarrow$ (*x*, *y*) ∈ *f* <∗*mlex*∗> *R*
⟨*proof*⟩

lemma *mlex-leq*: *f x* ≤ *f y* $\Longrightarrow$ (*x*, *y*) ∈ *R* $\Longrightarrow$ (*x*, *y*) ∈ *f* <∗*mlex*∗> *R*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *trans-finite-psubset*: *trans finite-psubset*
⟨*proof*⟩

lemma *in-finite-psubset*[*simp*]: (*A*, *B*) ∈ *finite-psubset* = (*A* < *B* & *finite B*)
⟨*proof*⟩

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* ≠ {} $\Longrightarrow$ (⋀*x*. *x* ∈ *X* $\Longrightarrow$ ∃ *y*∈*Y*. (*x*, *y*) ∈ *R*) $\Longrightarrow$ (*X*, *Y*) ∈ *max-ext R*

lemma *max-ext-wf*:
assumes *wf*: *wf r*
shows *wf* (*max-ext r*)
⟨*proof*⟩

lemma *max-ext-additive*:
(*A*, *B*) ∈ *max-ext R* $\Longrightarrow$ (*C*, *D*) ∈ *max-ext R* $\Longrightarrow$
(*A* ∪ *C*, *B* ∪ *D*) ∈ *max-ext R*
⟨*proof*⟩

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*⟩

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*^*
⟨*proof*⟩

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*⟩

lemma *weak-decr-stable*:
ALL i. *f* (*Suc i*) <= ((*f i*)::*nat*) ==> *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
⟨*proof*⟩

24.9 size of a datatype value

⟨*ML*⟩

lemma *size-bool* [*code*]:
size (*b*::*bool*) = *0* ⟨*proof*⟩

lemma *nat-size* [*simp*, *code*]: *size* (*n*::*nat*) = *n*
⟨*proof*⟩

declare *prod.size* [*no-atp*]

lemma [*code*]:
size (*P* :: *'a Predicate.pred*) = *0* ⟨*proof*⟩

lemma [*code*]:
pred-size f P = *0* ⟨*proof*⟩

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'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
⟨*proof*⟩

lemma *THE-default1-equality*:
$[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow THE\text{-}default\ d\ P = a$
⟨*proof*⟩

lemma *THE-default-none*:
$\neg(\exists!x.\ P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$
⟨*proof*⟩

lemma *fundef-ex1-existence*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $G\ x\ (f\ x)$
⟨*proof*⟩

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
assumes *elm*: $G\ x\ (h\ x)$
shows $h\ x = f\ x$

⟨*proof*⟩

lemma *fundef-ex1-iff*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $(G\ x\ y) = (f\ x = y)$
⟨*proof*⟩

lemma *fundef-default-value*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *graph*: $\bigwedge x\ y.\ G\ x\ y \Longrightarrow D\ x$
assumes $\neg\ D\ x$
shows $f\ x = d\ x$
⟨*proof*⟩

definition *in-rel-def*[*simp*]:
$in\text{-}rel\ R\ x\ y == (x,\ y) \in R$

lemma *wf-in-rel*:
$wf\ R \Longrightarrow wfP\ (in\text{-}rel\ R)$
⟨*proof*⟩

⟨*ML*⟩

25.2 Measure Functions

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$
where *is-measure-trivial*: *is-measure f*

⟨*ML*⟩

lemma *measure-size*[*measure-function*]: *is-measure size*
⟨*proof*⟩

lemma *measure-fst*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* $(\lambda p.\ f\ (fst\ p))$
⟨*proof*⟩
lemma *measure-snd*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* $(\lambda p.\ f\ (snd\ p))$
⟨*proof*⟩

⟨*ML*⟩

25.3 Congruence Rules

lemma *let-cong* [*fundef-cong*]:
$M = N \Longrightarrow (\bigwedge x.\ x = N \Longrightarrow f\ x = g\ x) \Longrightarrow Let\ M\ f = Let\ N\ g$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *comp-cong* [*fundef-cong*]:
$f\ (g\ x) = f'\ (g'\ x') \Longrightarrow (f\ o\ g)\ x = (f'\ o\ g')\ x'$
⟨*proof*⟩

25.4 Simp rules for termination proofs

lemma *termination-basic-simps*[*termination-simp*]:
$x < (y::nat) \Longrightarrow x < y + z$
$x < z \Longrightarrow x < y + z$
$x \leq y \Longrightarrow x \leq y + (z::nat)$
$x \leq z \Longrightarrow x \leq y + (z::nat)$
$x < y \Longrightarrow x \leq (y::nat)$
⟨*proof*⟩

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*
⟨*proof*⟩

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*:
$R\ O\ R = \{\} \Longrightarrow wf\ R$
⟨*proof*⟩

25.6 Reduction Pairs

definition
reduction-pair P = (*wf* (*fst P*) ∧ *fst P O snd P* ⊆ *fst P*)

lemma *reduction-pairI*[*intro*]: *wf R* $\Longrightarrow$ *R O S* ⊆ *R* $\Longrightarrow$ *reduction-pair* (*R*, *S*)
⟨*proof*⟩

lemma *reduction-pair-lemma*:
assumes *rp*: *reduction-pair P*
assumes *R* ⊆ *fst P*

assumes $S \subseteq snd\ P$
assumes $wf\ S$
shows $wf\ (R \cup S)$
⟨*proof*⟩

definition
$rp\text{-}inv\text{-}image = (\lambda(R,S)\ f.\ (inv\text{-}image\ R\ f,\ inv\text{-}image\ S\ f))$

lemma *rp-inv-image-rp*:
$reduction\text{-}pair\ P \Longrightarrow reduction\text{-}pair\ (rp\text{-}inv\text{-}image\ P\ f)$
⟨*proof*⟩

25.7 Concrete orders for SCNP termination proofs

definition $pair\text{-}less = less\text{-}than <*lex*> less\text{-}than$
definition [*code del*]: $pair\text{-}leq = pair\text{-}less\hat{}=$
definition $max\text{-}strict = max\text{-}ext\ pair\text{-}less$
definition [*code del*]: $max\text{-}weak = max\text{-}ext\ pair\text{-}leq \cup \{(\{\}, \{\})\}$
definition [*code del*]: $min\text{-}strict = min\text{-}ext\ pair\text{-}less$
definition [*code del*]: $min\text{-}weak = min\text{-}ext\ pair\text{-}leq \cup \{(\{\}, \{\})\}$

lemma *wf-pair-less*[*simp*]: $wf\ pair\text{-}less$
⟨*proof*⟩

Introduction rules for *pair-less*/*pair-leq*

lemma *pair-leqI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in pair\text{-}leq$
and *pair-leqI2*: $a \leq b \Longrightarrow s \leq t \Longrightarrow ((a, s), (b, t)) \in pair\text{-}leq$
and *pair-lessI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in pair\text{-}less$
and *pair-lessI2*: $a \leq b \Longrightarrow s < t \Longrightarrow ((a, s), (b, t)) \in pair\text{-}less$
⟨*proof*⟩

Introduction rules for max

lemma *smax-emptyI*:
$finite\ Y \Longrightarrow Y \neq \{\} \Longrightarrow (\{\}, Y) \in max\text{-}strict$
and *smax-insertI*:
$[\![y \in Y; (x, y) \in pair\text{-}less; (X, Y) \in max\text{-}strict]\!] \Longrightarrow (insert\ x\ X, Y) \in max\text{-}strict$
and *wmax-emptyI*:
$finite\ X \Longrightarrow (\{\}, X) \in max\text{-}weak$
and *wmax-insertI*:
$[\![y \in YS; (x, y) \in pair\text{-}leq; (XS, YS) \in max\text{-}weak]\!] \Longrightarrow (insert\ x\ XS, YS) \in max\text{-}weak$
⟨*proof*⟩

Introduction rules for min

lemma *smin-emptyI*:
$X \neq \{\} \Longrightarrow (X, \{\}) \in min\text{-}strict$
and *smin-insertI*:
$[\![x \in XS; (x, y) \in pair\text{-}less; (XS, YS) \in min\text{-}strict]\!] \Longrightarrow (XS, insert\ y\ YS) \in min\text{-}strict$

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*
⟨*proof*⟩

Reduction Pairs

lemma *max-ext-compat*:
assumes $R\ O\ S \subseteq R$
shows *max-ext* $R\ O$ (*max-ext* $S \cup \{(\{\},\{\})\}) \subseteq$ *max-ext* R
⟨*proof*⟩

lemma *max-rpair-set*: *reduction-pair* (*max-strict*, *max-weak*)
⟨*proof*⟩

lemma *min-ext-compat*:
assumes $R\ O\ S \subseteq R$
shows *min-ext* $R\ O$ (*min-ext* $S \cup \{(\{\},\{\})\}) \subseteq$ *min-ext* R
⟨*proof*⟩

lemma *min-rpair-set*: *reduction-pair* (*min-strict*, *min-weak*)
⟨*proof*⟩

25.8 Tool setup

⟨*ML*⟩

end

26 Extraction: Program extraction for HOL

theory *Extraction*
imports *Option*
uses *Tools/rewrite-hol-proof.ML*
begin

26.1 Setup

⟨*ML*⟩

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 ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ⟶ *Q*) ≡ *Type* (*TYPE*('*P* ⇒ '*Q*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*(*Null*))) ⟹
typeof (∀ *x*. *P x*) ≡ *Type* (*TYPE*(*Null*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*('*P*))) ⟹
typeof (∀ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a* ⇒ '*P*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*(*Null*))) ⟹
typeof (∃ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*('*P*))) ⟹
typeof (∃ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a* × '*P*))

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*(*Null*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*(*sumbool*))

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*('*Q option*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*(*Null*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*('*P option*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*('*P* + '*Q*))

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ∧ *Q*) ≡ *Type* (*TYPE*('*Q*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*(*Null*)) ⟹

$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

26.3 Realizability

realizability
$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv (\forall x{::}'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P{::}bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

26.4 Computational content of basic inference rules

theorem *disjE-realizer*:
assumes *r*: $case\ x\ of\ Inl\ p \Rightarrow P\ p \mid Inr\ q \Rightarrow Q\ q$
and *r1*: $\bigwedge p.\ P\ p \Longrightarrow R\ (f\ p)$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows $R\ (case\ x\ of\ Inl\ p \Rightarrow f\ p \mid Inr\ q \Rightarrow g\ q)$
⟨*proof*⟩

theorem *disjE-realizer2*:
assumes *r*: $case\ x\ of\ None \Rightarrow P \mid Some\ q \Rightarrow Q\ q$
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows $R\ (case\ x\ of\ None \Rightarrow f \mid Some\ q \Rightarrow g\ q)$
⟨*proof*⟩

theorem *disjE-realizer3*:
assumes *r*: $case\ x\ of\ Left \Rightarrow P \mid Right \Rightarrow Q$
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $Q \Longrightarrow R\ g$
shows $R\ (case\ x\ of\ Left \Rightarrow f \mid Right \Rightarrow g)$
⟨*proof*⟩

theorem *conjI-realizer*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
⟨*proof*⟩

theorem *exI-realizer*:
$P\ y\ x \Longrightarrow P\ (snd\ (x,\ y))\ (fst\ (x,\ y))$ ⟨*proof*⟩

theorem *exE-realizer*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$

⟨*proof*⟩

theorem *exE-realizer'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q\ \langle proof\rangle$

realizers

impI $(P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ (c\text{: -})\ (d\text{: -})\ P\ Q\ pq\ (h\text{: -}).\ allI \cdot - \cdot c \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot - \cdot (h \cdot x))$

impI (P): *Null*
$\Lambda\ (c\text{: -})\ P\ Q\ (h\text{: -}).\ allI \cdot - \cdot c \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot - \cdot (h \cdot x))$

impI (Q): $\lambda q.\ q\ \Lambda\ (c\text{: -})\ P\ Q\ q.\ impI \cdot - \cdot -$

impI: *Null impI*

mp $(P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ (c\text{: -})\ (d\text{: -})\ P\ Q\ pq\ (h\text{: -})\ p.\ mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot c \cdot h)$

mp (P): *Null*
$\Lambda\ (c\text{: -})\ P\ Q\ (h\text{: -})\ p.\ mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot c \cdot h)$

mp (Q): $\lambda q.\ q\ \Lambda\ (c\text{: -})\ P\ Q\ q.\ mp \cdot - \cdot -$

mp: *Null mp*

allI (P): $\lambda p.\ p\ \Lambda\ (c\text{: -})\ P\ (d\text{: -})\ p.\ allI \cdot - \cdot d$

allI: *Null allI*

spec (P): $\lambda x\ p.\ p\ x\ \Lambda\ (c\text{: -})\ P\ x\ (d\text{: -})\ p.\ spec \cdot - \cdot x \cdot d$

spec: *Null spec*

exI (P): $\lambda x\ p.\ (x,\ p)\ \Lambda\ (c\text{: -})\ P\ x\ (d\text{: -})\ p.\ exI\text{-}realizer \cdot P \cdot p \cdot x \cdot c \cdot d$

exI: $\lambda x.\ x\ \Lambda\ P\ x\ (c\text{: -})\ (h\text{: -}).\ h$

exE $(P,\ Q)$: $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
$\Lambda\ (c\text{: -})\ (d\text{: -})\ P\ Q\ (e\text{: -})\ p\ (h\text{: -})\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h$

exE (P): *Null*
$\Lambda\ (c\text{: -})\ P\ Q\ (d\text{: -})\ p.\ exE\text{-}realizer' \cdot - \cdot - \cdot - \cdot - \cdot c \cdot d$

exE (Q): $\lambda x\ pq.\ pq\ x$
$\Lambda\ (c\text{: -})\ P\ Q\ (d\text{: -})\ x\ (h1\text{: -})\ pq\ (h2\text{: -}).\ h2 \cdot x \cdot h1$

exE: *Null*
$\Lambda\ P\ Q\ (c\text{: -})\ x\ (h1\text{: -})\ (h2\text{: -}).\ h2 \cdot x \cdot h1$

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* · - · - • (*sum.cases-2* · - · *Q* · *q* • *arity-type-bool* •

$c \cdot d$)

disjI2 (*P*): *None*
Λ (*c*: -) *Q P*. *iffD2* · - · - · (*option.cases-1* · - · - · *arity-type-bool* · *c*)

disjI2 (*Q*): *Some*
Λ (*c*: -) *Q P q*. *iffD2* · - · - · (*option.cases-2* · - · *Q* · *q* · *arity-type-bool* · *c*)

disjI2: *Right*
Λ *Q P*. *iffD2* · - · - · (*sumbool.cases-2* · - · - · *arity-type-bool*)

disjE (*P*, *Q*, *R*): λ*pq pr qr*.
(*case pq of Inl p* ⇒ *pr p* | *Inr q* ⇒ *qr q*)
Λ (*c*: -) (*d*: -) (*e*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
disjE-realizer · - · - · *pq* · *R* · *pr* · *qr* · *c* · *d* · *e* · *h1* · *h2*

disjE (*Q*, *R*): λ*pq pr qr*.
(*case pq of None* ⇒ *pr* | *Some q* ⇒ *qr q*)
Λ (*c*: -) (*d*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
disjE-realizer2 · - · - · *pq* · *R* · *pr* · *qr* · *c* · *d* · *h1* · *h2*

disjE (*P*, *R*): λ*pq pr qr*.
(*case pq of None* ⇒ *qr* | *Some p* ⇒ *pr p*)
Λ (*c*: -) (*d*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr* (*h3*: -).
disjE-realizer2 · - · - · *pq* · *R* · *qr* · *pr* · *c* · *d* · *h1* · *h3* · *h2*

disjE (*R*): λ*pq pr qr*.
(*case pq of Left* ⇒ *pr* | *Right* ⇒ *qr*)
Λ (*c*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
disjE-realizer3 · - · - · *pq* · *R* · *pr* · *qr* · *c* · *h1* · *h2*

disjE (*P*, *Q*): *Null*
Λ (*c*: -) (*d*: -) *P Q R pq*. *disjE-realizer* · - · - · *pq* · (λ*x*. *R*) · - · - · *c* · *d* · *arity-type-bool*

disjE (*Q*): *Null*
Λ (*c*: -) *P Q R pq*. *disjE-realizer2* · - · - · *pq* · (λ*x*. *R*) · - · - · *c* · *arity-type-bool*

disjE (*P*): *Null*
Λ (*c*: -) *P Q R pq* (*h1*: -) (*h2*: -) (*h3*: -).
disjE-realizer2 · - · - · *pq* · (λ*x*. *R*) · - · - · *c* · *arity-type-bool* · *h1* · *h3* · *h2*

disjE: *Null*
Λ *P Q R pq*. *disjE-realizer3* · - · - · *pq* · (λ*x*. *R*) · - · - · *arity-type-bool*

FalseE (*P*): *default*
Λ (*c*: -) *P*. *FalseE* · -

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* ·
($\lambda pq.\ \forall x.\ P\ x \longrightarrow Q\ (pq\ x)$) · pq ·
($\lambda qp.\ \forall x.\ Q\ x \longrightarrow P\ (qp\ x)$) · qp ·
(*arity-type-fun* · c · d) ·
(*arity-type-fun* · d · c) ·
(*allI* · - · c · ($\Lambda\ x.$ *impI* · - · - · ($h1$ · x))) ·
(*allI* · - · d · ($\Lambda\ x.$ *impI* · - · - · ($h2$ · x)))

iffI (P): $\lambda p.\ p$
Λ (c: -) P Q ($h1$: -) p ($h2$: -). *conjI* · - · - ·
(*allI* · - · c · ($\Lambda\ x.$ *impI* · - · - · ($h1$ · x))) ·
(*impI* · - · - · $h2$)

iffI (Q): $\lambda q.\ q$
Λ (c: -) P Q q ($h1$: -) ($h2$: -). *conjI* · - · - ·
(*impI* · - · - · $h1$) ·
(*allI* · - · c · ($\Lambda\ x.$ *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*⟩

end

References