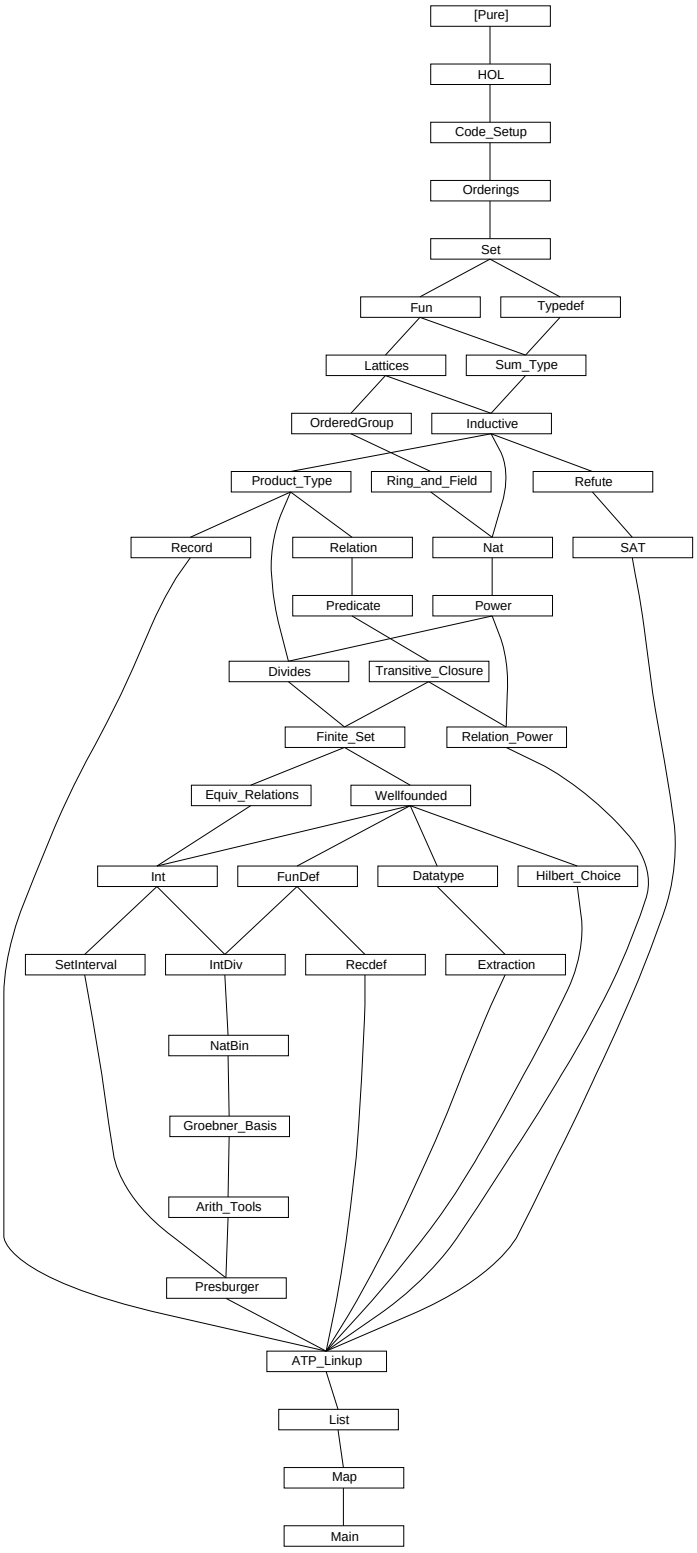
Isabelle/HOL — Higher-Order Logic

June 8, 2008

Contents


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

1 HOL: The basis of Higher-Order Logic

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

1.1 Primitive logic

1.1.1 Core syntax

classes *type*
defaultsort *type*
⟨*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*
arbitrary :: *'a*

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

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

local

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

1.1.2 Additional concrete syntax

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

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

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

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

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

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

notation (*xsymbols*)
iff (**infixr** $\longleftrightarrow$ *25*)

nonterminals
letbinds letbind
case-syn cases-syn

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

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

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

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

⟨*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*)

1.1.3 Axioms and basic definitions

axioms
eq-reflection: $(x=y) ==> (x==y)$

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

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

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

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

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

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

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

finalconsts
op =
op -->
The
arbitrary

axiomatization
undefined :: *'a*

axiomatization where
undefined-fun: *undefined x = undefined*

1.1.4 Generic classes and algebraic operations

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

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

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

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

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

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

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

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

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

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

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

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

end

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

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

notation

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

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

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

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

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

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

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

end

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

⟨*ML*⟩

1.2 Fundamental rules

1.2.1 Equality

Thanks to Stephan Merz

lemma *subst*:
assumes *eq*: *s* = *t* **and** *p*: *P s*
shows *P t*
⟨*proof*⟩

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

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

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

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

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

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

For calculational reasoning:

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

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

1.2.2 Congruence rules for application

lemma *fun-cong*: $(f::'a \Rightarrow 'b) = g \Longrightarrow f(x){=}g(x)$
⟨*proof*⟩

lemma *arg-cong*: $x{=}y \Longrightarrow f(x){=}f(y)$
⟨*proof*⟩

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

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

1.2.3 Equality of booleans – iff

lemma *iffI*: **assumes** $P \Longrightarrow Q$ **and** $Q \Longrightarrow P$ **shows** $P{=}Q$
⟨*proof*⟩

lemma *iffD2*: $[\![P{=}Q;\ Q\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *rev-iffD2*: $[\![Q;\ P{=}Q\ ]\!] \Longrightarrow P$
⟨*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*: $[\!|\ P \dashrightarrow Q;\ Q \dashrightarrow P\ |\!] \Longrightarrow R$
shows R
⟨*proof*⟩

1.2.4 True

lemma *TrueI*: *True*
⟨*proof*⟩

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

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

1.2.5 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*⟩

1.2.6 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* ==> *P*
⟨*proof*⟩

1.2.7 Negation

lemma *notI*:
assumes *P* ==> *False*
shows ~*P*
⟨*proof*⟩

lemma *False-not-True*: *False* ~= *True*
⟨*proof*⟩

lemma *True-not-False*: *True* ~= *False*
⟨*proof*⟩

lemma *notE*: [| ~*P*; *P* |] ==> *R*
⟨*proof*⟩

lemma *notI2*: (*P* ⟹ ¬ *Pa*) ⟹ (*P* ⟹ *Pa*) ⟹ ¬ *P*
⟨*proof*⟩

1.2.8 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*: [| $x = a$; $a \sim= b$; $b = y$ |] ==> $x \sim= y$
⟨*proof*⟩

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

1.2.9 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*⟩

1.2.10 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*⟩

1.2.11 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*⟩

1.2.12 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*⟩

1.2.13 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*: $EX!\ x.\ P\ x \Longrightarrow EX\ x.\ P\ x$
⟨*proof*⟩

1.2.14 THE: definite description operator

lemma *the-equality*:
assumes *prema*: $P\ a$
and *premx*: $!!x.\ P\ x \Longrightarrow x=a$
shows $(THE\ x.\ P\ x) = a$
⟨*proof*⟩

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

lemma *theI'*: $EX!\ x.\ P\ x \Longrightarrow P\ (THE\ x.\ P\ x)$
⟨*proof*⟩

lemma *theI2*:
assumes $P\ a\ !!x.\ P\ x \Longrightarrow x=a\ !!x.\ P\ x \Longrightarrow Q\ x$
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?*]: $[|\ EX!x.\ P\ x;\ P\ a\ |] \Longrightarrow (THE\ x.\ P\ x) = a$
⟨*proof*⟩

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

1.2.15 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes $\sim Q \Longrightarrow P$ **shows** $P|Q$
⟨*proof*⟩

lemma *excluded-middle*: $\sim P \mid P$
⟨*proof*⟩

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

lemma *case-split-thm*:
assumes *prem1*: $P \Longrightarrow Q$
and *prem2*: $\sim P \Longrightarrow Q$
shows Q
⟨*proof*⟩
lemmas *case-split* = *case-split-thm* [*case-names True False*]

lemma *impCE*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $\sim P \Longrightarrow R$ $Q \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *impCE'*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $Q \Longrightarrow R$ $\sim P \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *iffCE*:
assumes *major*: $P = Q$
and *minor*: $[\![P; Q]\!] \Longrightarrow R$ $[\![\sim P; \sim Q]\!] \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *exCI*:
assumes $ALL\ x.\ \sim P(x) \Longrightarrow P(a)$
shows $EX\ x.\ P(x)$
⟨*proof*⟩

1.2.16 Intuitionistic Reasoning

lemma *impE'*:
assumes *1*: $P \longrightarrow Q$
and *2*: $Q \Longrightarrow R$
and *3*: $P \longrightarrow Q \Longrightarrow P$
shows R
⟨*proof*⟩

lemma *allE'*:
assumes *1*: $ALL\ x.\ P\ x$
and *2*: $P\ x \Longrightarrow ALL\ x.\ P\ x \Longrightarrow Q$
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*⟩

1.2.17 Atomizing meta-level connectives

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

lemma *atomize-imp* [*atomize*]: (*A* ==> *B*) == *Trueprop* (*A* --> *B*)
⟨*proof*⟩

lemma *atomize-not*: (*A* ==> *False*) == *Trueprop* (~*A*)
⟨*proof*⟩

lemma *atomize-eq* [*atomize*]: (*x* == *y*) == *Trueprop* (*x* = *y*)
⟨*proof*⟩

lemma *atomize-conj* [*atomize*]:
includes *meta-conjunction-syntax*
shows (*A* && *B*) == *Trueprop* (*A* & *B*)
⟨*proof*⟩

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

1.2.18 Atomizing elimination rules

⟨*ML*⟩

lemma *atomize-exL*[*atomize-elim*]: (!!*x*. *P x* ==> *Q*) == ((*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*]: $(!!B.\ (A ==> B) ==> B) == Trueprop\ A$
⟨*proof*⟩

1.3 Package setup

1.3.1 Classical Reasoner setup

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

lemma *swap*: $\sim P ==> (\sim R ==> P) ==> R$
⟨*proof*⟩

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

⟨*ML*⟩

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

⟨*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*!]
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*⟩

1.3.2 Simplifier

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

lemma *simp-thms*:
shows *not-not*: (~ ~ *P*) = *P*
and *Not-eq-iff*: ((~*P*) = (~*Q*)) = (*P* = *Q*)
and
(*P* ~= *Q*) = (*P* = (~*Q*))
(*P* | ~*P*) = *True* (~*P* | *P*) = *True*
(*x* = *x*) = *True*
and *not-True-eq-False*: (¬ *True*) = *False*
and *not-False-eq-True*: (¬ *False*) = *True*
and
(~*P*) ~= *P* *P* ~= (~*P*)
(*True*=*P*) = *P*
and *eq-True*: (*P* = *True*) = *P*
and (*False*=*P*) = (~*P*)
and *eq-False*: (*P* = *False*) = (¬ *P*)
and
(*True* −−> *P*) = *P* (*False* −−> *P*) = *True*
(*P* −−> *True*) = *True* (*P* −−> *P*) = *True*
(*P* −−> *False*) = (~*P*) (*P* −−> ~*P*) = (~*P*)
(*P* & *True*) = *P* (*True* & *P*) = *P*
(*P* & *False*) = *False* (*False* & *P*) = *False*
(*P* & *P*) = *P* (*P* & (*P* & *Q*)) = (*P* & *Q*)
(*P* & ~*P*) = *False* (~*P* & *P*) = *False*

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

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'*) ==> (*Q'* ==> (*P* = *P'*)) ==> ((*P* & *Q*) = (*P'* & *Q'*))
⟨*proof*⟩

The | congruence rule: not included by default!

lemma *disj-cong*:
(*P* = *P'*) ==> (~*P'* ==> (*Q* = *Q'*)) ==> ((*P* | *Q*) = (*P'* | *Q'*))
⟨*proof*⟩

if-then-else rules

lemma *if-True*: (*if True then x else y*) = *x*
⟨*proof*⟩

lemma *if-False*: (*if False then x else y*) = *y*
⟨*proof*⟩

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

lemma *if-not-P*: ~*P* ==> (*if P then x else y*) = *y*
⟨*proof*⟩

lemma *split-if*: *P* (*if Q then x else y*) = ((*Q* --> *P*(*x*)) & (~*Q* --> *P*(*y*)))
⟨*proof*⟩

lemma *split-if-asm*: *P* (*if Q then x else y*) = (~((*Q* & ~*P x*) | (~*Q* & ~*P y*)))
⟨*proof*⟩

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

lemma *if-cancel*: (*if c then x else x*) = *x*
⟨*proof*⟩

lemma *if-eq-cancel*: (*if x* = *y then y else x*) = *x*
⟨*proof*⟩

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

lemma *if-bool-eq-disj*: (*if P then Q else R*) = ((*P*&*Q*) | (~*P*&*R*))
— And this form is useful for expanding *if*s on the LEFT.
⟨*proof*⟩

lemma *Eq-TrueI*: *P* ==> *P* == *True* ⟨*proof*⟩
lemma *Eq-FalseI*: ~*P* ==> *P* == *False* ⟨*proof*⟩

let rules for simproc

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

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

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

constdefs
simp-implies :: [*prop*, *prop*] => *prop* (**infixr** =*simp*=> *1*)
[*code func del*]: *simp-implies* ≡ *op* ==>

lemma *simp-impliesI*:
assumes *PQ*: $(\mathit{PROP}\ P \Longrightarrow \mathit{PROP}\ Q)$
shows $\mathit{PROP}\ P$ =*simp*=> $\mathit{PROP}\ Q$
⟨*proof*⟩

lemma *simp-impliesE*:
assumes *PQ*: $\mathit{PROP}\ P$ =*simp*=> $\mathit{PROP}\ Q$
and *P*: $\mathit{PROP}\ P$
and *QR*: $\mathit{PROP}\ Q \Longrightarrow \mathit{PROP}\ R$
shows $\mathit{PROP}\ R$
⟨*proof*⟩

lemma *simp-implies-cong*:
assumes PP' :$\mathit{PROP}\ P == \mathit{PROP}\ P'$
and $P'QQ'$: $\mathit{PROP}\ P' ==> (\mathit{PROP}\ Q == \mathit{PROP}\ Q')$
shows $(\mathit{PROP}\ P$ =*simp*=> $\mathit{PROP}\ Q) == (\mathit{PROP}\ P'$ =*simp*=> $\mathit{PROP}\ Q')$
⟨*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 $(y = x) == \mathit{False}$ from premise $\sim(x = y)$:

⟨*ML*⟩

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

lemma *ex-simps*:
$!!P\ Q.\ (EX\ x.\ P\ x\ \&\ Q) = ((EX\ x.\ P\ x)\ \&\ Q)$
$!!P\ Q.\ (EX\ x.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x.\ Q\ x))$
$!!P\ Q.\ (EX\ x.\ P\ x \mid Q) = ((EX\ x.\ P\ x) \mid Q)$
$!!P\ Q.\ (EX\ x.\ P \mid Q\ x) = (P \mid (EX\ x.\ Q\ x))$
$!!P\ Q.\ (EX\ x.\ P\ x \longrightarrow Q) = ((ALL\ x.\ P\ x) \longrightarrow Q)$
$!!P\ Q.\ (EX\ x.\ P \longrightarrow Q\ x) = (P \longrightarrow (EX\ x.\ Q\ x))$
— Miniscoping: pushing in existential quantifiers.
⟨*proof*⟩

lemma *all-simps*:
$!!P\ Q.\ (ALL\ x.\ P\ x\ \&\ Q) = ((ALL\ x.\ P\ x)\ \&\ Q)$
$!!P\ Q.\ (ALL\ x.\ P\ \&\ Q\ x) = (P\ \&\ (ALL\ x.\ Q\ x))$
$!!P\ Q.\ (ALL\ x.\ P\ x \mid Q) = ((ALL\ x.\ P\ x) \mid Q)$
$!!P\ Q.\ (ALL\ x.\ P \mid Q\ x) = (P \mid (ALL\ x.\ Q\ x))$
$!!P\ Q.\ (ALL\ x.\ P\ x \longrightarrow Q) = ((EX\ x.\ P\ x) \longrightarrow Q)$
$!!P\ Q.\ (ALL\ x.\ P \longrightarrow Q\ x) = (P \longrightarrow (ALL\ x.\ Q\ x))$
— Miniscoping: pushing in universal quantifiers.
⟨*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*⟩

1.3.3 Generic cases and induction

Rule projections:

⟨*ML*⟩

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

lemma *induct-forall-eq*: $(!!x.\ P\ x) == Trueprop\ (induct\text{-}forall\ (\lambda x.\ P\ x))$
⟨*proof*⟩

lemma *induct-implies-eq*: $(A ==> B) == Trueprop\ (induct\text{-}implies\ A\ B)$
⟨*proof*⟩

lemma *induct-equal-eq*: $(x == y) == Trueprop\ (induct\text{-}equal\ x\ y)$
⟨*proof*⟩

lemma *induct-conj-eq*:
includes *meta-conjunction-syntax*
shows $(A\ \&\&\ B) == Trueprop\ (induct\text{-}conj\ A\ B)$
⟨*proof*⟩

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

lemma *induct-forall-conj*: $induct\text{-}forall\ (\lambda x.\ induct\text{-}conj\ (A\ x)\ (B\ x)) =$
$induct\text{-}conj\ (induct\text{-}forall\ A)\ (induct\text{-}forall\ B)$
⟨*proof*⟩

lemma *induct-implies-conj*: $induct\text{-}implies\ C\ (induct\text{-}conj\ A\ B) =$
$induct\text{-}conj\ (induct\text{-}implies\ C\ A)\ (induct\text{-}implies\ C\ B)$
⟨*proof*⟩

lemma *induct-conj-curry*: $(induct\text{-}conj\ A\ B ==> PROP\ C) == (A ==> B ==> PROP\ C)$
⟨*proof*⟩

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

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

Method setup.

⟨*ML*⟩

1.4 Other simple lemmas and lemma duplicates

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

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

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*⟩

lemma *mk-left-commute*:
fixes *f* (**infix** ⊗ *60*)
assumes *a*: ⋀*x y z*. *(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)* **and**
c: ⋀*x y*. *x ⊗ y = y ⊗ x*
shows *x ⊗ (y ⊗ z) = y ⊗ (x ⊗ z)*
⟨*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))
(¬ ¬(P)) = P
⟨*proof*⟩

1.5 Basic ML bindings

⟨*ML*⟩

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

code-datatype *Trueprop prop*

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

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

lemma *If-case-cert*:
includes *meta-conjunction-syntax*
assumes $CASE \equiv (\lambda b.\ If\ b\ f\ g)$
shows $(CASE\ True \equiv f)$ && $(CASE\ False \equiv g)$
⟨*proof*⟩

⟨*ML*⟩

class *eq* = *type* +
fixes *eq* :: $'a \Rightarrow 'a \Rightarrow bool$
assumes *eq*: $eq\ x\ y \longleftrightarrow x = y$
begin

lemma *equals-eq* [*code inline*, *code func*]: $op = \equiv eq$
⟨*proof*⟩

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

end

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

⟨*ML*⟩

lemma [*code func*]:
shows $False \wedge x \longleftrightarrow False$
and $True \wedge x \longleftrightarrow x$
and $x \wedge False \longleftrightarrow False$
and $x \wedge True \longleftrightarrow x$ ⟨*proof*⟩

lemma [*code func*]:
shows $False \vee x \longleftrightarrow x$
and $True \vee x \longleftrightarrow True$
and $x \vee False \longleftrightarrow x$
and $x \vee True \longleftrightarrow True$ ⟨*proof*⟩

lemma [*code func*]:
shows $\neg\ True \longleftrightarrow False$
and $\neg\ False \longleftrightarrow True$ ⟨*proof*⟩

1.7 Legacy tactics and ML bindings

⟨*ML*⟩

end

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

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

2.1 SML code generator setup

⟨*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*⟩

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

Evaluation

⟨*ML*⟩

2.2 Generic code generator setup

using built-in Haskell equality

code-class *eq*

(*Haskell Eq* **where** *op* = ≡ (==))

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

type bool

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

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

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

code-reserved *SML*
bool true false not

code-reserved *OCaml*
bool not

code generation for undefined as exception

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

Let and If

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

2.3 Evaluation oracle

⟨*ML*⟩

2.4 Normalization by evaluation

⟨*ML*⟩

end

3 Orderings: Abstract orderings

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

3.1 Partial orders

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

Reflexivity.

lemma *eq-refl*: $x = y \Longrightarrow x \leq y$
— This form is useful with the classical reasoner.
⟨*proof*⟩

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

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

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

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*⟩

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-order*:
  order $(op \geq)$ $(op >)$
⟨*proof*⟩

end

3.2 Linear (total) orders

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

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

Replacing the old Nat.leI

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*:
linorder $(op \geq)$ $(op >)$
⟨*proof*⟩

min/max

for historic reasons, definitions are done in context ord

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

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

lemma *min-le-iff-disj*:
$min\ x\ y \leq z \longleftrightarrow x \leq z \vee y \leq z$
⟨*proof*⟩

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

lemma *min-less-iff-disj*:
$min\ x\ y < z \longleftrightarrow x < z \vee y < z$
⟨*proof*⟩

lemma *less-max-iff-disj*:
$z < max\ x\ y \longleftrightarrow z < x \vee z < y$
⟨*proof*⟩

lemma *min-less-iff-conj* [*simp*]:
$z < min\ x\ y \longleftrightarrow z < x \wedge z < y$
⟨*proof*⟩

lemma *max-less-iff-conj* [*simp*]:
$max\ x\ y < z \longleftrightarrow x < z \wedge y < z$
⟨*proof*⟩

lemma *split-min* [*noatp*]:
$P\ (min\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ i) \wedge (\neg\ i \leq j \longrightarrow P\ j)$
⟨*proof*⟩

lemma *split-max* [*noatp*]:
$P\ (max\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ j) \wedge (\neg\ i \leq j \longrightarrow P\ i)$
⟨*proof*⟩

end

3.3 Reasoning tools setup

⟨*ML*⟩

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

context *order*
begin

lemmas
[*order add less-reflE*: *order op* = :: *'a* ⇒ *'a* ⇒ *bool op* <= *op* <] =
less-irrefl [*THEN notE*]
lemmas
[*order add le-refl*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
order-refl
lemmas
[*order add less-imp-le*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-imp-le
lemmas
[*order add eqI*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
antisym
lemmas
[*order add eqD1*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
eq-refl
lemmas
[*order add eqD2*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
sym [*THEN eq-refl*]
lemmas
[*order add less-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-trans
lemmas
[*order add less-le-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-le-trans
lemmas
[*order add le-less-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
le-less-trans
lemmas
[*order add le-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
order-trans
lemmas
[*order add le-neq-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
le-neq-trans
lemmas
[*order add neq-le-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
neq-le-trans
lemmas
[*order add less-imp-neq*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-imp-neq
lemmas

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

end

context *linorder*
begin

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

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

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

end

⟨*ML*⟩

3.4 Name duplicates

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

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

lemmas *order-less-imp-not-less = order-class.less-imp-not-less*
lemmas *order-less-imp-triv = order-class.less-imp-triv*
lemmas *order-less-asym' = order-class.less-asym'*

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

3.5 Bounded quantifiers

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

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

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

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

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

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

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

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

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

⟨*ML*⟩

3.6 Transitivity reasoning

context *ord*
begin

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

lemma *ord-eq-le-trans*: $a = b \Longrightarrow b \leq c \Longrightarrow a \leq c$
⟨*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* ==> *f b* < (*c*::*'c*::*order*) ==>
(!!*x y*. *x* < *y* ==> *f x* < *f y*) ==> *f a* < *c*
⟨*proof*⟩

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

lemma *order-le-less-subst2*: (*a*::*'a*::*order*) <= *b* ==> *f b* < (*c*::*'c*::*order*) ==>
(!!*x y*. *x* <= *y* ==> *f x* <= *f y*) ==> *f a* < *c*

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemmas *order-trans-rules* [*trans*] =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

3.7 Order on bool

instantiation *bool* :: *order*
begin

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

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

instance
⟨*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 [*code func*]:
$\mathit{False} \leq b \longleftrightarrow \mathit{True}$
$\mathit{True} \leq b \longleftrightarrow b$
$\mathit{False} < b \longleftrightarrow b$
$\mathit{True} < b \longleftrightarrow \mathit{False}$
⟨*proof*⟩

3.8 Order on functions

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

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

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

instance ⟨*proof*⟩

end

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

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*⟩

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

lemma *predicate1I*:
assumes *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $P \leq Q$
⟨*proof*⟩

lemma *predicate1D* [*Pure.dest*, *dest*]: $P \leq Q \Longrightarrow P\ x \Longrightarrow Q\ x$

⟨*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-predicate1D*: $P\ x$ ==> P <= Q ==> $Q\ x$
⟨*proof*⟩

lemma *rev-predicate2D*: $P\ x\ y$ ==> P <= Q ==> $Q\ x\ y$
⟨*proof*⟩

3.9 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*⟩

end

context *linorder*
begin

lemma *min-of-mono*:
fixes $f :: 'a \Rightarrow 'b{::}linorder$
shows $mono\ f \Longrightarrow min\ (f\ m)\ (f\ n) = f\ (min\ m\ n)$
⟨*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 *LeastI2-order*:
[| *P* (*x*::'*a*::*order*);
!!*y*. *P y* ==> *x* <= *y*;
!!*x*. [| *P x*; *ALL y*. *P y* --> *x* ≤ *y* |] ==> *Q x* |]
==> *Q* (*Least P*)
⟨*proof*⟩

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*: (⋀*x*::'*a*::*order*. *least* ≤ *x*) ⟹ *min x least* = *least*
⟨*proof*⟩

lemma *max-leastR*: (⋀*x*::'*a*::*order*. *least* ≤ *x*) ⟹ *max x least* = *x*
⟨*proof*⟩

end

4 Set: Set theory for higher-order logic

theory *Set*
imports *Orderings*
begin

A set in HOL is simply a predicate.

4.1 Basic syntax

global

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

consts

{}	:: '*a set*	({})
UNIV	:: '*a set*	
insert	:: '*a* => '*a set* => '*a set*	
Collect	:: ('*a* => *bool*) => '*a set*	— comprehension
op Int	:: '*a set* => '*a set* => '*a set*	(**infixl** *Int 70*)
op Un	:: '*a set* => '*a set* => '*a set*	(**infixl** *Un 65*)
UNION	:: '*a set* => ('*a* => '*b set*) => '*b set*	— general union
INTER	:: '*a set* => ('*a* => '*b set*) => '*b set*	— general intersection

Union :: *'a set set => 'a set* — union of a set
Inter :: *'a set set => 'a set* — intersection of a set
Pow :: *'a set => 'a set set* — powerset
Ball :: *'a set => ('a => bool) => bool* — bounded universal quantifiers

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

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

local

4.2 Additional concrete syntax

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

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

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

notation (*xsymbols*)
op Int (**infixl** ∩ *70*) **and**
op Un (**infixl** ∪ *65*) **and**
op : (*op* ∈) **and**
op : ((*-/* ∈ *-*) [*50*, *51*] *50*) **and**
not-mem (*op* ∉) **and**
not-mem ((*-/* ∉ *-*) [*50*, *51*] *50*) **and**
Union (⋃ *-* [*90*] *90*) **and**
Inter (⋂ *-* [*90*] *90*)

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

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

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

translations
{x, xs} == *insert x {xs}*
{x} == *insert x {}*
{x. P} == *Collect (%x. P)*
{x:A. P} => *{x. x:A & P}*
UN x y. B == *UN x. UN y. B*
UN x. B == *UNION UNIV (%x. B)*
UN x. B == *UN x:UNIV. B*
INT x y. B == *INT x. INT y. B*
INT x. B == *INTER UNIV (%x. B)*
INT x. B == *INT x:UNIV. B*
UN x:A. B == *UNION A (%x. B)*
INT x:A. B == *INTER A (%x. B)*
ALL x:A. P == *Ball A (%x. P)*
EX x:A. P == *Bex A (%x. P)*
EX! x:A. P == *Bex1 A (%x. P)*
LEAST x:A. P => *LEAST x. x:A & P*

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

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

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

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

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

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

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

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

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

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

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

supset ≡ *greater*

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

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

4.2.1 Bounded quantifiers

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

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

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

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

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

⟨*ML*⟩

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

⟨*ML*⟩

4.3 Rules and definitions

Isomorphisms between predicates and sets.

defs
mem-def: $x : S == S\ x$
Collect-def: $Collect\ P == P$

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

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

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

instance ⟨*proof*⟩

end

instantiation *bool* :: *minus*
begin

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

instantiation *bool* :: *uminus*
begin

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

instance ⟨*proof*⟩

end

defs
Un-def: $A \ Un \ B == \{x.\ x{:}A \mid x{:}B\}$
Int-def: $A \ Int \ B == \{x.\ x{:}A \ \&\ x{:}B\}$
INTER-def: $INTER \ A \ B == \{y.\ ALL \ x{:}A.\ y{:}\ B(x)\}$
UNION-def: $UNION \ A \ B == \{y.\ EX \ x{:}A.\ y{:}\ B(x)\}$
Inter-def: $Inter \ S == (INT \ x{:}S.\ x)$
Union-def: $Union \ S == (UN \ x{:}S.\ x)$
Pow-def: $Pow \ A == \{B.\ B <= A\}$
empty-def: $\{\} == \{x.\ False\}$
UNIV-def: $UNIV == \{x.\ True\}$
insert-def: $insert \ a \ B == \{x.\ x{=}a\} \ Un \ B$
image-def: $f`A == \{y.\ EX \ x{:}A.\ y = f(x)\}$

4.4 Lemmas and proof tool setup

4.4.1 Relating predicates and sets

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

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

4.4.2 Bounded quantifiers

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*⟩

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

⟨*ML*⟩

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

⟨*ML*⟩
Gives better instantiation for bound:

⟨*ML*⟩

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*]: $(ALL\ x{:}A.\ x = a \longrightarrow P\ x) = (a{:}A \longrightarrow P\ a)$
⟨*proof*⟩

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

⟨*ML*⟩

4.4.3 Congruence rules

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

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

4.4.4 Subsets

lemma *subsetI* [*atp*,*intro*!]: $(!!x.\ x{:}A \Longrightarrow x{:}B) \Longrightarrow A \subseteq B$
⟨*proof*⟩

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

lemma *subsetD* [*elim*]: $A \subseteq B \Longrightarrow c \in A \Longrightarrow c \in B$
— Rule in Modus Ponens style.
⟨*proof*⟩

declare *subsetD* [*intro?*] — FIXME

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

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

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

⟨*ML*⟩

lemma *subsetCE* [*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*: $A \leq B = (\forall x{\in}A.\ x \in B)$ ⟨*proof*⟩

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

⟨*ML*⟩

lemma *contra-subsetD*: $A \subseteq B \Longrightarrow c \notin B \Longrightarrow c \notin A$
⟨*proof*⟩

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

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

4.4.5 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!*]: $A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow A = B$
— Anti-symmetry of the subset relation.
⟨*proof*⟩

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

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

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

lemma *equalityD2*: $A = B \Longrightarrow B \subseteq A$
⟨*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*: $A = B \Longrightarrow (A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

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

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

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

4.4.6 The universal set – UNIV

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*]: $A \subseteq UNIV$
⟨*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*: $(\bigwedge x.\ x \in A) \Longrightarrow UNIV = A$
⟨*proof*⟩

4.4.7 The empty set

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

lemma *emptyE* [*elim!*]: $a : \{\} \Longrightarrow P$
⟨*proof*⟩

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

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

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

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

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

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

4.4.8 The Powerset operator – Pow

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

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

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

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

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

4.4.9 Set complement

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

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

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

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

4.4.10 Binary union – Un

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*⟩

4.4.11 Binary intersection – Int

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*⟩

4.4.12 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*⟩

4.4.13 Augmenting a set − insert

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

4.4.14 Singletons, using insert

lemma *singletonI* [*intro*!,*noatp*]: $a : \{a\}$
— Redundant? But unlike *insertCI*, it proves the subgoal immediately!
⟨*proof*⟩

lemma *singletonD* [*dest*!,*noatp*]: $b : \{a\} \Longrightarrow b = a$
⟨*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*,*noatp*]:
$(\{b\} = \mathit{insert}\ a\ A) = (a = b\ \&\ A \subseteq \{b\})$
⟨*proof*⟩

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

lemma *subset-singletonD*: $A \subseteq \{x\} \Longrightarrow A = \{\} \mid A = \{x\}$
⟨*proof*⟩

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

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

lemma *diff-single-insert*: $A - \{x\} \subseteq B \Longrightarrow x \in A \Longrightarrow A \subseteq \mathit{insert}\ x\ B$
⟨*proof*⟩

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

4.4.15 Unions of families

UN $x{:}A.\ B\ x$ is $\bigcup B$ ‘ A.

declare *UNION-def* [*noatp*]

lemma *UN-iff* [*simp*]: $(b{:}\ (\mathit{UN}\ x{:}A.\ B\ x)) = (\mathit{EX}\ x{:}A.\ b{:}\ B\ x)$
⟨*proof*⟩

lemma *UN-I* [*intro*]: $a{:}A \Longrightarrow b{:}\ B\ a \Longrightarrow b{:}\ (\mathit{UN}\ x{:}A.\ B\ x)$
— The order of the premises presupposes that A is rigid; b may be flexible.
⟨*proof*⟩

lemma *UN-E* [*elim*!]: $b : (\mathit{UN}\ x{:}A.\ B\ x) \Longrightarrow (!!x.\ x{:}A \Longrightarrow b{:}\ B\ x \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *UN-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (\mathit{UN}\ x{:}A.\ C\ x) = (\mathit{UN}\ x{:}B.\ D$

x)
⟨*proof*⟩

4.4.16 Intersections of families

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

lemma *INT-iff* [*simp*]: $(b{:}\ (INT\ x{:}A.\ B\ x)) = (ALL\ x{:}A.\ b{:}\ B\ x)$
⟨*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*⟩

4.4.17 Union

lemma *Union-iff* [*simp*,*noatp*]: $(A : Union\ C) = (EX\ X{:}C.\ A{:}X)$
⟨*proof*⟩

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

lemma *UnionE* [*elim*!]: $A : Union\ C \Longrightarrow (!!X.\ A{:}X \Longrightarrow X{:}C \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

4.4.18 Inter

lemma *Inter-iff* [*simp*,*noatp*]: $(A : Inter\ C) = (ALL\ X{:}C.\ A{:}X)$
⟨*proof*⟩

lemma *InterI* [*intro*!]: $(!!X.\ X{:}C \Longrightarrow A{:}X) \Longrightarrow A : Inter\ C$
⟨*proof*⟩

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

lemma *InterD* [*elim*]: $A : Inter\ C \Longrightarrow X{:}C \Longrightarrow A{:}X$
⟨*proof*⟩

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

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

declare *image-def* [*noatp*]

lemma *image-eqI* [*simp*, *intro*]: $b = f\ x \Longrightarrow x{:}A \Longrightarrow b : f`A$
⟨*proof*⟩

lemma *imageI*: $x : A \Longrightarrow f\ x : f\ `\ A$
⟨*proof*⟩

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

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

lemma *image-Un*: $f`(A\ Un\ B) = f`A\ Un\ f`B$
⟨*proof*⟩

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

4.4.19 Set reasoning tools

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

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

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

lemmas *mem-simps =*
insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [*simp*] above.

⟨*ML*⟩

4.4.20 The "proper subset" relation

lemma *psubsetI* [*intro!*,*noatp*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
⟨*proof*⟩

lemma *psubsetE* [*elim!*,*noatp*]:
$[|A \subset B;\ [|A \subseteq B;\ \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
⟨*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*:
$(!!x.\ x \in A \Longrightarrow P\ x) \equiv \mathit{Trueprop}\ (\forall\, x \in A.\ P\ x)$
⟨*proof*⟩

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

4.5 Further set-theory lemmas

4.5.1 Derived rules involving subsets.

insert.

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

Big Union – least upper bound of a set.

lemma *Union-upper*: $B \in A \Longrightarrow B \subseteq \mathit{Union}\ A$
⟨*proof*⟩

lemma *Union-least*: $(!!X.\ X \in A \Longrightarrow X \subseteq C) \Longrightarrow \mathit{Union}\ A \subseteq C$
⟨*proof*⟩

General union.

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*⟩

Big Intersection – greatest lower bound of a set.

lemma *Inter-lower*: $B \in A \Longrightarrow \mathit{Inter}\ A \subseteq B$
⟨*proof*⟩

lemma *Inter-subset*:
$[|\ !!X.\ X \in A \Longrightarrow X \subseteq B;\ A \mathrel{\tilde{}}= \{\}\ |] \Longrightarrow \bigcap A \subseteq B$
⟨*proof*⟩

lemma *Inter-greatest*: $(!!X.\ X \in A \Longrightarrow C \subseteq X) \Longrightarrow C \subseteq \mathit{Inter}\ A$
⟨*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*⟩

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*⟩

4.5.2 Equalities involving union, intersection, inclusion, etc.

{}.

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

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

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

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

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

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

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

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

lemma *Collect-bex-eq* [*noatp*]: $\{x.\ \exists\, y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
⟨*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* ∈ *A* ==> *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* ⊆ *B*) = (*x* ∈ *B* & *A* ⊆ *B*)
⟨*proof*⟩

lemma *mk-disjoint-insert*: *a* ∈ *A* ==> ∃ *B*. *A* = *insert a B* & *a* ∉ *B*
— use new *B* rather than *A* − {*a*} to avoid infinite unfolding
⟨*proof*⟩

lemma *insert-Collect*: *insert a* (*Collect P*) = {*u*. *u* ≠ *a* −−> *P u*}
⟨*proof*⟩

lemma *UN-insert-distrib*: *u* ∈ *A* ==> ($\bigcup x{\in}A$. *insert a* (*B x*)) = *insert a* ($\bigcup x{\in}A$. *B x*)
⟨*proof*⟩

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

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

lemma *disjoint-insert* [*simp*,*noatp*]:
$(B \cap \mathit{insert}\ a\ A = \{\}) = (a \notin B \wedge B \cap A = \{\})$
$(\{\} = A \cap \mathit{insert}\ b\ B) = (b \notin A \wedge \{\} = A \cap B)$
⟨*proof*⟩

image.

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

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

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

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

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

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

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

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

lemma *range-composition* [*simp*]: $\mathit{range}\ (\lambda x.\ f\ (g\ x)) = f`\mathit{range}\ g$

⟨*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-eq-Inter*: $A \cap B = \bigcap \{A, B\}$
⟨*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*,*noatp*]: $(A \cap B = UNIV) = (A = UNIV \ \& \ B = UNIV)$
⟨*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*: $B \subseteq A \Longrightarrow A \cup B = A$
⟨*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-eq-Union*: $A \cup B = \bigcup \{A, B\}$
⟨*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-right*:
$A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
⟨*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 *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*⟩

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*⟩

Union.

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

lemma *Union-UNIV* [*simp*]: $Union\ UNIV = UNIV$
⟨*proof*⟩

lemma *Union-insert* [*simp*]: $\bigcup (\mathit{insert}\ a\ B) = a \cup \bigcup B$
⟨*proof*⟩

lemma *Union-Un-distrib* [*simp*]: $\bigcup (A\ \mathit{Un}\ B) = \bigcup A \cup \bigcup B$
⟨*proof*⟩

lemma *Union-Int-subset*: $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$
⟨*proof*⟩

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

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

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

Inter.

lemma *Inter-empty* [*simp*]: $\bigcap \{\} = \mathit{UNIV}$
⟨*proof*⟩

lemma *Inter-UNIV* [*simp*]: $\bigcap \mathit{UNIV} = \{\}$
⟨*proof*⟩

lemma *Inter-insert* [*simp*]: $\bigcap (\mathit{insert}\ a\ B) = a \cap \bigcap B$
⟨*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*,*noatp*]:
$(\bigcap A = \mathit{UNIV}) = (\forall x{\in}A.\ x = \mathit{UNIV})$
$(\mathit{UNIV} = \bigcap A) = (\forall x{\in}A.\ x = \mathit{UNIV})$
⟨*proof*⟩

UN and *INT*.

Basic identities:

lemma *UN-empty* [*simp*,*noatp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
⟨*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 *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 *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 *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 *Union-image-eq* [*simp*]: $\bigcup(B`A) = (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

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

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

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

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

⟨*proof*⟩

lemma *UN-eq*: $(\bigcup x{\in}A.\ B\ x) = \bigcup(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
⟨*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 *UNION-empty-conv*[*simp*]:
$(\{\} = (UN\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = \{\})$
$((UN\ x{:}A.\ B\ x) = \{\}) = (\forall x{\in}A.\ B\ x = \{\})$
⟨*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*⟩

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*⟩

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*⟩

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*⟩

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
⟨*proof*⟩

lemma *Diff-eq-empty-iff* [*simp*,*noatp*]: $(A - B = \{\}) = (A \subseteq B)$
⟨*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*,*noatp*]: $x \notin A \Longrightarrow A - insert\ x\ B = A - B$
⟨*proof*⟩

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

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

lemma *insert-Diff-if*: $insert\ x\ A - B = (if\ x \in B\ then\ A - B\ else\ insert\ x\ (A - B))$
⟨*proof*⟩

lemma *insert-Diff1* [*simp*]: $x \in B \Longrightarrow insert\ x\ A - B = A - B$
⟨*proof*⟩

lemma *insert-Diff-single*[*simp*]: $insert\ a\ (A - \{a\}) = insert\ a\ A$
⟨*proof*⟩

lemma *insert-Diff*: $a \in A \Longrightarrow insert\ a\ (A - \{a\}) = A$
⟨*proof*⟩

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

lemma *Un-eq-UN*: $A \cup B = (\bigcup b.\ if\ b\ then\ A\ else\ B)$
⟨*proof*⟩

lemma *UN-bool-eq*: $(\bigcup b{::}bool.\ A\ b) = (A\ True \cup A\ False)$
⟨*proof*⟩

lemma *INT-bool-eq*: $(\bigcap b{::}bool.\ A\ b) = (A\ True \cap A\ False)$
⟨*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 *UN-Pow-subset*: $(\bigcup x{\in}A.\ Pow\ (B\ x)) \subseteq Pow\ (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *subset-Pow-Union*: $A \subseteq Pow\ (\bigcup A)$
⟨*proof*⟩

lemma *Union-Pow-eq* [*simp*]: $\bigcup(Pow\ A) = A$
⟨*proof*⟩

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

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

Miscellany.

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

lemma *subset-iff*: $(A \subseteq B) = (\forall t.\ t \in A \dashrightarrow t \in B)$
⟨*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*⟩

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*,*noatp*]:
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ |\ Q) = ((ALL\ x{:}A.\ P\ x)\ |\ Q)$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ |\ Q\ x) = (P\ |\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ --{>}\ Q\ x) = (P\ --{>}\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ --{>}\ Q) = ((EX\ x{:}A.\ P\ x)\ --{>}\ Q)$
!!$P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
!!$P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
!!$a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
!!$A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$
!!$P\ Q.\ (ALL\ x{:}Collect\ Q.\ P\ x) = (ALL\ x.\ Q\ x\ --{>}\ P\ x)$
!!$A\ P\ f.\ (ALL\ x{:}f`A.\ P\ x) = (ALL\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(ALL\ x{:}A.\ P\ x)) = (EX\ x{:}A.\ \sim P\ x)$
⟨*proof*⟩

lemma *bex-simps* [*simp*,*noatp*]:
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ x\ \&\ Q) = ((EX\ x{:}A.\ P\ x)\ \&\ Q)$
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x{:}A.\ Q\ x))$
!!$P.\ (EX\ x{:}\{\}.\ P\ x) = False$

!!P. $(EX\ x{:}UNIV.\ P\ x) = (EX\ x.\ P\ x)$
!!$a\ B\ P$. $(EX\ x{:}insert\ a\ B.\ P\ x) = (P(a) \mid (EX\ x{:}B.\ P\ x))$
!!$A\ P$. $(EX\ x{:}Union\ A.\ P\ x) = (EX\ y{:}A.\ EX\ x{:}y.\ P\ x)$
!!$A\ B\ P$. $(EX\ x{:}\ UNION\ A\ B.\ P\ x) = (EX\ a{:}A.\ EX\ x{:}B\ a.\ P\ x)$
!!$P\ Q$. $(EX\ x{:}Collect\ Q.\ P\ x) = (EX\ x.\ Q\ x\ \&\ P\ x)$
!!$A\ P\ f$. $(EX\ x{:}f`A.\ P\ x) = (EX\ x{:}A.\ P\ (f\ x))$
!!$A\ P$. $(\sim(EX\ x{:}A.\ P\ x)) = (ALL\ x{:}A.\ \sim P\ x)$
⟨*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*:
$(EX\ x{:}A.\ P\ x \mid Q\ x) = ((EX\ x{:}A.\ P\ x) \mid (EX\ x{:}A.\ Q\ x))$
⟨*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*⟩

4.5.3 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 *Union-mono*: $A \subseteq B \Longrightarrow \bigcup A \subseteq \bigcup B$
⟨*proof*⟩

lemma *Inter-anti-mono*: $B \subseteq A \Longrightarrow \bigcap A \subseteq \bigcap B$
⟨*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 *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 *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*: $P1 \longrightarrow Q1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1 \mid P2) \longrightarrow (Q1 \mid Q2)$
⟨*proof*⟩

lemma *imp-mono*: *Q1* −−> *P1* ==> *P2* −−> *Q2* ==> (*P1* −−> *P2*) −−> (*Q1* −−> *Q2*)
⟨*proof*⟩

lemma *imp-refl*: *P* −−> *P* ⟨*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*⟩

lemma *eq-to-mono2*: *a* = *b* ==> *c* = *d* ==> ~ *b* −−> ~ *d* ==> ~ *a* −−> ~ *c*
⟨*proof*⟩

4.6 Inverse image of a function

constdefs
vimage :: ('*a* => '*b*) => '*b set* => '*a set* (**infixr** −‘ *90*)
f −‘ *B* == {*x*. *f x* : *B*}

4.6.1 Basic rules

lemma *vimage-eq* [*simp*]: (*a* : *f* −‘ *B*) = (*f a* : *B*)
⟨*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* ==> *a* : *f* −‘ *A*
⟨*proof*⟩

lemma *vimageE* [*elim*!]: a: f $-`$ B ==> (!!x. f a = x ==> x:B ==> P) ==> P
⟨*proof*⟩

lemma *vimageD*: a : f $-`$ A ==> f a : A
⟨*proof*⟩

4.6.2 Equations

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-Union*: f $-`$ $(Union$ $A)$ = $(UN$ $X$:$A$. $f$ $-`$ $X)$
⟨*proof*⟩

lemma *vimage-UN*: $f-`(UN$ $x$:$A$. $B$ $x)$ = $(UN$ $x$:$A$. $f$ $-`$ B $x)$
⟨*proof*⟩

lemma *vimage-INT*: $f-`(INT$ $x$:$A$. $B$ $x)$ = $(INT$ $x$:$A$. $f$ $-`$ B $x)$
⟨*proof*⟩

lemma *vimage-Collect-eq* [*simp*]: f $-`$ $Collect$ P = {y. P $(f$ $y)$}
⟨*proof*⟩

lemma *vimage-Collect*: (!!x. P $(f$ $x)$ = Q $x)$ ==> 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-eq-UN*: $f-`B$ = $(UN$ $y$: $B$. $f-`\{y\})$
— NOT suitable for rewriting
⟨*proof*⟩

lemma *vimage-mono*: $A \subseteq B \Longrightarrow f -` A \subseteq f -` B$
— monotonicity
⟨*proof*⟩

lemma *vimage-image-eq* [*noatp*]: $f -` (f ` A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
⟨*proof*⟩

lemma *image-vimage-subset*: $f ` (f -` A) <= A$
⟨*proof*⟩

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

lemma *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*⟩

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

4.7 Getting the Contents of a Singleton Set

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

lemma *contents-eq* [*simp*]: $contents\ \{x\} = x$
⟨*proof*⟩

4.8 Transitivity rules for calculational reasoning

lemma *set-rev-mp*: $x{:}A \Longrightarrow A \subseteq B \Longrightarrow x{:}B$
⟨*proof*⟩

lemma *set-mp*: $A \subseteq B \Longrightarrow x{:}A \Longrightarrow x{:}B$
⟨*proof*⟩

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

4.9 Dense orders

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

begin

lemma *interval-empty-iff*:
$\{y.\ x < y \wedge y < z\} = \{\} \longleftrightarrow \neg\ x < z$
⟨*proof*⟩

end

4.10 Least value operator

lemma *Least-mono*:
mono (*f*::'*a*::*order* => '*b*::*order*) ==> *EX x*:*S*. *ALL y*:*S*. *x* <= *y*
==> (*LEAST y*. *y* : *f* ' *S*) = *f* (*LEAST x*. *x* : *S*)
— Courtesy of Stephan Merz
⟨*proof*⟩

lemma *Least-equality*:
[| *P* (*k*::'*a*::*order*); !!*x*. *P x* ==> *k* <= *x* |] ==> (*LEAST x*. *P x*) = *k*
⟨*proof*⟩

4.11 Basic ML bindings

⟨*ML*⟩

end

5 Fun: Notions about functions

theory *Fun*
imports *Set*
begin

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

lemma *expand-fun-eq*: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$
⟨*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*⟩

5.1 The Identity Function *id*

definition
id :: '*a* ⇒ '*a*
where
id = (λ*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*⟩

5.2 The Composition Operator *f ∘ g*

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

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

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

compatibility

lemmas *o-def = comp-def*

lemma *o-apply* [*simp*]: *(f o g) x = f (g x)*
⟨*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 *image-compose*: *(f o g) ‘ r = f‘(g‘r)*
⟨*proof*⟩

lemma *UN-o*: *UNION A (g o f) = UNION (f‘A) g*
⟨*proof*⟩

5.3 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*⟩

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

5.4 Injectivity and Surjectivity

constdefs
inj-on :: [$'a$ => $'b$, $'a$ *set*] => *bool* — injective
inj-on $f\ A$ == ! x:A. ! y:A. $f(x)$=$f(y)$ −−> x=y

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

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

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

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

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

lemma *injI*:
assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
shows *inj* f
⟨*proof*⟩

For Proofs in *Tools/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-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-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-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*⟩

5.5 Function Updating

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

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

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

lemma *fun-upd-idem-iff*: *(f(x:=y) = f) = (f x = y)*
⟨*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*: $z^{\sim}{=}x \Longrightarrow (f(x{:=}y))\ z = f\ z$
⟨*proof*⟩

lemma *fun-upd-upd* [*simp*]: $f(x{:=}y,x{:=}z) = f(x{:=}z)$
⟨*proof*⟩

lemma *fun-upd-twist*: $a \mathrel{\sim=} c \Longrightarrow (m(a{:=}b))(c{:=}d) = (m(c{:=}d))(a{:=}b)$
⟨*proof*⟩

lemma *inj-on-fun-updI*: $[\![\ \mathit{inj\text{-}on}\ f\ A;\ y \notin f`A\ ]\!] \Longrightarrow \mathit{inj\text{-}on}\ (f(x{:=}y))\ A$
⟨*proof*⟩

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

5.6 *override-on*

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

lemma *override-on-emptyset*[*simp*]: $\mathit{override\text{-}on}\ f\ g\ \{\} = f$
⟨*proof*⟩

lemma *override-on-apply-notin*[*simp*]: $a \mathrel{\sim:} A \Longrightarrow (\mathit{override\text{-}on}\ f\ g\ A)\ a = f\ a$
⟨*proof*⟩

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

5.7 *swap*

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

lemma *swap-self*: $\mathit{swap}\ a\ a\ f = f$
⟨*proof*⟩

lemma *swap-commute*: $\mathit{swap}\ a\ b\ f = \mathit{swap}\ b\ a\ f$
⟨*proof*⟩

lemma *swap-nilpotent* [*simp*]: $\mathit{swap}\ a\ b\ (\mathit{swap}\ a\ b\ f) = f$
⟨*proof*⟩

lemma *inj-on-imp-inj-on-swap*:
$[|\mathit{inj\text{-}on}\ f\ A;\ a \in A;\ b \in A|] \Longrightarrow \mathit{inj\text{-}on}\ (\mathit{swap}\ a\ b\ f)\ A$

⟨*proof*⟩

lemma *inj-on-swap-iff* [*simp*]:
assumes *A*: $a \in A$ $b \in A$ **shows** *inj-on* (*swap a b f*) *A* = *inj-on f A*
⟨*proof*⟩

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*⟩

5.8 Proof tool setup

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

⟨*ML*⟩

5.9 Code generator setup

types-code
fun ((- ->/ -))
attach (*term-of*) ⟪
fun term-of-fun-type - aT - bT - = Free (<function>, aT --> bT);
⟫
attach (*test*) ⟪
fun gen-fun-type aF aT bG bT i =
let
val tab = ref [];
fun mk-upd (x, (-, y)) t = Const (Fun.fun-upd,
(aT --> bT) --> aT --> bT --> aT --> bT) $ t $ aF x $ y ()
in
(fn x =>
case AList.lookup op = (!tab) x of
NONE =>
let val p as (y, -) = bG i
in (tab := (x, p) :: !tab; y) end
| SOME (y, -) => y,
fn () => Basics.fold mk-upd (!tab) (Const (arbitrary, aT --> bT)))
end;
⟫

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

code-const *id*

(*Haskell id*)

end

6 Lattices: Abstract lattices

theory *Lattices*
imports *Fun*
begin

6.1 Lattices

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

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

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

Dual lattice

lemma *dual-lattice*:
lower-semilattice (*op* $\geq$) (*op* $>$) *sup*
⟨*proof*⟩

end

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

6.1.1 Intro and elim rules

context *lower-semilattice*
begin

lemma *le-infI1*[*intro*]:
assumes $a \sqsubseteq x$
shows $a \sqcap b \sqsubseteq x$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-infI1*

lemma *le-infI2*[*intro*]:
assumes $b \sqsubseteq x$
shows $a \sqcap b \sqsubseteq x$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-infI2*

lemma *le-infI*[*intro*!]: $x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcap b$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-infI*

lemma *le-infE* [*elim*!]: $x \sqsubseteq a \sqcap b \Longrightarrow (x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-infE*

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

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

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

end

context *upper-semilattice*
begin

lemma *le-supI1*[*intro*]: $x \sqsubseteq a \Longrightarrow x \sqsubseteq a \sqcup b$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-supI1*

lemma *le-supI2*[*intro*]: $x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcup b$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-supI2*

lemma *le-supI*[*intro*!]: $a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow a \sqcup b \sqsubseteq x$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-supI*

lemma *le-supE*[*elim*!]: $a \sqcup b \sqsubseteq x \Longrightarrow (a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩
lemmas (**in** −) [*rule del*] = *le-supE*

lemma *ge-sup-conv*[*simp*]:

$x \sqcup y \sqsubseteq z = (x \sqsubseteq z \wedge y \sqsubseteq z)$
⟨*proof*⟩

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

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

end

6.1.2 Equational laws

context *lower-semilattice*
begin

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

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

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

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

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

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

end

context *upper-semilattice*
begin

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

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

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

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

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

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

end

context *lattice*
begin

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

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

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

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

Towards distributivity

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

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

end

6.2 Distributive lattices

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

context *distrib-lattice*
begin

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

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

end

6.3 Uniqueness of inf and sup

lemma (**in** *lower-semilattice*) *inf-unique*:
  fixes *f* (**infixl** $\triangle$ *70*)
  assumes *le1*: $\bigwedge x\ y.\ x \triangle y \leq x$ **and** *le2*: $\bigwedge x\ y.\ x \triangle y \leq y$
  and *greatest*: $\bigwedge x\ y\ z.\ x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \triangle z$
  shows $x \sqcap y = x \triangle y$
⟨*proof*⟩

lemma (**in** *upper-semilattice*) *sup-unique*:
  fixes *f* (**infixl** ∇ *70*)
  assumes *ge1* [*simp*]: $\bigwedge x\ y.\ x \leq x \nabla y$ **and** *ge2*: $\bigwedge x\ y.\ y \leq x \nabla y$
  and *least*: $\bigwedge x\ y\ z.\ y \leq x \Longrightarrow z \leq x \Longrightarrow y \nabla z \leq x$
  shows $x \sqcup y = x \nabla y$

⟨*proof*⟩

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

lemma (**in** *linorder*) *distrib-lattice-min-max*:
distrib-lattice (*op* ≤) (*op* <) *min max*
⟨*proof*⟩

interpretation *min-max*:
distrib-lattice [*op* ≤ :: 'a::*linorder* ⇒ 'a ⇒ *bool op* < *min max*]
⟨*proof*⟩

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

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

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

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

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

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

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

6.5 Complete lattices

class *complete-lattice* = *lattice* +
fixes *Inf* :: 'a *set* ⇒ 'a (⨅- [900] 900)
and *Sup* :: 'a *set* ⇒ 'a (⨆- [900] 900)
assumes *Inf-lower*: $x \in A \Longrightarrow \bigsqcap A \sqsubseteq x$
and *Inf-greatest*: $(\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x) \Longrightarrow z \sqsubseteq \bigsqcap A$
assumes *Sup-upper*: $x \in A \Longrightarrow x \sqsubseteq \bigsqcup A$
and *Sup-least*: $(\bigwedge x.\ x \in A \Longrightarrow x \sqsubseteq z) \Longrightarrow \bigsqcup A \sqsubseteq z$
begin

lemma *Inf-Sup*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \leq a\}$
⟨*proof*⟩

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \leq b\}$
⟨*proof*⟩

lemma *Inf-Univ*: $\bigsqcap UNIV = \bigsqcup \{\}$
⟨*proof*⟩

lemma *Sup-Univ*: $\bigsqcup UNIV = \bigsqcap \{\}$
⟨*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-insert-simp*:
$\bigsqcap insert\ a\ A = (if\ A = \{\}\ then\ a\ else\ a \sqcap \bigsqcap A)$
⟨*proof*⟩

lemma *Sup-insert-simp*:
$\bigsqcup insert\ a\ A = (if\ A = \{\}\ then\ a\ else\ a \sqcup \bigsqcup A)$
⟨*proof*⟩

lemma *Inf-binary*:
$\bigsqcap \{a, b\} = a \sqcap b$
⟨*proof*⟩

lemma *Sup-binary*:
$\bigsqcup \{a, b\} = a \sqcup b$
⟨*proof*⟩

definition
top :: $'a$ **where**
$top = \bigsqcap \{\}$

definition
bot :: $'a$ **where**
$bot = \bigsqcup \{\}$

lemma *top-greatest* [*simp*]: $x \leq top$
⟨*proof*⟩

lemma *bot-least* [*simp*]: $bot \leq x$

⟨*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*, *10*] *10*)
-INF1 :: *pttrns* => *'b* => *'b* ((*3INF -./ -*) [*0*, *10*] *10*)
-INF :: *pttrn* => *'a set* => *'b* => *'b* ((*3INF -:-./ -*) [*0*, *10*] *10*)

translations
SUP x y. B == *SUP x. SUP y. B*
SUP x. B == *CONST SUPR UNIV* (%*x. B*)
SUP x. B == *SUP x*:*UNIV. B*
SUP x:*A. B* == *CONST SUPR A* (%*x. B*)
INF x y. B == *INF x. INF y. B*
INF x. B == *CONST INFI UNIV* (%*x. B*)
INF x. B == *INF x*:*UNIV. B*
INF x:*A. B* == *CONST INFI A* (%*x. B*)

⟨*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-const*[*simp*]: *A* ≠ {} ⟹ (*SUP i*:*A. M*) = *M*

⟨*proof*⟩

lemma *INF-const*[*simp*]: $A \neq \{\} \Longrightarrow (INF\ i{:}A.\ M) = M$
⟨*proof*⟩

end

6.6 Bool as lattice

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

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

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

instance
⟨*proof*⟩

end

instantiation *bool* :: *complete-lattice*
begin

definition
Inf-bool-def: $\bigsqcap A \longleftrightarrow (\forall x{\in}A.\ x)$

definition
Sup-bool-def: $\bigsqcup A \longleftrightarrow (\exists x{\in}A.\ x)$

instance
⟨*proof*⟩

end

lemma *Inf-empty-bool* [*simp*]:
$\bigsqcap \{\}$
⟨*proof*⟩

lemma *not-Sup-empty-bool* [*simp*]:
$\neg$ *Sup* $\{\}$
⟨*proof*⟩

lemma *top-bool-eq*: *top* = *True*
⟨*proof*⟩

lemma *bot-bool-eq*: *bot* = *False*

⟨*proof*⟩

6.7 Fun as lattice

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

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

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

instance
⟨*proof*⟩

end

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

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

definition
Inf-fun-def [*code func del*]: $\bigsqcap A = (\lambda x.\ \bigsqcap \{y.\ \exists f{\in}A.\ y = f\ x\})$

definition
Sup-fun-def [*code func del*]: $\bigsqcup A = (\lambda x.\ \bigsqcup \{y.\ \exists f{\in}A.\ y = f\ x\})$

instance
⟨*proof*⟩

end

lemma *Inf-empty-fun*:
$\bigsqcap \{\} = (\lambda\text{-}.\ \bigsqcap \{\})$
⟨*proof*⟩

lemma *Sup-empty-fun*:
$\bigsqcup \{\} = (\lambda\text{-}.\ \bigsqcup \{\})$
⟨*proof*⟩

lemma *top-fun-eq*: $top = (\lambda x.\ top)$
⟨*proof*⟩

lemma *bot-fun-eq*: $bot = (\lambda x.\ bot)$
⟨*proof*⟩

6.8 Set as lattice

lemma *inf-set-eq*: $A \sqcap B = A \cap B$
⟨*proof*⟩

lemma *sup-set-eq*: $A \sqcup B = A \cup B$
⟨*proof*⟩

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

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

lemma *Inf-set-eq*: $\bigsqcap S = \bigcap S$
⟨*proof*⟩

lemma *Sup-set-eq*: $\bigsqcup S = \bigcup S$
⟨*proof*⟩

lemma *top-set-eq*: $top = UNIV$
⟨*proof*⟩

lemma *bot-set-eq*: $bot = \{\}$
⟨*proof*⟩

redundant bindings

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

no-notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
Inf ($\bigsqcap$- [*900*] *900*) **and**
Sup ($\bigsqcup$- [*900*] *900*)

end

7 Typedef: HOL type definitions

theory *Typedef*
imports *Set*
uses
(*Tools/typedef-package.ML*)
(*Tools/typecopy-package.ML*)
(*Tools/typedef-codegen.ML*)

begin

⟨*ML*⟩

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*:
  shows *range Rep* = *A*
⟨*proof*⟩

end

⟨*ML*⟩

This class is just a workaround for classes without parameters; it shall disappear as soon as possible.

class *itself* = *type* +
  fixes *itself* :: *'a itself*

⟨*ML*⟩

instantiation *bool* :: *itself*
begin

definition *itself* = *TYPE*(*bool*)

instance ⟨*proof*⟩

end

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

definition *itself* = *TYPE*(*'a* ⇒ *'b*)

instance ⟨*proof*⟩

end

hide (**open**) *const itself*

end

8 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*
imports *Typedef Fun*
begin

The representations of the two injections

constdefs
  Inl-Rep :: [*'a*, *'a*, *'b*, *bool*] => *bool*
  Inl-Rep == (%*a*. %*x y p*. *x*=*a* & *p*)

  Inr-Rep :: [*'b*, *'a*, *'b*, *bool*] => *bool*
  Inr-Rep == (%*b*. %*x y p*. *y*=*b* & ~*p*)

global

typedef (*Sum*)
(*'a*, *'b*) + (**infixr** + *10*)
= {*f*. (*? a*. *f* = *Inl-Rep*(*a*::*'a*)) | (*? b*. *f* = *Inr-Rep*(*b*::*'b*))}
⟨*proof*⟩

local

abstract constants and syntax

constdefs
Inl :: *'a* => *'a* + *'b*
Inl == (%*a*. *Abs-Sum*(*Inl-Rep*(*a*)))

Inr :: *'b* => *'a* + *'b*
Inr == (%*b*. *Abs-Sum*(*Inr-Rep*(*b*)))

Plus :: [*'a set*, *'b set*] => (*'a* + *'b*) *set* (**infixr** <+> *65*)
A <+> *B* == (*Inl*‘*A*) *Un* (*Inr*‘*B*)
— disjoint sum for sets; the operator + is overloaded with wrong type!

Part :: [*'a set*, *'b* => *'a*] => *'a set*
Part A h == *A Int* {*x*. *? z*. *x* = *h*(*z*)}
— for selecting out the components of a mutually recursive definition

lemma *Inl-RepI*: *Inl-Rep*(*a*) : *Sum*
⟨*proof*⟩

lemma *Inr-RepI*: *Inr-Rep*(*b*) : *Sum*
⟨*proof*⟩

lemma *inj-on-Abs-Sum*: *inj-on Abs-Sum Sum*
⟨*proof*⟩

8.1 Freeness Properties for *Inl* and *Inr*

Distinctness

lemma *Inl-Rep-not-Inr-Rep*: *Inl-Rep*(*a*) ~= *Inr-Rep*(*b*)
⟨*proof*⟩

lemma *Inl-not-Inr* [*iff*]: *Inl*(*a*) ~= *Inr*(*b*)
⟨*proof*⟩

lemmas *Inr-not-Inl* = *Inl-not-Inr* [*THEN not-sym*, *standard*]
declare *Inr-not-Inl* [*iff*]

lemmas *Inl-neq-Inr* = *Inl-not-Inr* [*THEN notE*, *standard*]
lemmas *Inr-neq-Inl* = *sym* [*THEN Inl-neq-Inr*, *standard*]

Injectiveness

lemma *Inl-Rep-inject*: *Inl-Rep*(*a*) = *Inl-Rep*(*c*) ==> *a*=*c*
⟨*proof*⟩

lemma *Inr-Rep-inject*: *Inr-Rep*(*b*) = *Inr-Rep*(*d*) ==> *b*=*d*
⟨*proof*⟩

lemma *inj-Inl*: *inj*(*Inl*)
⟨*proof*⟩
lemmas *Inl-inject* = *inj-Inl* [*THEN injD*, *standard*]

lemma *inj-Inr*: *inj*(*Inr*)
⟨*proof*⟩

lemmas *Inr-inject* = *inj-Inr* [*THEN injD*, *standard*]

lemma *Inl-eq* [*iff*]: (*Inl*(*x*)=*Inl*(*y*)) = (*x*=*y*)
⟨*proof*⟩

lemma *Inr-eq* [*iff*]: (*Inr*(*x*)=*Inr*(*y*)) = (*x*=*y*)
⟨*proof*⟩

8.2 Projections

definition
sum-case f g x =
(*if* (∃!*y*. *x* = *Inl y*)
then f (*THE y*. *x* = *Inl y*)
else g (*THE y*. *x* = *Inr y*))
definition *Projl x* = *sum-case id arbitrary x*
definition *Projr x* = *sum-case arbitrary id x*

lemma *sum-cases*[*simp*]:
sum-case f g (*Inl x*) = *f x*
sum-case f g (*Inr y*) = *g y*
⟨*proof*⟩

lemma *Projl-Inl*[*simp*]: *Projl* (*Inl x*) = *x*
⟨*proof*⟩

lemma *Projr-Inr*[*simp*]: *Projr* (*Inr x*) = *x*
⟨*proof*⟩

8.3 The Disjoint Sum of Sets

lemma *InlI* [*intro*!]: *a* : *A* ==> *Inl*(*a*) : *A* <+> *B*

⟨*proof*⟩

lemma *InrI* [*intro*!]: $b : B \Longrightarrow Inr(b) : A <+> B$
⟨*proof*⟩

lemma *PlusE* [*elim*!]:
$[\!| u: A <+> B;$
$\quad !!x.\ [\!| x{:}A;\ u{=}Inl(x) |\!] \Longrightarrow P;$
$\quad !!y.\ [\!| y{:}B;\ u{=}Inr(y) |\!] \Longrightarrow P$
$|\!] \Longrightarrow P$
⟨*proof*⟩

Exhaustion rule for sums, a degenerate form of induction

lemma *sumE*:
$[\!| !!x{::}'a.\ s = Inl(x) \Longrightarrow P;\ !!y{::}'b.\ s = Inr(y) \Longrightarrow P$
$|\!] \Longrightarrow P$
⟨*proof*⟩

lemma *sum-induct*: $[\!| !!x.\ P\ (Inl\ x); !!x.\ P\ (Inr\ x) |\!] \Longrightarrow P\ x$
⟨*proof*⟩

lemma *UNIV-Plus-UNIV* [*simp*]: $UNIV <+> UNIV = UNIV$
⟨*proof*⟩

8.4 The *Part* Primitive

lemma *Part-eqI* [*intro*]: $[\!| a : A;\ a{=}h(b) |\!] \Longrightarrow a : Part\ A\ h$
⟨*proof*⟩

lemmas *PartI* = *Part-eqI* [*OF - refl*, *standard*]

lemma *PartE* [*elim*!]: $[\!| a : Part\ A\ h;\ !!z.\ [\!| a : A;\ a{=}h(z) |\!] \Longrightarrow P |\!] \Longrightarrow P$
⟨*proof*⟩

lemma *Part-subset*: $Part\ A\ h <= A$
⟨*proof*⟩

lemma *Part-mono*: $A<=B \Longrightarrow Part\ A\ h <= Part\ B\ h$
⟨*proof*⟩

lemmas *basic-monos* = *basic-monos Part-mono*

lemma *PartD1*: $a : Part\ A\ h \Longrightarrow a : A$
⟨*proof*⟩

lemma *Part-id*: *Part A* (%*x*. *x*) = *A*
⟨*proof*⟩

lemma *Part-Int*: *Part* (*A Int B*) *h* = (*Part A h*) *Int* (*Part B h*)
⟨*proof*⟩

lemma *Part-Collect*: *Part* (*A Int* {*x*. *P x*}) *h* = (*Part A h*) *Int* {*x*. *P x*}
⟨*proof*⟩

⟨*ML*⟩

end

9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Lattices Sum-Type*
uses
(*Tools/inductive-package.ML*)
Tools/dseq.ML
(*Tools/inductive-codegen.ML*)
(*Tools/datatype-aux.ML*)
(*Tools/datatype-prop.ML*)
(*Tools/datatype-rep-proofs.ML*)
(*Tools/datatype-abs-proofs.ML*)
(*Tools/datatype-case.ML*)
(*Tools/datatype-package.ML*)
(*Tools/old-primrec-package.ML*)
(*Tools/primrec-package.ML*)
(*Tools/datatype-codegen.ML*)
begin

9.1 Least and greatest fixed points

context *complete-lattice*
begin

definition
lfp :: ('*a* ⇒ '*a*) ⇒ '*a* **where**
lfp f = *Inf* {*u*. *f u* ≤ *u*} — least fixed point

definition
gfp :: ('*a* ⇒ '*a*) ⇒ '*a* **where**
gfp f = *Sup* {*u*. *u* ≤ *f u*} — greatest fixed point

9.2 Proof of Knaster-Tarski Theorem using *lfp*

lfp f is the least upper bound of the set $\{u.\ f\ u \leq u\}$

lemma *lfp-lowerbound*: $f\ A \leq A \Longrightarrow lfp\ f \leq A$
⟨*proof*⟩

lemma *lfp-greatest*: $(!!u.\ f\ u \leq u \Longrightarrow A \leq u) \Longrightarrow A \leq lfp\ f$
⟨*proof*⟩

end

lemma *lfp-lemma2*: $mono\ f \Longrightarrow f\ (lfp\ f) \leq lfp\ f$
⟨*proof*⟩

lemma *lfp-lemma3*: $mono\ f \Longrightarrow lfp\ f \leq f\ (lfp\ f)$
⟨*proof*⟩

lemma *lfp-unfold*: $mono\ f \Longrightarrow lfp\ f = f\ (lfp\ f)$
⟨*proof*⟩

lemma *lfp-const*: $lfp\ (\lambda x.\ t) = t$
⟨*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*: $!!x.\ [|\ x{:}\ f(lfp(f)\ Int\ \{x.\ P(x)\})\ |] \Longrightarrow P(x)$
shows $P(a)$
⟨*proof*⟩

lemma *lfp-ordinal-induct*:
fixes f :: $'a{::}complete\text{-}lattice \Rightarrow 'a$
assumes *mono*: $mono\ f$
and *P-f*: $\bigwedge S.\ P\ S \Longrightarrow P\ (f\ S)$
and *P-Union*: $\bigwedge M.\ \forall S{\in}M.\ P\ S \Longrightarrow P\ (Sup\ M)$
shows $P\ (lfp\ f)$
⟨*proof*⟩

lemma *lfp-ordinal-induct-set*:
assumes *mono*: $mono\ f$
and *P-f*: $!!S.\ P\ S \Longrightarrow P(f\ S)$
and *P-Union*: $!!M.\ !S{:}M.\ P\ S \Longrightarrow P(Union\ M)$

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*:
[| $A == lfp(f)$; $mono(f)$;
$f\ (inf\ A\ P) \leq P$
|] ==> $A \leq P$
⟨*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. $f\ Z \leq g\ Z$) ==> $lfp\ f \leq lfp\ g$
⟨*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*: $mono\ f$ ==> $f\ (gfp\ f) \leq gfp\ f$
⟨*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* |] ==> $sup\ X\ (gfp\ f) \leq f\ (sup\ X\ (gfp\ f))$
⟨*proof*⟩

strong version, thanks to Coen and Frost

lemma *coinduct-set*: [| *mono*(*f*); *a*: *X*; $X \subseteq f(X\ Un\ gfp(f))$ |] ==> $a : gfp(f)$
⟨*proof*⟩

lemma *coinduct*: [| *mono*(*f*); $X \leq f\ (sup\ X\ (gfp\ f))$ |] ==> $X \leq gfp(f)$
⟨*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 $X \subseteq f\ X$ to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: *mono*(*f*) ==> *mono*(%*x*. $f(x)\ Un\ X\ Un\ B$)
⟨*proof*⟩

lemma *coinduct3-lemma*:
[| $X \subseteq f(lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)))$; *mono*(*f*) |]
==> $lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)) \subseteq f(lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)))$
⟨*proof*⟩

lemma *coinduct3*:
[| *mono*(*f*); *a*:*X*; $X \subseteq f(lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)))$ |] ==> $a : gfp(f)$
⟨*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*); $X \leq f(sup\ X\ A)$ |] ==> $X \leq A$
⟨*proof*⟩

lemma *def-coinduct-set*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; $X \subseteq f(X\ Un\ A)$ |] ==> *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* ≤ *g Z*) ==> *gfp f* ≤ *gfp g*
⟨*proof*⟩

9.7 Inductive predicates and sets

Inversion of injective functions.

constdefs
myinv :: ('*a* => '*b*) => ('*b* => '*a*)
myinv (*f* :: '*a* => '*b*) == λ*y*. *THE x*. *f x* = *y*

lemma *myinv-f-f*: *inj f* ==> *myinv f* (*f x*) = *x*
⟨*proof*⟩

lemma *f-myinv-f*: *inj f* ==> *y* ∈ *range f* ==> *f* (*myinv f y*) = *y*
⟨*proof*⟩

hide *const myinv*

Package setup.

theorems *basic-monos* =
subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
Collect-mono in-mono vimage-mono
imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

⟨*ML*⟩

theorems [*mono*] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

9.8 Inductive datatypes and primitive recursion

Package setup.

⟨*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 Product-Type: Cartesian products

theory *Product-Type*
imports *Inductive*
uses
  (*Tools/split-rule.ML*)
  (*Tools/inductive-set-package.ML*)
  (*Tools/inductive-realizer.ML*)
  (*Tools/datatype-realizer.ML*)
begin

10.1 *bool* is a datatype

rep-datatype *bool*
  distinct *True-not-False False-not-True*
  induction *bool-induct*

declare *case-split* [*cases type*: *bool*]
  — prefer plain propositional version

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

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

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

10.2 Unit

typedef *unit* = {*True*}
⟨*proof*⟩

definition

Unity :: *unit* (′())
where
() = *Abs-unit True*

lemma *unit-eq* [*noatp*]: *u* = ()
⟨*proof*⟩

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

⟨*ML*⟩

lemma *unit-induct* [*noatp*,*induct type*: *unit*]: *P* () ==> *P x*
⟨*proof*⟩

rep-datatype *unit*
induction *unit-induct*

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*,*noatp*]: (%*u*::*unit*. *f* ()) = *f*
⟨*proof*⟩

code generator setup

instance *unit* :: *eq* ⟨*proof*⟩

lemma [*code func*]:
(*u*::*unit*) = *v* ⟷ *True* ⟨*proof*⟩

code-type *unit*
(*SML unit*)
(*OCaml unit*)
(*Haskell* ())

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

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

code-const *Unity*
(*SML* ())
(*OCaml* ())

(*Haskell* ())

code-reserved *SML*
unit

code-reserved *OCaml*
unit

10.3 Pairs

10.3.1 Product type, basic operations and concrete syntax

definition
Pair-Rep :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where
Pair-Rep a b = $(\lambda x\ y.\ x = a \wedge y = b)$

global

typedef (*Prod*)
$('a, 'b) *$ (**infixr** $*$ *20*)
$= \{f.\ \exists\ a\ b.\ f =$ *Pair-Rep* $(a::'a)\ (b::'b)\}$
⟨*proof*⟩

syntax (*xsymbols*)
$*$ :: [*type*, *type*] => *type* ((- ×/ -) [*21*, *20*] *20*)
syntax (*HTML* **output**)
$*$ :: [*type*, *type*] => *type* ((- ×/ -) [*21*, *20*] *20*)

consts
Pair :: $'a \Rightarrow 'b \Rightarrow 'a \times 'b$
fst :: $'a \times 'b \Rightarrow 'a$
snd :: $'a \times 'b \Rightarrow 'b$
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
curry :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

local

defs
Pair-def: *Pair a b* == *Abs-Prod* (*Pair-Rep a b*)
fst-def: *fst p* == *THE a. EX b. p = Pair a b*
snd-def: *snd p* == *THE b. EX a. p = Pair a b*
split-def: *split* == (%*c p. c* (*fst p*) (*snd p*))
curry-def: *curry* == (%*c x y. c* (*Pair x y*))

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminals
tuple-args patterns

syntax

-tuple :: *'a => tuple-args => 'a * 'b* ((*1'(-,/ -')*))
-tuple-arg :: *'a => tuple-args* (-)
-tuple-args :: *'a => tuple-args => tuple-args* (-,/ -)
-pattern :: *[pttrn, patterns] => pttrn* (*'(-,/ -')*)
:: *pttrn => patterns* (-)
-patterns :: *[pttrn, patterns] => patterns* (-,/ -)

translations
(*x*, *y*) == *Pair x y*
-tuple x (*-tuple-args y z*) == *-tuple x* (*-tuple-arg* (*-tuple y z*))
%(*x*,*y*,*zs*).*b* == *split*(%*x* (*y*,*zs*).*b*)
%(*x*,*y*).*b* == *split*(%*x y*. *b*)
-abs (*Pair x y*) *t* => %(*x*,*y*).*t*

⟨*ML*⟩

Towards a datatype declaration

lemma *surj-pair* [*simp*]: *EX x y*. *p* = (*x*, *y*)
⟨*proof*⟩

lemma *PairE* [*cases type*: *]:
obtains *x y* **where** *p* = (*x*, *y*)
⟨*proof*⟩

lemma *prod-induct* [*induct type*: *]: (⋀*a b*. *P* (*a*, *b*)) ⟹ *P x*
⟨*proof*⟩

lemma *ProdI*: *Pair-Rep a b* ∈ *Prod*
⟨*proof*⟩

lemma *Pair-Rep-inject*: *Pair-Rep a b* = *Pair-Rep a' b'* ⟹ *a* = *a'* ∧ *b* = *b'*
⟨*proof*⟩

lemma *inj-on-Abs-Prod*: *inj-on Abs-Prod Prod*
⟨*proof*⟩

lemma *Pair-inject*:
assumes (*a*, *b*) = (*a'*, *b'*)
and *a* = *a'* ==> *b* = *b'* ==> *R*
shows *R*
⟨*proof*⟩

lemma *Pair-eq* [*iff*]: ((*a*, *b*) = (*a'*, *b'*)) = (*a* = *a'* & *b* = *b'*)
⟨*proof*⟩

lemma *fst-conv* [*simp*, *code*]: *fst* (*a*, *b*) = *a*
⟨*proof*⟩

lemma *snd-conv* [*simp*, *code*]: *snd* (*a*, *b*) = *b*
⟨*proof*⟩

rep-datatype *prod*
inject *Pair-eq*
induction *prod-induct*

10.3.2 Basic rules and proof tools

lemma *fst-eqD*: *fst* (*x*, *y*) = *a* ==> *x* = *a*
⟨*proof*⟩

lemma *snd-eqD*: *snd* (*x*, *y*) = *a* ==> *y* = *a*
⟨*proof*⟩

lemma *pair-collapse* [*simp*]: (*fst p*, *snd p*) = *p*
⟨*proof*⟩

lemmas *surjective-pairing* = *pair-collapse* [*symmetric*]

lemma *split-paired-all*: (!!*x*. *PROP P x*) == (!!*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 *Pair-fst-snd-eq*: *s* = *t* ⟷ *fst s* = *fst t* ∧ *snd s* = *snd t*
⟨*proof*⟩

lemma *prod-eqI* [*intro?*]: *fst p* = *fst q* ⟹ *snd p* = *snd q* ⟹ *p* = *q*
⟨*proof*⟩

10.3.3 *split* and *curry*

lemma *split-conv* [*simp*, *code func*]: *split f* (*a*, *b*) = *f a b*

⟨*proof*⟩

lemma *curry-conv* [*simp*, *code func*]: *curry f a b* = *f* (*a*, *b*)
⟨*proof*⟩

lemmas *split* = *split-conv* — for backwards compatibility

lemma *splitI*: *f a b* ⟹ *split f* (*a*, *b*)
⟨*proof*⟩

lemma *splitD*: *split f* (*a*, *b*) ⟹ *f a b*
⟨*proof*⟩

lemma *curryI* [*intro*!]: *f* (*a*, *b*) ⟹ *curry f a b*
⟨*proof*⟩

lemma *curryD* [*dest*!]: *curry f a b* ⟹ *f* (*a*, *b*)
⟨*proof*⟩

lemma *curryE*: *curry f a b* ⟹ (*f* (*a*, *b*) ⟹ *Q*) ⟹ *Q*
⟨*proof*⟩

lemma *curry-split* [*simp*]: *curry* (*split f*) = *f*
⟨*proof*⟩

lemma *split-curry* [*simp*]: *split* (*curry f*) = *f*
⟨*proof*⟩

lemma *split-Pair* [*simp*]: (λ(*x*, *y*). (*x*, *y*)) = *id*
⟨*proof*⟩

lemma *split-eta*: (λ(*x*, *y*). *f* (*x*, *y*)) = *f*
— Subsumes the old *split-Pair* when *f* is the identity function.
⟨*proof*⟩

lemma *split-comp*: *split* (*f* ∘ *g*) *x* = *f* (*g* (*fst x*)) (*snd x*)
⟨*proof*⟩

lemma *split-twice*: *split f* (*split g p*) = *split* (λ*x y*. *split f* (*g x y*)) *p*
⟨*proof*⟩

lemma *split-paired-The*: (*THE x*. *P x*) = (*THE* (*a*, *b*). *P* (*a*, *b*))
— Can't be added to simpset: loops!
⟨*proof*⟩

lemma *The-split*: *The* (*split P*) = (*THE xy*. *P* (*fst xy*) (*snd xy*))
⟨*proof*⟩

lemma *split-weak-cong*: *p* = *q* ⟹ *split c p* = *split c q*

— Prevents simplification of c: much faster
⟨*proof*⟩

lemma *cond-split-eta*: $(!!x\ y.\ f\ x\ y = g\ (x,\ y)) ==> (\%(x,\ y).\ f\ x\ y) = g$
⟨*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* [*noatp*]: $R(split\ c\ p) = (ALL\ x\ y.\ p = (x,\ y) --> R(c\ x\ y))$
— 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* [*noatp*]: $R\ (split\ c\ p) = (\sim(EX\ x\ y.\ p = (x,\ y)\ \&\ (\sim R\ (c\ x\ y))))$
⟨*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*,*noatp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
⟨*proof*⟩

lemma *split-eta-SetCompr2* [*simp*,*noatp*]: (%*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*.

definition
internal-split :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a* × *'b* ⇒ *'c*
where
internal-split == *split*

lemma *internal-split-conv*: *internal-split c* (*a*, *b*) = *c a b*

⟨*proof*⟩

hide *const internal-split*

⟨*ML*⟩

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*⟩

lemma *prod-case-unfold*: *prod-case* = (%*c p*. *c* (*fst p*) (*snd p*))
⟨*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*⟩

10.4 Further cases/induct rules for tuples

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*⟩

10.4.1 Derived operations

The composition-uncurry combinator.

notation *fcomp* (**infixl** *o>* *60*)

definition
scomp :: ('*a* ⇒ '*b* × '*c*) ⇒ ('*b* ⇒ '*c* ⇒ '*d*) ⇒ '*a* ⇒ '*d* (**infixl** *o→* *60*)
where
f o→ g = (λ*x*. *split g* (*f x*))

lemma *scomp-apply*: (*f o→ g*) *x* = *split g* (*f x*)
⟨*proof*⟩

lemma *Pair-scomp*: *Pair x o→ f* = *f x*
⟨*proof*⟩

lemma *scomp-Pair*: *x o→ Pair* = *x*
⟨*proof*⟩

lemma *scomp-scomp*: (*f o→ g*) *o→ h* = *f o→* (λ*x*. *g x o→ 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*⟩

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 func del*]: $prod\text{-}fun\ f\ g = (\lambda(x,\ y).\ (f\ x,\ g\ y))$

lemma *prod-fun* [*simp*, *code func*]: $prod\text{-}fun\ f\ g\ (a,\ b) = (f\ a,\ g\ b)$
⟨*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*]: $(a,\ b) : r \Longrightarrow (f\ a,\ g\ b) : prod\text{-}fun\ f\ g\ `\ r$
⟨*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 :: $('a \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'b$
where
[*code func del*]: *apfst f* = *prod-fun f id*

definition
apsnd :: $('b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'c$
where
[*code func del*]: *apsnd f* = *prod-fun id f*

lemma *apfst-conv* [*simp*, *code*]:
$apfst\ f\ (x,\ y) = (f\ x,\ y)$
⟨*proof*⟩

lemma *upd-snd-conv* [*simp*, *code*]:

apsnd f (x, y) = (x, f y)
⟨*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 == Product-Type.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*⟧

$\Longrightarrow$ (*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*,*noatp*]:
(*ALL* z: *Sigma* A B. $P\ z$) = (*ALL* x:A. *ALL* y: $B\ x$. P(x,y))
⟨*proof*⟩

lemma *split-paired-Bex-Sigma* [*simp*,*noatp*]:
(*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*⟩

10.4.2 Code generator setup

instance ∗ :: (*eq*, *eq*) *eq* ⟨*proof*⟩

lemma [*code func*]:
(*x1*::'*a*::*eq*, *y1*::'*b*::*eq*) = (*x2*, *y2*) ⟷ *x1* = *x2* ∧ *y1* = *y2* ⟨*proof*⟩

lemma *split-case-cert*:
assumes *CASE* ≡ *split f*

shows *CASE* $(a,\ b) \equiv f\ a\ b$
⟨*proof*⟩

⟨*ML*⟩

code-type $*$
(*SML* **infix** *2* $*$)
(*OCaml* **infix** *2* $*$)
(*Haskell* !((-),/ (-)))

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

code-const *op* = :: $'a$::*eq* × $'b$::*eq* ⇒ $'a$ × $'b$ ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-const *Pair*
(*SML* !((-),/ (-)))
(*OCaml* !((-),/ (-)))
(*Haskell* !((-),/ (-)))

code-const *fst* **and** *snd*
(*Haskell fst* **and** *snd*)

types-code
$*$ ((- */ -))
attach (*term-of*) ⟪
fun term-of-id-42 aF aT bF bT $(x,\ y)$ = *HOLogic.pair-const aT bT* $ *aF x* $ *bF y*;
⟫
attach (*test*) ⟪
fun gen-id-42 aG aT bG bT i =
let
val $(x,\ t)$ = *aG i*;
val $(y,\ u)$ = *bG i*
in $((x,\ y),$ *fn* () => *HOLogic.pair-const aT bT* $ *t* () $ *u* ()) *end*;
⟫

consts-code
Pair ((-,/ -))

⟨*ML*⟩

10.5 Legacy bindings

⟨*ML*⟩

10.6 Further inductive packages

⟨*ML*⟩

end

11 Record: Extensible records with structural subtyping

theory *Record*
imports *Product-Type*
uses (*Tools/record-package.ML*)
begin

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
⟨*proof*⟩

lemma *rec-UNIV-I*: $\bigwedge x.\ x{\in}UNIV \equiv True$
⟨*proof*⟩

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

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
⟨*proof*⟩

11.1 Concrete record syntax

nonterminals
ident field-type field-types field fields update updates
syntax
-constify :: *id* => *ident* (-)
-constify :: *longid* => *ident* (-)

-field-type :: [*ident*, *type*] => *field-type* ((*2*- ::/ -))
:: *field-type* => *field-types* (-)
-field-types :: [*field-type*, *field-types*] => *field-types* (-,/ -)
-record-type :: *field-types* => *type* ((*3*'(| - |')))
-record-type-scheme :: [*field-types*, *type*] => *type* ((*3*'(| -,/ (*2*... ::/ -) |')))

-field :: [*ident*, 'a] => *field* ((*2*- =/ -))
:: *field* => *fields* (-)
-fields :: [*field*, *fields*] => *fields* (-,/ -)
-record :: *fields* => 'a ((*3*'(| - |')))
-record-scheme :: [*fields*, 'a] => 'a ((*3*'(| -,/ (*2*... =/ -) |')))

-update-name :: *idt*
-update :: [*ident*, 'a] => *update* ((*2*- :=/ -))
:: *update* => *updates* (-)
-updates :: [*update*, *updates*] => *updates* (-,/ -)

-record-update :: [*'a*, *updates*] => *'b* (-/(*3'*(| - |')) [*900*,*0*] *900*)

syntax (*xsymbols*)
-record-type :: *field-types* => *type* ((*3*(⦇-⦈)))
-record-type-scheme :: [*field-types*, *type*] => *type* ((*3*(⦇-,/ (*2*... ::/ -)⦈)))
-record :: *fields* => *'a* ((*3*(⦇-⦈)))
-record-scheme :: [*fields*, *'a*] => *'a* ((*3*(⦇-,/ (*2*... =/ -)⦈)))
-record-update :: [*'a*, *updates*] => *'b* (-/(*3*(⦇-⦈)) [*900*,*0*] *900*)

⟨*ML*⟩

end

12 OrderedGroup: Ordered Groups

theory *OrderedGroup*
imports *Lattices*
uses ~~/*src*/*Provers*/*Arith*/*abel-cancel.ML*
begin

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

12.1 Semigroups and Monoids

class *semigroup-add* = *plus* +
assumes *add-assoc*: (*a* + *b*) + *c* = *a* + (*b* + *c*)

class *ab-semigroup-add* = *semigroup-add* +
assumes *add-commute*: *a* + *b* = *b* + *a*
begin

lemma *add-left-commute*: *a* + (*b* + *c*) = *b* + (*a* + *c*)
⟨*proof*⟩

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

end

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

class *semigroup-mult* = *times* +
  assumes *mult-assoc*: $(a * b) * c = a * (b * c)$

class *ab-semigroup-mult* = *semigroup-mult* +
  assumes *mult-commute*: $a * b = b * a$
begin

lemma *mult-left-commute*: $a * (b * c) = b * (a * c)$
  ⟨*proof*⟩

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

end

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

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

lemma *mult-left-idem*: $x * (x * y) = x * y$
  ⟨*proof*⟩

lemmas *mult-ac-idem* = *mult-ac mult-idem mult-left-idem*

end

lemmas *mult-ac-idem* = *mult-ac mult-idem mult-left-idem*

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

lemma *zero-reorient*: $0 = x \longleftrightarrow x = 0$
  ⟨*proof*⟩

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

subclass *monoid-add*
  ⟨*proof*⟩

end

class *monoid-mult* = *one* + *semigroup-mult* +
  assumes *mult-1-left* [*simp*]: *1* ∗ *a* = *a*
  assumes *mult-1-right* [*simp*]: *a* ∗ *1* = *a*

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

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

subclass *monoid-mult*
  ⟨*proof*⟩

end

class *cancel-semigroup-add* = *semigroup-add* +
  assumes *add-left-imp-eq*: *a* + *b* = *a* + *c* ⟹ *b* = *c*
  assumes *add-right-imp-eq*: *b* + *a* = *c* + *a* ⟹ *b* = *c*

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

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

end

context *cancel-ab-semigroup-add*
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

12.2 Groups

class *group-add* = *minus* + *uminus* + *monoid-add* +
  assumes *left-minus* [*simp*]: − *a* + *a* = *0*
  assumes *diff-minus*: *a* − *b* = *a* + (− *b*)
begin

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

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

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

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

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

lemma *equals-zero-I*:
assumes $a + b = 0$
shows $- a = b$
⟨*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*⟩

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-ab-semigroup-add*
⟨*proof*⟩

lemma *uminus-add-conv-diff*:
$- a + b = b - a$
⟨*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*: $a + (b - c) = (a + b) - c$
⟨*proof*⟩

lemma *diff-add-eq*: $(a - b) + c = (a + c) - b$
⟨*proof*⟩

lemma *diff-eq-eq*: $a - b = c \longleftrightarrow a = c + b$
⟨*proof*⟩

lemma *eq-diff-eq*: $a = c - b \longleftrightarrow a + b = c$
⟨*proof*⟩

lemma *diff-diff-eq*: $(a - b) - c = a - (b + c)$
⟨*proof*⟩

lemma *diff-diff-eq2*: $a - (b - c) = (a + c) - b$
⟨*proof*⟩

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

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

lemmas *compare-rls* =
diff-minus [*symmetric*]
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq

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

end

12.3 (Partially) Ordered Groups

class *pordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
assumes *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
begin

lemma *add-right-mono*:
$a \leq b \Longrightarrow a + c \leq b + c$
⟨*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 *pordered-cancel-ab-semigroup-add* =
pordered-ab-semigroup-add + *cancel-ab-semigroup-add*
begin

lemma *add-strict-left-mono*:
$a < b \Longrightarrow c + a < c + b$
⟨*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 *pordered-ab-semigroup-add-imp-le* =
pordered-cancel-ab-semigroup-add +
assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$
shows $a < b$
⟨*proof*⟩

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

12.4 Support for reasoning about signs

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

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

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*:
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*⟩

end

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

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

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

subclass *pordered-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*: $a - b < c \longleftrightarrow a < c + b$
⟨*proof*⟩

lemma *less-diff-eq*: $a < c - b \longleftrightarrow a + b < c$
⟨*proof*⟩

lemma *diff-le-eq*: $a - b \leq c \longleftrightarrow a \leq c + b$
⟨*proof*⟩

lemma *le-diff-eq*: $a \leq c - b \longleftrightarrow a + b \leq c$
⟨*proof*⟩

lemmas *compare-rls* =
diff-minus [*symmetric*]
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-less-eq less-diff-eq diff-le-eq le-diff-eq
diff-eq-eq eq-diff-eq

This list of rewrites simplifies (in)equalities by bringing subtractions to the top and then moving negative terms to the other side. Use with *add-ac*

lemmas (**in** −) *compare-rls* =
diff-minus [*symmetric*]
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-less-eq less-diff-eq diff-le-eq le-diff-eq
diff-eq-eq eq-diff-eq

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

lemmas *group-simps* =
add-ac
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq diff-minus [*symmetric*] *uminus-add-conv-diff*
diff-less-eq less-diff-eq diff-le-eq le-diff-eq

end

lemmas *group-simps* =
mult-ac
add-ac
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq diff-minus [*symmetric*] *uminus-add-conv-diff*
diff-less-eq less-diff-eq diff-le-eq le-diff-eq

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

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

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

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

end

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

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

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

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

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

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

end

— FIXME localize the following

lemma *add-increasing*:
fixes $c :: 'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, comm\text{-}monoid\text{-}add\}$
shows $[|0 \leq a; b \leq c|] ==> b \leq a + c$
⟨*proof*⟩

lemma *add-increasing2*:
fixes $c :: 'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, comm\text{-}monoid\text{-}add\}$
shows $[|0 \leq c; b \leq a|] ==> b \leq a + c$
⟨*proof*⟩

lemma *add-strict-increasing*:
fixes $c :: 'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, comm\text{-}monoid\text{-}add\}$
shows $[|0 < a; b \leq c|] ==> b < a + c$
⟨*proof*⟩

lemma *add-strict-increasing2*:
fixes $c :: 'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, comm\text{-}monoid\text{-}add\}$
shows $[|0 \leq a; b < c|] ==> b < a + c$
⟨*proof*⟩

class *pordered-ab-group-add-abs* = *pordered-ab-group-add* + *abs* +
assumes *abs-ge-zero* [*simp*]: $|a| \geq 0$
and *abs-ge-self*: $a \leq |a|$
and *abs-leI*: $a \leq b \Longrightarrow -a \leq b \Longrightarrow |a| \leq b$
and *abs-minus-cancel* [*simp*]: $|-a| = |a|$
and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$
begin

lemma *abs-minus-le-zero*: $-|a| \leq 0$
⟨*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*, *noatp*]: $0 = |a| \longleftrightarrow a = 0$

⟨*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-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

12.5 Lattice Ordered (Abelian) Groups

class *lordered-ab-group-add-meet* = *pordered-ab-group-add* + *lower-semilattice*
begin

lemma *add-inf-distrib-left*:
  $a + inf\ b\ c = inf\ (a + b)\ (a + c)$
⟨*proof*⟩

lemma *add-inf-distrib-right*:
  $inf\ a\ b + c = inf\ (a + c)\ (b + c)$
⟨*proof*⟩

end

class *lordered-ab-group-add-join* = *pordered-ab-group-add* + *upper-semilattice*
begin

lemma *add-sup-distrib-left*:
  $a + sup\ b\ c = sup\ (a + b)\ (a + c)$
⟨*proof*⟩

lemma *add-sup-distrib-right*:
  $sup\ a\ b + c = sup\ (a+c)\ (b+c)$
⟨*proof*⟩

end

class *lordered-ab-group-add* = *pordered-ab-group-add* + *lattice*
begin

subclass *lordered-ab-group-add-meet* ⟨*proof*⟩
subclass *lordered-ab-group-add-join* ⟨*proof*⟩

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

lemma *inf-eq-neg-sup*: $inf\ a\ b = -\ sup\ (-a)\ (-b)$
⟨*proof*⟩

lemma *sup-eq-neg-inf*: $sup\ a\ b = -\ inf\ (-a)\ (-b)$
⟨*proof*⟩

lemma *neg-inf-eq-sup*: $- \ inf\ a\ b = sup\ (-a)\ (-b)$
⟨*proof*⟩

lemma *neg-sup-eq-inf*: $- \ sup\ a\ b = inf\ (-a)\ (-b)$
⟨*proof*⟩

lemma *add-eq-inf-sup*: $a + b = sup\ a\ b + inf\ a\ b$
⟨*proof*⟩

12.6 Positive Part, Negative Part, Absolute Value

definition
$nprt :: 'a \Rightarrow 'a$ **where**
$nprt\ x = inf\ x\ 0$

definition
$pprt :: 'a \Rightarrow 'a$ **where**
$pprt\ x = sup\ x\ 0$

lemma *pprt-neg*: $pprt\ (- \ x) = - \ nprt\ x$
⟨*proof*⟩

lemma *nprt-neg*: $nprt\ (- \ x) = - \ pprt\ x$
⟨*proof*⟩

lemma *prts*: $a = pprt\ a + nprt\ a$
⟨*proof*⟩

lemma *zero-le-pprt*[*simp*]: $0 \leq pprt\ a$
⟨*proof*⟩

lemma *nprt-le-zero*[*simp*]: $nprt\ a \leq 0$
⟨*proof*⟩

lemma *le-eq-neg*: $a \leq - \ b \longleftrightarrow a + b \leq 0$ (**is** $?l = ?r$)
⟨*proof*⟩

lemma *pprt-0*[*simp*]: $pprt\ 0 = 0$ ⟨*proof*⟩
lemma *nprt-0*[*simp*]: $nprt\ 0 = 0$ ⟨*proof*⟩

lemma *pprt-eq-id* [*simp*, *noatp*]: $0 \leq x \Longrightarrow pprt\ x = x$
⟨*proof*⟩

lemma *nprt-eq-id* [*simp*, *noatp*]: $x \leq 0 \Longrightarrow nprt\ x = x$
⟨*proof*⟩

lemma *pprt-eq-0* [*simp*, *noatp*]: $x \leq 0 \Longrightarrow pprt\ x = 0$
⟨*proof*⟩

lemma *nprt-eq-0* [*simp*, *noatp*]: $0 \leq x \Longrightarrow nprt\ x = 0$
⟨*proof*⟩

lemma *sup-0-imp-0*: $sup\ a\ (-\ a) = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inf-0-imp-0*: $inf\ a\ (-a) = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inf-0-eq-0* [*simp*, *noatp*]: $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *sup-0-eq-0* [*simp*, *noatp*]: $sup\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

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

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

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

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

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

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

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 *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
⟨*proof*⟩

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
⟨*proof*⟩

lemma *pprt-mono* [*simp*, *noatp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
⟨*proof*⟩

lemma *nprt-mono* [*simp*, *noatp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
⟨*proof*⟩

end

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

class *lordered-ab-group-add-abs* = *lordered-ab-group-add* + *abs* +
assumes *abs-lattice*: $|a| = sup\ a\ (-\ a)$
begin

lemma *abs-prts*: $|a| = pprt\ a - nprt\ a$
⟨*proof*⟩

subclass *pordered-ab-group-add-abs*
⟨*proof*⟩

end

lemma *sup-eq-if*:
fixes $a :: 'a::\{lordered\text{-}ab\text{-}group\text{-}add, linorder\}$
shows $sup\ a\ (-\ a) = (if\ a < 0\ then\ -\ a\ else\ a)$
⟨*proof*⟩

lemma *abs-if-lattice*:
fixes $a :: 'a::\{lordered\text{-}ab\text{-}group\text{-}add\text{-}abs, linorder\}$
shows $|a| = (if\ a < 0\ then\ -\ a\ else\ a)$
⟨*proof*⟩

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*: $(x::'a::pordered\text{-}ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x < y) = (x' < y')$

⟨*proof*⟩

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

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

lemma *estimate-by-abs*:
$a + b <= (c::'a::lordered\text{-}ab\text{-}group\text{-}add\text{-}abs) \Longrightarrow a <= c + abs\ b$
⟨*proof*⟩

12.7 Tools setup

lemma *add-mono-thms-ordered-semiring* [*noatp*]:
fixes $i\ j\ k :: 'a::pordered\text{-}ab\text{-}semigroup\text{-}add$
shows $i \leq j \wedge k \leq l \Longrightarrow i + k \leq j + l$
and $i = j \wedge k \leq l \Longrightarrow i + k \leq j + l$
and $i \leq j \wedge k = l \Longrightarrow i + k \leq j + l$
and $i = j \wedge k = l \Longrightarrow i + k = j + l$
⟨*proof*⟩

lemma *add-mono-thms-ordered-field* [*noatp*]:
fixes $i\ j\ k :: 'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add$
shows $i < j \wedge k = l \Longrightarrow i + k < j + l$
and $i = j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k \leq l \Longrightarrow i + k < j + l$
and $i \leq j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k < l \Longrightarrow i + k < j + l$
⟨*proof*⟩

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

lemmas *diff-less-0-iff-less* [*simp*] = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-eq-0-iff-eq* [*simp*, *noatp*] = *eq-iff-diff-eq-0* [*symmetric*]
lemmas *diff-le-0-iff-le* [*simp*] = *le-iff-diff-le-0* [*symmetric*]

⟨*ML*⟩

end

13 Ring-and-Field: (Ordered) Rings and Fields

theory *Ring-and-Field*
imports *OrderedGroup*
begin

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
  assumes *left-distrib*: $(a + b) * c = a * c + b * c$
  assumes *right-distrib*: $a * (b + c) = a * b + a * c$
begin

For the *combine-numerals* simproc

lemma *combine-common-factor*:
  $a * e + (b * e + c) = (a + b) * e + c$
  ⟨*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* + *comm-monoid-add* + *cancel-ab-semigroup-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* + *comm-monoid-add* + *cancel-ab-semigroup-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

end

class *zero-neq-one* = *zero* + *one* +
  assumes *zero-neq-one* [*simp*]: $0 \neq 1$
begin

lemma *one-neq-zero* [*simp*]: $1 \neq 0$
  ⟨*proof*⟩

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*

begin

subclass *semiring-1* ⟨*proof*⟩

end

class *no-zero-divisors* = *zero* + *times* +
  assumes *no-zero-divisors*: $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$

class *semiring-1-cancel* = *semiring* + *comm-monoid-add* + *zero-neq-one*
  + *cancel-ab-semigroup-add* + *monoid-mult*

begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *semiring-1* ⟨*proof*⟩

end

class *comm-semiring-1-cancel* = *comm-semiring* + *comm-monoid-add* + *comm-monoid-mult* + *zero-neq-one* + *cancel-ab-semigroup-add*
begin

subclass *semiring-1-cancel* ⟨*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*⟩

lemma *minus-mult-minus* [*simp*]: $-a * -b = a * b$
⟨*proof*⟩

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

lemma *right-diff-distrib*: $a * (b - c) = a * b - a * c$
⟨*proof*⟩

lemma *left-diff-distrib*: $(a - b) * c = a * c - b * c$
⟨*proof*⟩

lemmas *ring-distribs* =
right-distrib left-distrib left-diff-distrib right-diff-distrib

lemmas *ring-simps* =
add-ac
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq diff-minus [*symmetric*] *uminus-add-conv-diff*

ring-distribs

lemma *eq-add-iff1*:
$a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
⟨*proof*⟩

lemma *eq-add-iff2*:
$a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$
⟨*proof*⟩

end

lemmas *ring-distribs* =
right-distrib left-distrib left-diff-distrib right-diff-distrib

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* ⟨*proof*⟩
subclass *comm-semiring-0* ⟨*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*⟩

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*, *noatp*]:
$a * c = b * c \longleftrightarrow c = 0 \vee a = b$

⟨*proof*⟩

lemma *mult-cancel-left* [*simp*, *noatp*]:
  $c * a = c * b \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *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*⟩

end

class *division-ring* = *ring-1* + *inverse* +
  assumes *left-inverse* [*simp*]: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
  assumes *right-inverse* [*simp*]: $a \neq 0 \Longrightarrow a * inverse\ a = 1$
begin

subclass *ring-1-no-zero-divisors*
⟨*proof*⟩

lemma *nonzero-imp-inverse-nonzero*:
  $a \neq 0 \Longrightarrow inverse\ a \neq 0$
⟨*proof*⟩

lemma *inverse-zero-imp-zero*:

inverse a = 0 ⟹ a = 0
⟨*proof*⟩

lemma *nonzero-inverse-minus-eq*:
assumes *a ≠ 0*
shows *inverse (− a) = − inverse a*
⟨*proof*⟩

lemma *nonzero-inverse-inverse-eq*:
assumes *a ≠ 0*
shows *inverse (inverse a) = a*
⟨*proof*⟩

lemma *nonzero-inverse-eq-imp-eq*:
assumes *inveq*: *inverse a = inverse b*
and *anz*: *a ≠ 0*
and *bnz*: *b ≠ 0*
shows *a = b*
⟨*proof*⟩

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

lemma *inverse-unique*:
assumes *ab*: *a * b = 1*
shows *inverse a = b*
⟨*proof*⟩

lemma *nonzero-inverse-mult-distrib*:
assumes *anz*: *a ≠ 0*
and *bnz*: *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*⟩

end

class *field = comm-ring-1 + inverse +*
assumes *field-inverse*: *a ≠ 0 ⟹ inverse a * a = 1*
assumes *divide-inverse*: *a / b = a * inverse b*
begin

subclass *division-ring*
⟨*proof*⟩

subclass *idom* ⟨*proof*⟩

lemma *right-inverse-eq*: $b \neq 0 \Longrightarrow a \mathbin{/} b = 1 \longleftrightarrow a = b$
⟨*proof*⟩

lemma *nonzero-inverse-eq-divide*: $a \neq 0 \Longrightarrow inverse\ a = 1 \mathbin{/} a$
⟨*proof*⟩

lemma *divide-self* [*simp*]: $a \neq 0 \Longrightarrow a \mathbin{/} a = 1$
⟨*proof*⟩

lemma *divide-zero-left* [*simp*]: $0 \mathbin{/} a = 0$
⟨*proof*⟩

lemma *inverse-eq-divide*: $inverse\ a = 1 \mathbin{/} a$
⟨*proof*⟩

lemma *add-divide-distrib*: $(a+b) \mathbin{/} c = a/c + b/c$
⟨*proof*⟩

end

class *division-by-zero* = *zero* + *inverse* +
assumes *inverse-zero* [*simp*]: *inverse 0* = *0*

lemma *divide-zero* [*simp*]:
$a \mathbin{/} 0 = (0::'a::\{field, division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *divide-self-if* [*simp*]:
$a \mathbin{/} (a::'a::\{field, division\text{-}by\text{-}zero\}) = (if\ a=0\ then\ 0\ else\ 1)$
⟨*proof*⟩

class *mult-mono* = *times* + *zero* + *ord* +
assumes *mult-left-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
assumes *mult-right-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * c$

class *pordered-semiring* = *mult-mono* + *semiring-0* + *pordered-ab-semigroup-add*

begin

lemma *mult-mono*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq b \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
⟨*proof*⟩

lemma *mult-mono'*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq a \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
⟨*proof*⟩

end

class *pordered-cancel-semiring* = *mult-mono* + *pordered-ab-semigroup-add*
+ *semiring* + *comm-monoid-add* + *cancel-ab-semigroup-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩
subclass *pordered-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-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 *ordered-semiring* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add*
+ *mult-mono*
begin

subclass *pordered-cancel-semiring* ⟨*proof*⟩

subclass *pordered-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 *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$

assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *ordered-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-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 *mult-mono1* = *times* + *zero* + *ord* +
assumes *mult-mono1*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$

class *pordered-comm-semiring* = *comm-semiring-0*
+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-semiring*
⟨*proof*⟩

end

class *pordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-comm-semiring* ⟨*proof*⟩
subclass *pordered-cancel-semiring* ⟨*proof*⟩

end

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-left-mono-comm*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *ordered-semiring-strict*
⟨*proof*⟩

subclass *pordered-cancel-comm-semiring*
⟨*proof*⟩

end

class *pordered-ring* = *ring* + *pordered-cancel-semiring*
begin

subclass *pordered-ab-group-add* ⟨*proof*⟩

lemmas *ring-simps* = *ring-simps group-simps*

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 *abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +
assumes *abs-if*: $|a| = (\mathit{if}\ a < 0\ \mathit{then} - a\ \mathit{else}\ a)$

class *sgn-if* = *minus* + *uminus* + *zero* + *one* + *ord* + *sgn* +
  assumes *sgn-if*: *sgn x* = (*if x* = *0 then 0 else if 0* < *x then 1 else* − *1*)

lemma (**in** *sgn-if*) *sgn0*[*simp*]: *sgn 0* = *0*
⟨*proof*⟩

class *ordered-ring* = *ring* + *ordered-semiring*
  + *ordered-ab-group-add* + *abs-if*
begin

subclass *pordered-ring* ⟨*proof*⟩

subclass *pordered-ab-group-add-abs*
⟨*proof*⟩

end

class *ordered-ring-strict* = *ring* + *ordered-semiring-strict*
  + *ordered-ab-group-add* + *abs-if*
begin

subclass *ordered-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*⟩

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
⟨*proof*⟩

lemma *not-square-less-zero* [*simp*]: $\neg (a * a < 0)$
⟨*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*⟩

end

This list of rewrites simplifies ring terms by multiplying everything out and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides ring equalities but also helps with inequalities.

lemmas *ring-simps* = *group-simps ring-distribs*

class *pordered-comm-ring* = *comm-ring* + *pordered-comm-semiring*
begin

subclass *pordered-ring* ⟨*proof*⟩
subclass *pordered-cancel-comm-semiring* ⟨*proof*⟩

end

class *ordered-semidom* = *comm-semiring-1-cancel* + *ordered-comm-semiring-strict*
+

assumes *zero-less-one* [*simp*]: *0* < *1*
begin

lemma *pos-add-strict*:
shows *0* < *a* ⟹ *b* < *c* ⟹ *b* < *a* + *c*
⟨*proof*⟩

lemma *zero-le-one* [*simp*]: *0* ≤ *1*
⟨*proof*⟩

lemma *not-one-le-zero* [*simp*]: ¬ *1* ≤ *0*
⟨*proof*⟩

lemma *not-one-less-zero* [*simp*]: ¬ *1* < *0*
⟨*proof*⟩

lemma *less-1-mult*:
assumes *1* < *m* **and** *1* < *n*
shows *1* < *m* ∗ *n*
⟨*proof*⟩

end

class *ordered-idom* = *comm-ring-1* +
ordered-comm-semiring-strict + *ordered-ab-group-add* +
abs-if + *sgn-if*

begin

subclass *ordered-ring-strict* ⟨*proof*⟩
subclass *pordered-comm-ring* ⟨*proof*⟩
subclass *idom* ⟨*proof*⟩

subclass *ordered-semidom*
⟨*proof*⟩

lemma *linorder-neqE-ordered-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*⟩

end

class *ordered-field* = *field* + *ordered-idom*

Simprules for comparisons where common factors can be cancelled.

lemmas *mult-compare-simps* =
mult-le-cancel-right *mult-le-cancel-left*
mult-le-cancel-right1 *mult-le-cancel-right2*
mult-le-cancel-left1 *mult-le-cancel-left2*
mult-less-cancel-right *mult-less-cancel-left*

mult-less-cancel-right1 mult-less-cancel-right2
mult-less-cancel-left1 mult-less-cancel-left2
mult-cancel-right mult-cancel-left
mult-cancel-right1 mult-cancel-right2
mult-cancel-left1 mult-cancel-left2

— FIXME continue localization here

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
(*inverse a* = *0*) = (*a* = (*0*::'*a*::{*division-ring*,*division-by-zero*}))
⟨*proof*⟩

lemma *inverse-minus-eq* [*simp*]:
inverse(−*a*) = −*inverse*(*a*::'*a*::{*division-ring*,*division-by-zero*})
⟨*proof*⟩

lemma *inverse-eq-imp-eq*:
inverse a = *inverse b* ==> *a* = (*b*::'*a*::{*division-ring*,*division-by-zero*})
⟨*proof*⟩

lemma *inverse-eq-iff-eq* [*simp*]:
(*inverse a* = *inverse b*) = (*a* = (*b*::'*a*::{*division-ring*,*division-by-zero*}))
⟨*proof*⟩

lemma *inverse-inverse-eq* [*simp*]:
inverse(*inverse* (*a*::'*a*::{*division-ring*,*division-by-zero*})) = *a*
⟨*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*::'*a*::{*field*,*division-by-zero*})
⟨*proof*⟩

There is no slick version using division by zero.

lemma *inverse-add*:
[|*a* ≠ *0*; *b* ≠ *0*|]
==> *inverse a* + *inverse b* = (*a*+*b*) ∗ *inverse a* ∗ *inverse* (*b*::'*a*::*field*)
⟨*proof*⟩

lemma *inverse-divide* [*simp*]:
inverse (*a*/*b*) = *b* / (*a*::'*a*::{*field*,*division-by-zero*})
⟨*proof*⟩

13.1 Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemma *nonzero-mult-divide-mult-cancel-left*[*simp*,*noatp*]:
assumes [*simp*]: $b \neq 0$ **and** [*simp*]: $c \neq 0$ **shows** $(c*a)/(c*b) = a/(b::'a::field)$
⟨*proof*⟩

lemma *mult-divide-mult-cancel-left*:
$c \neq 0 \Longrightarrow (c*a) \,/\, (c*b) = a \,/\, (b::'a::\{field, division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-right* [*noatp*]:
$[|b \neq 0;\ c \neq 0|] \Longrightarrow (a*c) \,/\, (b*c) = a/(b::'a::field)$
⟨*proof*⟩

lemma *mult-divide-mult-cancel-right*:
$c \neq 0 \Longrightarrow (a*c) \,/\, (b*c) = a \,/\, (b::'a::\{field, division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *divide-1* [*simp*]: $a/1 = (a::'a::field)$
⟨*proof*⟩

lemma *times-divide-eq-right*: $a * (b/c) = (a*b) \,/\, (c::'a::field)$
⟨*proof*⟩

lemma *times-divide-eq-left*: $(b/c) * a = (b*a) \,/\, (c::'a::field)$
⟨*proof*⟩

lemmas *times-divide-eq* = *times-divide-eq-right times-divide-eq-left*

lemma *divide-divide-eq-right* [*simp*,*noatp*]:
$a \,/\, (b/c) = (a*c) \,/\, (b::'a::\{field, division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *divide-divide-eq-left* [*simp*,*noatp*]:
$(a \,/\, b) \,/\, (c::'a::\{field, division\text{-}by\text{-}zero\}) = a \,/\, (b*c)$
⟨*proof*⟩

lemma *add-frac-eq*: $(y::'a::field) \neq 0 \Longrightarrow z \neq 0 \Longrightarrow$
$x \,/\, y + w \,/\, z = (x * z + w * y) \,/\, (y * z)$
⟨*proof*⟩

13.1.1 Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if*[*simp*,*noatp*]:
fixes $c :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $(c*a) \,/\, (c*b) = (if\ c=0\ then\ 0\ else\ a/b)$
⟨*proof*⟩

lemma *nonzero-mult-divide-cancel-right*[*simp*,*noatp*]:
$b \neq 0 \Longrightarrow a * b \,/\, b = (a::'a::field)$
⟨*proof*⟩

lemma *nonzero-mult-divide-cancel-left*[*simp*,*noatp*]:
$a \neq 0 \Longrightarrow a * b \mathbin{/} a = (b::'a::field)$
⟨*proof*⟩

lemma *nonzero-divide-mult-cancel-right*[*simp*,*noatp*]:
$[\![a \neq 0;\ b \neq 0]\!] \Longrightarrow b \mathbin{/} (a * b) = 1/(a::'a::field)$
⟨*proof*⟩

lemma *nonzero-divide-mult-cancel-left*[*simp*,*noatp*]:
$[\![a \neq 0;\ b \neq 0]\!] \Longrightarrow a \mathbin{/} (a * b) = 1/(b::'a::field)$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-left2*[*simp*,*noatp*]:
$[|b \neq 0;\ c \neq 0|] ==> (c*a) \mathbin{/} (b*c) = a/(b::'a::field)$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-right2*[*simp*,*noatp*]:
$[|b \neq 0;\ c \neq 0|] ==> (a*c) \mathbin{/} (c*b) = a/(b::'a::field)$
⟨*proof*⟩

13.2 Division and Unary Minus

lemma *nonzero-minus-divide-left*: $b \neq 0 ==> -(a/b) = (-a) \mathbin{/} (b::'a::field)$
⟨*proof*⟩

lemma *nonzero-minus-divide-right*: $b \neq 0 ==> -(a/b) = a \mathbin{/} -(b::'a::field)$
⟨*proof*⟩

lemma *nonzero-minus-divide-divide*: $b \neq 0 ==> (-a)/(-b) = a \mathbin{/} (b::'a::field)$
⟨*proof*⟩

lemma *minus-divide-left*: $-(a/b) = (-a) \mathbin{/} (b::'a::field)$
⟨*proof*⟩

lemma *minus-divide-right*: $-(a/b) = a \mathbin{/} -(b::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

The effect is to extract signs from divisions

lemmas *divide-minus-left* = *minus-divide-left* [*symmetric*]
lemmas *divide-minus-right* = *minus-divide-right* [*symmetric*]
declare *divide-minus-left* [*simp*] *divide-minus-right* [*simp*]

Also, extract signs from products

lemmas *mult-minus-left* = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* = *minus-mult-right* [*symmetric*]
declare *mult-minus-left* [*simp*] *mult-minus-right* [*simp*]

lemma *minus-divide-divide* [*simp*]:
$(-a)/(-b) = a\ /\ (b::'a::\{field, division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *diff-divide-distrib*: $(a-b)/(c::'a::field) = a/c - b/c$
⟨*proof*⟩

lemma *add-divide-eq-iff*:
$(z::'a::field) \neq 0 \Longrightarrow x + y/z = (z*x + y)/z$
⟨*proof*⟩

lemma *divide-add-eq-iff*:
$(z::'a::field) \neq 0 \Longrightarrow x/z + y = (x + z*y)/z$
⟨*proof*⟩

lemma *diff-divide-eq-iff*:
$(z::'a::field) \neq 0 \Longrightarrow x - y/z = (z*x - y)/z$
⟨*proof*⟩

lemma *divide-diff-eq-iff*:
$(z::'a::field) \neq 0 \Longrightarrow x/z - y = (x - z*y)/z$
⟨*proof*⟩

lemma *nonzero-eq-divide-eq*: $c\neq 0 ==> ((a::'a::field) = b/c) = (a*c = b)$
⟨*proof*⟩

lemma *nonzero-divide-eq-eq*: $c\neq 0 ==> (b/c = (a::'a::field)) = (b = a*c)$
⟨*proof*⟩

lemma *eq-divide-eq*:
$((a::'a::\{field, division\text{-}by\text{-}zero\}) = b/c) = (if\ c\neq 0\ then\ a*c = b\ else\ a=0)$
⟨*proof*⟩

lemma *divide-eq-eq*:
$(b/c = (a::'a::\{field, division\text{-}by\text{-}zero\})) = (if\ c\neq 0\ then\ b = a*c\ else\ a=0)$
⟨*proof*⟩

lemma *divide-eq-imp*: $(c::'a::\{division\text{-}by\text{-}zero, field\}) \sim= 0 ==>$
$b = a * c ==> b\ /\ c = a$
⟨*proof*⟩

lemma *eq-divide-imp*: $(c::'a::\{division\text{-}by\text{-}zero, field\}) \sim= 0 ==>$
$a * c = b ==> a = b\ /\ c$
⟨*proof*⟩

lemmas *field-eq-simps* = *ring-simps*

add-divide-eq-iff divide-add-eq-iff
diff-divide-eq-iff divide-diff-eq-iff

nonzero-eq-divide-eq nonzero-divide-eq-eq

An example:

lemma fixes *a b c d e f* :: *'a::field*
shows $[\![a{\neq}b;\ c{\neq}d;\ e{\neq}f\ ]\!] \Longrightarrow ((a-b)*(c-d)*(e-f))/((c-d)*(e-f)*(a-b)) = 1$
⟨*proof*⟩

lemma *diff-frac-eq*: $(y::'a::field) \sim= 0 \Longrightarrow z \sim= 0 \Longrightarrow$
$x / y - w / z = (x * z - w * y) / (y * z)$
⟨*proof*⟩

lemma *frac-eq-eq*: $(y::'a::field) \sim= 0 \Longrightarrow z \sim= 0 \Longrightarrow$
$(x / y = w / z) = (x * z = w * y)$
⟨*proof*⟩

13.3 Ordered Fields

lemma *positive-imp-inverse-positive*:
assumes *a-gt-0*: $0 < a$ **shows** $0 <$ *inverse* $(a::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *negative-imp-inverse-negative*:
$a < 0 \Longrightarrow$ *inverse* $a < (0::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-le-imp-le*:
assumes *invle*: *inverse* $a \leq$ *inverse* b **and** *apos*: $0 < a$
shows $b \leq (a::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 <$ *inverse* a **and** *nz*: $a \neq 0$
shows $0 < (a::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 <$ *inverse* $a) = (0 < (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *inverse-negative-imp-negative*:
assumes *inv-less-0*: *inverse* $a < 0$ **and** *nz*: $a \neq 0$
shows $a < (0::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-negative-iff-negative* [*simp*]:

$(inverse\ a < 0) = (a < (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
$(0 \leq inverse\ a) = (0 \leq (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
$(inverse\ a \leq 0) = (a \leq (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *ordered-field-no-lb*: $\forall\ x.\ \exists\, y.\ y < (x::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *ordered-field-no-ub*: $\forall\ x.\ \exists\, y.\ y > (x::'a::ordered\text{-}field)$
⟨*proof*⟩

13.4 Anti-Monotonicity of *inverse*

lemma *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows $inverse\ b < inverse\ (a::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-less-imp-less*:
$[|inverse\ a < inverse\ b;\ 0 < a|] ==> b < (a::'a::ordered\text{-}field)$
⟨*proof*⟩

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*,*noatp*]:
$[|0 < a;\ 0 < b|] ==> (inverse\ a < inverse\ b) = (b < (a::'a::ordered\text{-}field))$
⟨*proof*⟩

lemma *le-imp-inverse-le*:
$[|a \leq b;\ 0 < a|] ==> inverse\ b \leq inverse\ (a::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-le-iff-le* [*simp*,*noatp*]:
$[|0 < a;\ 0 < b|] ==> (inverse\ a \leq inverse\ b) = (b \leq (a::'a::ordered\text{-}field))$
⟨*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*:
$[|inverse\ a \leq inverse\ b;\ b < 0|] ==> b \leq (a::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *less-imp-inverse-less-neg*:
$[|a < b;\ b < 0|] ==> inverse\ b < inverse\ (a::'a::ordered\text{-}field)$

⟨*proof*⟩

lemma *inverse-less-imp-less-neg*:
[|*inverse a < inverse b*; *b < 0*|] ==> *b < (a::'a::ordered-field)*
⟨*proof*⟩

lemma *inverse-less-iff-less-neg* [*simp*,*noatp*]:
[|*a < 0*; *b < 0*|] ==> *(inverse a < inverse b) = (b < (a::'a::ordered-field))*
⟨*proof*⟩

lemma *le-imp-inverse-le-neg*:
[|*a ≤ b*; *b < 0*|] ==> *inverse b ≤ inverse (a::'a::ordered-field)*
⟨*proof*⟩

lemma *inverse-le-iff-le-neg* [*simp*,*noatp*]:
[|*a < 0*; *b < 0*|] ==> *(inverse a ≤ inverse b) = (b ≤ (a::'a::ordered-field))*
⟨*proof*⟩

13.5 Inverses and the Number One

lemma *one-less-inverse-iff*:
(1 < inverse x) = (0 < x & x < (1::'a::{ordered-field,division-by-zero}))
⟨*proof*⟩

lemma *inverse-eq-1-iff* [*simp*]:
(inverse x = 1) = (x = (1::'a::{field,division-by-zero}))
⟨*proof*⟩

lemma *one-le-inverse-iff*:
(1 ≤ inverse x) = (0 < x & x ≤ (1::'a::{ordered-field,division-by-zero}))
⟨*proof*⟩

lemma *inverse-less-1-iff*:
(inverse x < 1) = (x ≤ 0 | 1 < (x::'a::{ordered-field,division-by-zero}))
⟨*proof*⟩

lemma *inverse-le-1-iff*:
(inverse x ≤ 1) = (x ≤ 0 | 1 ≤ (x::'a::{ordered-field,division-by-zero}))
⟨*proof*⟩

13.6 Simplification of Inequalities Involving Literal Divisors

lemma *pos-le-divide-eq*: *0 < (c::'a::ordered-field) ==> (a ≤ b/c) = (a*c ≤ b)*
⟨*proof*⟩

lemma *neg-le-divide-eq*: *c < (0::'a::ordered-field) ==> (a ≤ b/c) = (b ≤ a*c)*
⟨*proof*⟩

lemma *le-divide-eq*:
(a ≤ b/c) =

$(\mathit{if}\ 0 < c\ \mathit{then}\ a*c \leq b$
$\quad \mathit{else\ if}\ c < 0\ \mathit{then}\ b \leq a*c$
$\quad \mathit{else}\ \ a \leq (0::'a::\{\mathit{ordered\text{-}field}, \mathit{division\text{-}by\text{-}zero}\}))$
⟨*proof*⟩

lemma *pos-divide-le-eq*: $0 < (c::'a::\mathit{ordered\text{-}field}) ==> (b/c \leq a) = (b \leq a*c)$
⟨*proof*⟩

lemma *neg-divide-le-eq*: $c < (0::'a::\mathit{ordered\text{-}field}) ==> (b/c \leq a) = (a*c \leq b)$
⟨*proof*⟩

lemma *divide-le-eq*:
$(b/c \leq a) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ b \leq a*c$
$\quad \mathit{else\ if}\ c < 0\ \mathit{then}\ a*c \leq b$
$\quad \mathit{else}\ 0 \leq (a::'a::\{\mathit{ordered\text{-}field}, \mathit{division\text{-}by\text{-}zero}\}))$
⟨*proof*⟩

lemma *pos-less-divide-eq*:
$0 < (c::'a::\mathit{ordered\text{-}field}) ==> (a < b/c) = (a*c < b)$
⟨*proof*⟩

lemma *neg-less-divide-eq*:
$c < (0::'a::\mathit{ordered\text{-}field}) ==> (a < b/c) = (b < a*c)$
⟨*proof*⟩

lemma *less-divide-eq*:
$(a < b/c) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ a*c < b$
$\quad \mathit{else\ if}\ c < 0\ \mathit{then}\ b < a*c$
$\quad \mathit{else}\ \ a < (0::'a::\{\mathit{ordered\text{-}field}, \mathit{division\text{-}by\text{-}zero}\}))$
⟨*proof*⟩

lemma *pos-divide-less-eq*:
$0 < (c::'a::\mathit{ordered\text{-}field}) ==> (b/c < a) = (b < a*c)$
⟨*proof*⟩

lemma *neg-divide-less-eq*:
$c < (0::'a::\mathit{ordered\text{-}field}) ==> (b/c < a) = (a*c < b)$
⟨*proof*⟩

lemma *divide-less-eq*:
$(b/c < a) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ b < a*c$
$\quad \mathit{else\ if}\ c < 0\ \mathit{then}\ a*c < b$
$\quad \mathit{else}\ 0 < (a::'a::\{\mathit{ordered\text{-}field}, \mathit{division\text{-}by\text{-}zero}\}))$
⟨*proof*⟩

13.7 Field simplification

Lemmas *field-simps* multiply with denominators in in(equations) if they can be proved to be non-zero (for equations) or positive/negative (for inequations).

lemmas *field-simps = field-eq-simps*

pos-divide-less-eq neg-divide-less-eq
pos-less-divide-eq neg-less-divide-eq
pos-divide-le-eq neg-divide-le-eq
pos-le-divide-eq neg-le-divide-eq

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

lemmas *sign-simps = group-simps*
zero-less-mult-iff mult-less-0-iff

13.8 Division and Signs

lemma *zero-less-divide-iff*:
$((0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}) < a/b) = (0 < a \ \&\ 0 < b \mid a < 0 \ \&\ b < 0)$
⟨*proof*⟩

lemma *divide-less-0-iff*:
$(a/b < (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})) =$
$(0 < a \ \&\ b < 0 \mid a < 0 \ \&\ 0 < b)$
⟨*proof*⟩

lemma *zero-le-divide-iff*:
$((0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}) \leq a/b) =$
$(0 \leq a \ \&\ 0 \leq b \mid a \leq 0 \ \&\ b \leq 0)$
⟨*proof*⟩

lemma *divide-le-0-iff*:
$(a/b \leq (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})) =$
$(0 \leq a \ \&\ b \leq 0 \mid a \leq 0 \ \&\ 0 \leq b)$
⟨*proof*⟩

lemma *divide-eq-0-iff* [*simp*,*noatp*]:
$(a/b = 0) = (a=0 \mid b=(0::'a::\{field, division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *divide-pos-pos*:
$0 < (x::'a::ordered\text{-}field) \Longrightarrow 0 < y \Longrightarrow 0 < x \ / \ y$
⟨*proof*⟩

lemma *divide-nonneg-pos*:
0 <= (x::'a::ordered-field) ==> 0 < y ==> 0 <= x / y
⟨*proof*⟩

lemma *divide-neg-pos*:
(x::'a::ordered-field) < 0 ==> 0 < y ==> x / y < 0
⟨*proof*⟩

lemma *divide-nonpos-pos*:
(x::'a::ordered-field) <= 0 ==> 0 < y ==> x / y <= 0
⟨*proof*⟩

lemma *divide-pos-neg*:
0 < (x::'a::ordered-field) ==> y < 0 ==> x / y < 0
⟨*proof*⟩

lemma *divide-nonneg-neg*:
0 <= (x::'a::ordered-field) ==> y < 0 ==> x / y <= 0
⟨*proof*⟩

lemma *divide-neg-neg*:
(x::'a::ordered-field) < 0 ==> y < 0 ==> 0 < x / y
⟨*proof*⟩

lemma *divide-nonpos-neg*:
(x::'a::ordered-field) <= 0 ==> y < 0 ==> 0 <= x / y
⟨*proof*⟩

13.9 Cancellation Laws for Division

lemma *divide-cancel-right* [*simp*,*noatp*]:
(a/c = b/c) = (c = 0 | a = (b::'a::{field,division-by-zero}))
⟨*proof*⟩

lemma *divide-cancel-left* [*simp*,*noatp*]:
(c/a = c/b) = (c = 0 | a = (b::'a::{field,division-by-zero}))
⟨*proof*⟩

13.10 Division and the Number One

Simplify expressions equated with 1

lemma *divide-eq-1-iff* [*simp*,*noatp*]:
(a/b = 1) = (b ≠ 0 & a = (b::'a::{field,division-by-zero}))
⟨*proof*⟩

lemma *one-eq-divide-iff* [*simp*,*noatp*]:
(1 = a/b) = (b ≠ 0 & a = (b::'a::{field,division-by-zero}))
⟨*proof*⟩

lemma *zero-eq-1-divide-iff* [*simp*,*noatp*]:
((0::'a::{*ordered-field*,*division-by-zero*}) = 1 / a) = (a = 0)
⟨*proof*⟩

lemma *one-divide-eq-0-iff* [*simp*,*noatp*]:
(1 / a = (0::'a::{*ordered-field*,*division-by-zero*})) = (a = 0)
⟨*proof*⟩

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemmas *zero-less-divide-1-iff* = *zero-less-divide-iff* [*of 1*, *simplified*]
lemmas *divide-less-0-1-iff* = *divide-less-0-iff* [*of 1*, *simplified*]
lemmas *zero-le-divide-1-iff* = *zero-le-divide-iff* [*of 1*, *simplified*]
lemmas *divide-le-0-1-iff* = *divide-le-0-iff* [*of 1*, *simplified*]

declare *zero-less-divide-1-iff* [*simp*]
declare *divide-less-0-1-iff* [*simp*,*noatp*]
declare *zero-le-divide-1-iff* [*simp*]
declare *divide-le-0-1-iff* [*simp*,*noatp*]

13.11 Ordering Rules for Division

lemma *divide-strict-right-mono*:
[|a < b; 0 < c|] ==> a / c < b / (c::'a::*ordered-field*)
⟨*proof*⟩

lemma *divide-right-mono*:
[|a ≤ b; 0 ≤ c|] ==> a/c ≤ b/(c::'a::{*ordered-field*,*division-by-zero*})
⟨*proof*⟩

lemma *divide-right-mono-neg*: (a::'a::{*division-by-zero*,*ordered-field*}) <= b
==> c <= 0 ==> b / c <= a / c
⟨*proof*⟩

lemma *divide-strict-right-mono-neg*:
[|b < a; c < 0|] ==> a / c < b / (c::'a::*ordered-field*)
⟨*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::'a::*ordered-field*)
⟨*proof*⟩

lemma *divide-left-mono*:
[|b ≤ a; 0 ≤ c; 0 < a*b|] ==> c / a ≤ c / (b::'a::*ordered-field*)
⟨*proof*⟩

lemma *divide-left-mono-neg*: (a::'a::{*division-by-zero*,*ordered-field*}) <= b
==> c <= 0 ==> 0 < a * b ==> c / a <= c / b

⟨*proof*⟩

lemma *divide-strict-left-mono-neg*:
[|*a* < *b*; *c* < *0*; *0* < *a*∗*b*|] ==> *c* / *a* < *c* / (*b*::'*a*::*ordered-field*)
⟨*proof*⟩

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* ≤ *b* / *a*) = ((*0* < *a* & *a* ≤ *b*) | (*a* < *0* & *b* ≤ *a*))
⟨*proof*⟩

lemma *divide-le-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*b* / *a* ≤ *1*) = ((*0* < *a* & *b* ≤ *a*) | (*a* < *0* & *a* ≤ *b*) | *a*=*0*)
⟨*proof*⟩

lemma *less-divide-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* < *b* / *a*) = ((*0* < *a* & *a* < *b*) | (*a* < *0* & *b* < *a*))
⟨*proof*⟩

lemma *divide-less-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*b* / *a* < *1*) = ((*0* < *a* & *b* < *a*) | (*a* < *0* & *a* < *b*) | *a*=*0*)
⟨*proof*⟩

13.12 Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows *0* < *a* ⟹ (*1* ≤ *b*/*a*) = (*a* ≤ *b*)
⟨*proof*⟩

lemma *le-divide-eq-1-neg* [*simp*,*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows *a* < *0* ⟹ (*1* ≤ *b*/*a*) = (*b* ≤ *a*)
⟨*proof*⟩

lemma *divide-le-eq-1-pos* [*simp*,*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows *0* < *a* ⟹ (*b*/*a* ≤ *1*) = (*b* ≤ *a*)
⟨*proof*⟩

lemma *divide-le-eq-1-neg* [*simp*,*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows *a* < *0* ⟹ (*b*/*a* ≤ *1*) = (*a* ≤ *b*)
⟨*proof*⟩

lemma *less-divide-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (1 < b/a) = (a < b)$
⟨*proof*⟩

lemma *less-divide-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (1 < b/a) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (b/a < 1) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow b/a < 1 <-> a < b$
⟨*proof*⟩

lemma *eq-divide-eq-1* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(1 = b/a) = ((a \neq 0 \ \& \ a = b))$
⟨*proof*⟩

lemma *divide-eq-eq-1* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(b/a = 1) = ((a \neq 0 \ \& \ a = b))$
⟨*proof*⟩

13.13 Reasoning about inequalities with division

lemma *mult-right-le-one-le*: *0 <= (x::'a::ordered-idom) ==> 0 <= y ==> y <= 1*
*==> x * y <= x*
⟨*proof*⟩

lemma *mult-left-le-one-le*: *0 <= (x::'a::ordered-idom) ==> 0 <= y ==> y <= 1*
*==> y * x <= x*
⟨*proof*⟩

lemma *mult-imp-div-pos-le*: *0 < (y::'a::ordered-field) ==> x <= z * y ==>*
x / y <= z
⟨*proof*⟩

lemma *mult-imp-le-div-pos*: *0 < (y::'a::ordered-field) ==> z * y <= x ==>*
z <= x / y
⟨*proof*⟩

lemma *mult-imp-div-pos-less*: $0 < (y::'a::ordered\text{-}field) ==> x < z * y ==>$
$x / y < z$
⟨*proof*⟩

lemma *mult-imp-less-div-pos*: $0 < (y::'a::ordered\text{-}field) ==> z * y < x ==>$
$z < x / y$
⟨*proof*⟩

lemma *frac-le*: $(0::'a::ordered\text{-}field) <= x ==>$
$x <= y ==> 0 < w ==> w <= z ==> x / z <= y / w$
⟨*proof*⟩

lemma *frac-less*: $(0::'a::ordered\text{-}field) <= x ==>$
$x < y ==> 0 < w ==> w <= z ==> x / z < y / w$
⟨*proof*⟩

lemma *frac-less2*: $(0::'a::ordered\text{-}field) < x ==>$
$x <= y ==> 0 < w ==> w < z ==> x / z < y / w$
⟨*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.

declare *times-divide-eq* [*simp*]

13.14 Ordered Fields are Dense

context *ordered-semidom*
begin

lemma *less-add-one*: $a < a + 1$
⟨*proof*⟩

lemma *zero-less-two*: $0 < 1 + 1$
⟨*proof*⟩

end

lemma *less-half-sum*: $a < b ==> a < (a+b) / (1+1::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *gt-half-sum*: $a < b ==> (a+b)/(1+1::'a::ordered\text{-}field) < b$
⟨*proof*⟩

instance *ordered-field* < *dense-linear-order*
⟨*proof*⟩

13.15 Absolute Value

context *ordered-idom*
begin

lemma *mult-sgn-abs*: $sgn\ x * abs\ x = x$
⟨*proof*⟩

end

lemma *abs-one* [*simp*]: $abs\ 1 = (1::'a::ordered\text{-}idom)$
⟨*proof*⟩

class *pordered-ring-abs* = *pordered-ring* + *pordered-ab-group-add-abs* +
assumes *abs-eq-mult*:
$(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \Longrightarrow |a * b| = |a| * |b|$

class *lordered-ring* = *pordered-ring* + *lordered-ab-group-add-abs*
begin

subclass *lordered-ab-group-add-meet* ⟨*proof*⟩
subclass *lordered-ab-group-add-join* ⟨*proof*⟩

end

lemma *abs-le-mult*: $abs\ (a * b) \leq (abs\ a) * (abs\ (b::'a::lordered\text{-}ring))$
⟨*proof*⟩

instance *lordered-ring* ⊆ *pordered-ring-abs*
⟨*proof*⟩

instance *ordered-idom* ⊆ *pordered-ring-abs*
⟨*proof*⟩

lemma *abs-mult*: $abs\ (a * b) = abs\ a * abs\ (b::'a::ordered\text{-}idom)$
⟨*proof*⟩

lemma *abs-mult-self*: $abs\ a * abs\ a = a * (a::'a::ordered\text{-}idom)$
⟨*proof*⟩

lemma *nonzero-abs-inverse*:
$a \neq 0 \Longrightarrow abs\ (inverse\ (a::'a::ordered\text{-}field)) = inverse\ (abs\ a)$
⟨*proof*⟩

lemma *abs-inverse* [*simp*]:
$abs\ (inverse\ (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})) =$
$inverse\ (abs\ a)$
⟨*proof*⟩

lemma *nonzero-abs-divide*:
b ≠ 0 ==> abs (a / (b::'a::ordered-field)) = abs a / abs b
⟨*proof*⟩

lemma *abs-divide* [*simp*]:
abs (a / (b::'a::{ordered-field,division-by-zero})) = abs a / abs b
⟨*proof*⟩

lemma *abs-mult-less*:
*[| abs a < c; abs b < d |] ==> abs a * abs b < c*(d::'a::ordered-idom)*
⟨*proof*⟩

lemmas *eq-minus-self-iff = equal-neg-zero*

lemma *less-minus-self-iff*: *(a < -a) = (a < (0::'a::ordered-idom))*
⟨*proof*⟩

lemma *abs-less-iff*: *(abs a < b) = (a < b & -a < (b::'a::ordered-idom))*
⟨*proof*⟩

lemma *abs-mult-pos*: *(0::'a::ordered-idom) <= x ==>*
*(abs y) * x = abs (y * x)*
⟨*proof*⟩

lemma *abs-div-pos*: *(0::'a::{division-by-zero,ordered-field}) < y ==>*
abs x / y = abs (x / y)
⟨*proof*⟩

13.16 Bounds of products via negative and positive Part

lemma *mult-le-prts*:
assumes
a1 <= (a::'a::lordered-ring)
a <= a2
b1 <= b
b <= b2
shows
*a * b <= pprt a2 * pprt b2 + pprt a1 * nprt b2 + nprt a2 * pprt b1 + nprt a1 * nprt b1*
⟨*proof*⟩

lemma *mult-ge-prts*:
assumes
a1 <= (a::'a::lordered-ring)
a <= a2
b1 <= b
b <= b2
shows
*a * b >= nprt a1 * pprt b2 + nprt a2 * nprt b2 + pprt a1 * pprt b1 + pprt a2*

∗ *nprt b1*
⟨*proof*⟩

end

14 Nat: Natural numbers

theory *Nat*
imports *Inductive Ring-and-Field*
uses
~~/*src*/*Tools*/*rat.ML*
~~/*src*/*Provers*/*Arith*/*cancel-sums.ML*
(*arith-data.ML*)
~~/*src*/*Provers*/*Arith*/*fast-lin-arith.ML*
(*Tools*/*lin-arith.ML*)
begin

14.1 Type *ind*

typedecl *ind*

axiomatization
Zero-Rep :: *ind* **and**
Suc-Rep :: *ind* => *ind*
where
— the axiom of infinity in 2 parts
inj-Suc-Rep: *inj Suc-Rep* **and**
Suc-Rep-not-Zero-Rep: *Suc-Rep x* ≠ *Zero-Rep*

14.2 Type nat

Type definition

inductive *Nat* :: *ind* ⇒ *bool*
where
Zero-RepI: *Nat Zero-Rep*
| *Suc-RepI*: *Nat i* ⟹ *Nat* (*Suc-Rep i*)

global

typedef (**open** *Nat*)
nat = *Collect Nat*
⟨*proof*⟩

constdefs
Suc :: *nat* => *nat*
Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))

local

instantiation *nat* :: *zero*
begin

definition *Zero-nat-def* [*code func del*]:
0 = Abs-Nat Zero-Rep

instance ⟨*proof*⟩

end

lemma *nat-induct*: *P 0 ==> (!!n. P n ==> P (Suc n)) ==> P n*
⟨*proof*⟩

lemma *Suc-not-Zero* [*iff*]: *Suc m ≠ 0*
⟨*proof*⟩

lemma *Zero-not-Suc* [*iff*]: *0 ≠ Suc m*
⟨*proof*⟩

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
⟨*proof*⟩

lemma *Suc-Suc-eq* [*iff*]: *Suc m = Suc n ⟷ m = n*
⟨*proof*⟩

rep-datatype *nat*
distinct *Suc-not-Zero Zero-not-Suc*
inject *Suc-Suc-eq*
induction *nat-induct*

declare *nat.induct* [*case-names 0 Suc*, *induct type*: *nat*]
declare *nat.exhaust* [*case-names 0 Suc*, *cases type*: *nat*]

lemmas *nat-rec-0 = nat.recs(1)*
and *nat-rec-Suc = nat.recs(2)*

lemmas *nat-case-0 = nat.cases(1)*
and *nat-case-Suc = nat.cases(2)*

Injectiveness and distinctness lemmas

lemma *Suc-neq-Zero*: *Suc m = 0 ⟹ R*
⟨*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*: $n \neq Suc\ n$
⟨*proof*⟩

lemma *Suc-n-not-n*: $Suc\ n \neq n$
⟨*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*⟩

14.3 Arithmetic operators

instantiation *nat* :: {*minus*, *comm-monoid-add*}
begin

primrec *plus-nat*
where
add-0: $0 + n = (n::nat)$
| *add-Suc*: $Suc\ m + n = Suc\ (m + n)$

lemma *add-0-right* [*simp*]: $m + 0 = (m::nat)$
⟨*proof*⟩

lemma *add-Suc-right* [*simp*]: $m + Suc\ n = Suc\ (m + n)$
⟨*proof*⟩

lemma *add-Suc-shift* [*code*]: $Suc\ m + n = m + Suc\ n$
⟨*proof*⟩

primrec *minus-nat*
where
diff-0: $m - 0 = (m::nat)$
| *diff-Suc*: $m - Suc\ n = (case\ m - n\ of\ 0 => 0 \mid Suc\ k => k)$

declare *diff-Suc* [*simp del*, *code del*]

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

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

14.3.1 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 | m=0 & n= Suc 0)

⟨*proof*⟩

lemma *one-is-add*:
(*Suc 0* = *m* + *n*) = (*m* = *Suc 0* & *n* = *0* | *m* = *0* & *n* = *Suc 0*)
⟨*proof*⟩

lemma *add-eq-self-zero*:
fixes *m n* :: *nat*
shows *m* + *n* = *m* ⟹ *n* = *0*
⟨*proof*⟩

lemma *inj-on-add-nat*[*simp*]: *inj-on* (%*n*::*nat*. *n*+*k*) *N*
⟨*proof*⟩

14.3.2 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*⟩

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*⟩

14.3.3 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*]: $(m * n = Suc\ 0) = (m = 1\ \&\ n = 1)$
⟨*proof*⟩

lemma *one-eq-mult-iff* [*simp*,*noatp*]: $(Suc\ 0 = m * n) = (m = 1\ \&\ n = 1)$
⟨*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*⟩

14.4 Orders on *nat*

14.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat* **where**
$(0::nat) \leq n \longleftrightarrow True$
| $Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*, *code del*]
lemma [*code*]: $(0::nat) \leq n \longleftrightarrow True$ ⟨*proof*⟩
lemma *le0* [*iff*]: $0 \leq (n::nat)$ ⟨*proof*⟩

definition *less-nat* **where**
less-eq-Suc-le [*code func del*]: $n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$
⟨*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*: $Suc\ m \leq n \Longrightarrow m \leq n$
⟨*proof*⟩

lemma *less-SucI*: $m < n \Longrightarrow m < Suc\ n$
⟨*proof*⟩

lemma *Suc-lessD*: $Suc\ m < n \Longrightarrow m < n$
⟨*proof*⟩

instance
⟨*proof*⟩

end

14.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
⟨*proof*⟩

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
⟨*proof*⟩

14.4.3 Elimination properties

lemma *less-not-refl*: $\sim\ n < (n::nat)$
⟨*proof*⟩

lemma *less-not-refl2*: $n < m \Longrightarrow m \neq (n::nat)$
⟨*proof*⟩

lemma *less-not-refl3*: $(s::nat) < t \Longrightarrow s \neq t$
⟨*proof*⟩

lemma *less-irrefl-nat*: $(n::nat) < n \Longrightarrow R$
⟨*proof*⟩

lemma *less-zeroE*: $(n::nat) < 0 \Longrightarrow R$
⟨*proof*⟩

lemma *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
⟨*proof*⟩

lemma *less-one* [*iff*, *noatp*]: $(n < (1::nat)) = (n = 0)$
⟨*proof*⟩

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
⟨*proof*⟩

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
⟨*proof*⟩

"Less than" is antisymmetric, sort of

lemma *less-antisym*: $[\![\neg\ n < m;\ n < Suc\ m\ ]\!] \Longrightarrow m = n$
⟨*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*⟩

14.4.4 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*: $[\!| \ i \leq j; j \leq k \ |\!] \Longrightarrow i \leq (k::nat)$
⟨*proof*⟩

lemma *le-anti-sym*: $[\!| \ m \leq n; n \leq m \ |\!] \Longrightarrow m = (n::nat)$
⟨*proof*⟩

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \ \& \ m \neq n)$
⟨*proof*⟩

lemma *le-neq-implies-less*: $(m::nat) \leq n \Longrightarrow m \neq n \Longrightarrow m < n$
⟨*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*: $m \leq n \Longrightarrow (n < Suc\ m) = (n = m)$
⟨*proof*⟩

lemma *not-less-less-Suc-eq*: $\sim n < m \Longrightarrow (n < Suc\ m) = (n = m)$
⟨*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*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n) \Longrightarrow f\ 0 = c$
⟨*proof*⟩

lemma *def-nat-rec-Suc*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n) \Longrightarrow f\ (Suc\ n) = h\ n\ (f\ n)$
⟨*proof*⟩

lemma *not0-implies-Suc*: $n \neq 0 \Longrightarrow \exists\, m.\ n = Suc\ m$
⟨*proof*⟩

lemma *gr0-implies-Suc*: $n > 0 \Longrightarrow \exists\, m.\ n = Suc\ m$

⟨*proof*⟩

lemma *gr-implies-not0*: **fixes** n :: *nat* **shows** $m<n \Longrightarrow n \neq 0$
⟨*proof*⟩

lemma *neq0-conv*[*iff*]: **fixes** n :: *nat* **shows** $(n \neq 0) = (0 < n)$
⟨*proof*⟩

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0 \Longrightarrow False) \Longrightarrow 0 < n$
⟨*proof*⟩

lemma *gr0-conv-Suc*: $(0 < n) = (\exists m.\ n = Suc\ m)$
⟨*proof*⟩

lemma *not-gr0* [*iff*,*noatp*]: $!!n::nat.\ (\sim (0 < n)) = (n = 0)$
⟨*proof*⟩

lemma *Suc-le-D*: $(Suc\ n \leq m') \Longrightarrow (?\ m.\ m' = Suc\ m)$
⟨*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*⟩

14.4.5 *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*:
$min\ (Suc\ n)\ m = (case\ m\ of\ 0 \Rightarrow 0 \mid Suc\ m' \Rightarrow Suc(min\ n\ m'))$
⟨*proof*⟩

lemma *min-Suc2*:
$min\ m\ (Suc\ n) = (case\ m\ of\ 0 \Rightarrow 0 \mid Suc\ m' \Rightarrow Suc(min\ m'\ n))$
⟨*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* | *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*⟩

14.4.6 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: *n*>*0* ==> *Suc* (*n* − *Suc 0*) = *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* :: *ordered-semidom*
⟨*proof*⟩

lemma *nat-mult-1*: $(1::nat) * n = n$
⟨*proof*⟩

lemma *nat-mult-1-right*: $n * (1::nat) = n$
⟨*proof*⟩

14.4.7 Additional theorems about *op* ≤

Complete induction, aka course-of-values induction

lemma *less-induct* [*case-names less*]:
fixes $P :: nat \Rightarrow bool$
assumes *step*: $\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows $P\ a$
⟨*proof*⟩

lemma *nat-less-induct*:
assumes $!!n.\ \forall m::nat.\ m < n \longrightarrow P\ m \Longrightarrow P\ n$ **shows** $P\ n$
⟨*proof*⟩

lemma *measure-induct-rule* [*case-names less*]:
fixes $f :: 'a \Rightarrow nat$
assumes *step*: $\bigwedge x.\ (\bigwedge y.\ f\ y < f\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows $P\ a$
⟨*proof*⟩

old style induction rules:

lemma *measure-induct*:
fixes $f :: 'a \Rightarrow nat$
shows $(\bigwedge x.\ \forall y.\ f\ y < f\ x \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
⟨*proof*⟩

lemma *full-nat-induct*:
assumes *step*: $(!!n.\ (ALL\ m.\ Suc\ m <= n \longrightarrow P\ m) \Longrightarrow P\ n)$
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*: $!!i\ j\ k.\ P\ i\ j \Longrightarrow P\ j\ k \Longrightarrow P\ i\ k$
shows $P\ i\ j$
⟨*proof*⟩

lemma *nat-induct2*: $[|P\ 0;\ P\ (Suc\ 0);\ !!k.\ P\ k \Longrightarrow P\ (Suc\ (Suc\ k))|] \Longrightarrow P\ n$
⟨*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*: $(i::nat) \leq j ==> i \leq j + m$
⟨*proof*⟩

lemma *trans-le-add2*: $(i::nat) \leq j ==> i \leq m + j$
⟨*proof*⟩

lemma *trans-less-add1*: $(i::nat) < j ==> i < j + m$
⟨*proof*⟩

lemma *trans-less-add2*: $(i::nat) < j ==> i < m + j$
⟨*proof*⟩

lemma *add-lessD1*: $i + j < (k::nat) ==> i < k$
⟨*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*: $m + k \leq n ==> m \leq (n::nat)$
⟨*proof*⟩

lemma *add-leD2*: $m + k \leq n ==> k \leq (n::nat)$
⟨*proof*⟩

lemma *add-leE*: $(m::nat) + k \leq n ==> (m \leq n ==> k \leq n ==> R) ==> R$
⟨*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*⟩

14.4.8 More results about difference

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse*: $\sim\ m < n \Longrightarrow n + (m - n) = (m{::}nat)$
⟨*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*⟩

14.4.9 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*]: $(Suc\ 0 \le m * n) = (1 \le m\ \&\ 1 \le n)$
⟨*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

14.5 Embedding of the Naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

primrec
of-nat :: $nat \Rightarrow 'a$
where
of-nat-0: $of\text{-}nat\ 0 = 0$
| *of-nat-Suc*: $of\text{-}nat\ (Suc\ m) = 1 + of\text{-}nat\ m$

lemma *of-nat-1* [*simp*]: *of-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*⟩

definition
of-nat-aux :: *nat ⇒ 'a ⇒ 'a*
where
[*code func del*]: *of-nat-aux n i = of-nat n + i*

lemma *of-nat-aux-code* [*code*]:
of-nat-aux 0 i = i
of-nat-aux (Suc n) i = of-nat-aux n (i + 1) — tail recursive
⟨*proof*⟩

lemma *of-nat-code* [*code*]:
of-nat n = of-nat-aux n 0
⟨*proof*⟩

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0 = semiring-1 +*
assumes *of-nat-eq-iff* [*simp*]: *of-nat m = of-nat n ⟷ m = n*
begin

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*, *noatp*]: *0 = of-nat n ⟷ 0 = n*
⟨*proof*⟩

lemma *of-nat-eq-0-iff* [*simp*, *noatp*]: *of-nat m = 0 ⟷ m = 0*
⟨*proof*⟩

lemma *inj-of-nat*: *inj of-nat*
⟨*proof*⟩

end

context *ordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0 ≤ of-nat m*
⟨*proof*⟩

lemma *less-imp-of-nat-less*: $m < n \Longrightarrow \mathit{of\text{-}nat}\ m < \mathit{of\text{-}nat}\ n$
⟨*proof*⟩

lemma *of-nat-less-imp-less*: $\mathit{of\text{-}nat}\ m < \mathit{of\text{-}nat}\ n \Longrightarrow m < n$
⟨*proof*⟩

lemma *of-nat-less-iff* [*simp*]: $\mathit{of\text{-}nat}\ m < \mathit{of\text{-}nat}\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *of-nat-le-iff* [*simp*]: $\mathit{of\text{-}nat}\ m \leq \mathit{of\text{-}nat}\ n \longleftrightarrow m \leq n$
⟨*proof*⟩

Every *ordered-semidom* has characteristic zero.

subclass *semiring-char-0*
⟨*proof*⟩

Special cases where either operand is zero

lemma *of-nat-0-le-iff* [*simp*]: $0 \leq \mathit{of\text{-}nat}\ n$
⟨*proof*⟩

lemma *of-nat-le-0-iff* [*simp*, *noatp*]: $\mathit{of\text{-}nat}\ m \leq 0 \longleftrightarrow m = 0$
⟨*proof*⟩

lemma *of-nat-0-less-iff* [*simp*]: $0 < \mathit{of\text{-}nat}\ n \longleftrightarrow 0 < n$
⟨*proof*⟩

lemma *of-nat-less-0-iff* [*simp*]: $\neg\ \mathit{of\text{-}nat}\ m < 0$
⟨*proof*⟩

end

context *ring-1*
begin

lemma *of-nat-diff*: $n \leq m \Longrightarrow \mathit{of\text{-}nat}\ (m - n) = \mathit{of\text{-}nat}\ m - \mathit{of\text{-}nat}\ n$
⟨*proof*⟩

end

context *ordered-idom*
begin

lemma *abs-of-nat* [*simp*]: $|\mathit{of\text{-}nat}\ n| = \mathit{of\text{-}nat}\ n$
⟨*proof*⟩

end

lemma *of-nat-id* [*simp*]: $\mathit{of\text{-}nat}\ n = n$

⟨*proof*⟩

lemma *of-nat-eq-id* [*simp*]: *of-nat* = *id*
⟨*proof*⟩

14.6 The Set of Natural Numbers

context *semiring-1*
begin

definition
Nats :: *'a set* **where**
Nats = *range of-nat*

notation (*xsymbols*)
Nats (ℕ)

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ ℕ
⟨*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*⟩

end

14.7 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*

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: [| *a* < (*b*::*nat*); *c* ≤ *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*: $k \leq j \Longrightarrow (i \leq j-k) = (i+k \leq (j::nat))$
⟨*proof*⟩

lemma *diff-diff-cancel* [*simp*]: $i \leq (n::nat) \Longrightarrow n - (n - i) = i$
⟨*proof*⟩

lemma *le-add-diff*: $k \leq (n::nat) \Longrightarrow m \leq n + m - k$
⟨*proof*⟩

lemma *diff-less*[*simp*]: $!!m::nat.\ [|\ 0<n;\ 0<m\ |] \Longrightarrow m - n < m$
⟨*proof*⟩

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow ((m-k) - (n-k)) = (m-n)$
⟨*proof*⟩

lemma *eq-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k = n-k) = (m=n)$
⟨*proof*⟩

lemma *less-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k < n-k) = (m<n)$
⟨*proof*⟩

lemma *le-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k \leq n-k) = (m \leq n)$
⟨*proof*⟩

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat) \Longrightarrow (m-l) \leq (n-l)$
⟨*proof*⟩

lemma *diff-le-mono2*: $m \leq (n::nat) \Longrightarrow (l-n) \leq (l-m)$
⟨*proof*⟩

lemma *diff-less-mono2*: $[|\ m < (n::nat);\ m<l\ |] \Longrightarrow (l-n) < (l-m)$
⟨*proof*⟩

lemma *diffs0-imp-equal*: $!!m::nat.\ [|\ m-n = 0;\ n-m = 0\ |] \Longrightarrow\ m=n$
⟨*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*]: $k \leq j \longrightarrow i - (j - k) = i + (k::nat) - j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j \Longrightarrow m - Suc\ (j - k) = m + k - Suc\ j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j \Longrightarrow Suc\ (j - k) - m = Suc\ j - (k + m)$
⟨*proof*⟩

Lemmas for ex/Factorization

lemma *one-less-mult*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow Suc\ 0 < m*n$
⟨*proof*⟩

lemma *n-less-m-mult-n*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow n<m*n$
⟨*proof*⟩

lemma *n-less-n-mult-m*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow n<n*m$
⟨*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*: $!!i.\ [|\ i < j;\ P\ (Suc\ i)\ |] \Longrightarrow P\ i$
shows $P\ i$
⟨*proof*⟩

lemma *strict-inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: $i < j$
assumes *base*: $!!i.\ j = Suc\ i \Longrightarrow P\ i$
assumes *step*: $!!i.\ [|\ i < j;\ P\ (Suc\ i)\ |] \Longrightarrow P\ i$
shows $P\ i$
⟨*proof*⟩

lemma *zero-induct-lemma*: $P\ k \Longrightarrow (!!n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ (k - i)$
⟨*proof*⟩

lemma *zero-induct*: $P\ k \Longrightarrow (!!n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ 0$
⟨*proof*⟩

lemma *nat-not-singleton*: $(\forall\, x.\ x = (0::nat)) = False$
⟨*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.

14.8 size of a datatype value

class *size* = *type* +
  fixes *size* :: *'a* ⇒ *nat* — see further theory *Wellfounded*

end

15 Power: Exponentiation

theory *Power*
imports *Nat*
begin

class *power* = *type* +
  fixes *power* :: *'a* ⇒ *nat* ⇒ *'a* (**infixr** ^ *80*)

15.1 Powers for Arbitrary Monoids

class *recpower* = *monoid-mult* + *power* +
  assumes *power-0* [*simp*]: *a ^ 0* = *1*
  assumes *power-Suc*: *a ^ Suc n* = *a * (a ^ n)*

lemma *power-0-Suc* [*simp*]: *(0::'a::{recpower,semiring-0}) ^ (Suc n)* = *0*
  ⟨*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::{recpower,semiring-0}))*
  ⟨*proof*⟩

lemma *power-one* [*simp*]: *1^n* = *(1::'a::recpower)*
  ⟨*proof*⟩

lemma *power-one-right* [*simp*]: *(a::'a::recpower) ^ 1* = *a*
  ⟨*proof*⟩

lemma *power-commutes*: *(a::'a::recpower) ^ n * a* = *a * a ^ n*
  ⟨*proof*⟩

lemma *power-add*: *(a::'a::recpower) ^ (m+n)* = *(a^m) * (a^n)*

⟨*proof*⟩

lemma *power-mult*: $(a::'a::recpower)\ \hat{}\ (m*n) = (a\hat{}m)\ \hat{}\ n$
⟨*proof*⟩

lemma *power-mult-distrib*: $((a::'a::\{recpower,comm\text{-}monoid\text{-}mult\})\ *\ b)\ \hat{}\ n = (a\hat{}n) * (b\hat{}n)$
⟨*proof*⟩

lemma *zero-less-power*[*simp*]:
$0 < (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 0 < a\hat{}n$
⟨*proof*⟩

lemma *zero-le-power*[*simp*]:
$0 \leq (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 0 \leq a\hat{}n$
⟨*proof*⟩

lemma *one-le-power*[*simp*]:
$1 \leq (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 1 \leq a\hat{}n$
⟨*proof*⟩

lemma *gt1-imp-ge0*: $1 < a ==> 0 \leq (a::'a::ordered\text{-}semidom)$
⟨*proof*⟩

lemma *power-gt1-lemma*:
assumes *gt1*: $1 < (a::'a::\{ordered\text{-}semidom,recpower\})$
shows $1 < a * a\hat{}n$
⟨*proof*⟩

lemma *one-less-power*[*simp*]:
$[\![1 < (a::'a::\{ordered\text{-}semidom,recpower\}); 0 < n]\!] \Longrightarrow 1 < a\ \hat{}\ n$
⟨*proof*⟩

lemma *power-gt1*:
$1 < (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 1 < a\ \hat{}\ (Suc\ n)$
⟨*proof*⟩

lemma *power-le-imp-le-exp*:
assumes *gt1*: $(1::'a::\{recpower,ordered\text{-}semidom\}) < a$
shows $!!n.\ a\hat{}m \leq a\hat{}n ==> m \leq n$
⟨*proof*⟩

Surely we can strengthen this? It holds for $0<a<1$ too.

lemma *power-inject-exp* [*simp*]:
$1 < (a::'a::\{ordered\text{-}semidom,recpower\}) ==> (a\hat{}m = a\hat{}n) = (m=n)$
⟨*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*::{*recpower*,*ordered-semidom*}) < *a*; *a* ^ *m* < *a* ^ *n* |] ==> *m* < *n*
⟨*proof*⟩

lemma *power-mono*:
[|*a* ≤ *b*; (*0*::'*a*::{*recpower*,*ordered-semidom*}) ≤ *a*|] ==> *a* ^ *n* ≤ *b* ^ *n*
⟨*proof*⟩

lemma *power-strict-mono* [*rule-format*]:
[|*a* < *b*; (*0*::'*a*::{*recpower*,*ordered-semidom*}) ≤ *a*|]
==> *0* < *n* --> *a* ^ *n* < *b* ^ *n*
⟨*proof*⟩

lemma *power-eq-0-iff* [*simp*]:
(*a* ^ *n* = *0*) = (*a* = (*0*::'*a*::{*ring-1-no-zero-divisors*,*recpower*}) & *n*>*0*)
⟨*proof*⟩

lemma *field-power-not-zero*:
a ≠ (*0*::'*a*::{*ring-1-no-zero-divisors*,*recpower*}) ==> *a* ^ *n* ≠ *0*
⟨*proof*⟩

lemma *nonzero-power-inverse*:
fixes *a* :: '*a*::{*division-ring*,*recpower*}
shows *a* ≠ *0* ==> *inverse* (*a* ^ *n*) = (*inverse a*) ^ *n*
⟨*proof*⟩

Perhaps these should be simprules.

lemma *power-inverse*:
fixes *a* :: '*a*::{*division-ring*,*division-by-zero*,*recpower*}
shows *inverse* (*a* ^ *n*) = (*inverse a*) ^ *n*
⟨*proof*⟩

lemma *power-one-over*: *1* / (*a*::'*a*::{*field*,*division-by-zero*,*recpower*}) ^ *n* =
(*1* / *a*) ^ *n*
⟨*proof*⟩

lemma *nonzero-power-divide*:
b ≠ *0* ==> (*a*/*b*) ^ *n* = ((*a*::'*a*::{*field*,*recpower*}) ^ *n*) / (*b* ^ *n*)
⟨*proof*⟩

lemma *power-divide*:
(*a*/*b*) ^ *n* = ((*a*::'*a*::{*field*,*division-by-zero*,*recpower*}) ^ *n* / *b* ^ *n*)
⟨*proof*⟩

lemma *power-abs*: *abs*(*a* ^ *n*) = *abs*(*a*::'*a*::{*ordered-idom*,*recpower*}) ^ *n*
⟨*proof*⟩

lemma *zero-less-power-abs-iff* [*simp*,*noatp*]:
(*0* < (*abs a*) ^ *n*) = (*a* ≠ (*0*::'*a*::{*ordered-idom*,*recpower*}) | *n*=*0*)

⟨*proof*⟩

lemma *zero-le-power-abs* [*simp*]:
(0::'a::{*ordered-idom*,*recpower*}) ≤ (*abs a*) ^ *n*
⟨*proof*⟩

lemma *power-minus*: (−*a*) ^ *n* = (− 1) ^ *n* ∗ (*a*::'*a*::{*comm-ring-1*,*recpower*}) ^ *n*
⟨*proof*⟩

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:
[|(0::'a::{*ordered-semidom*,*recpower*}) < *a*; *a* < 1|]
==> *a* ∗ *a* ^ *n* < *a* ^ *n*
⟨*proof*⟩

lemma *power-strict-decreasing*:
[|*n* < *N*; 0 < *a*; *a* < (1::'*a*::{*ordered-semidom*,*recpower*})|]
==> *a* ^ *N* < *a* ^ *n*
⟨*proof*⟩

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing*:
[|*n* ≤ *N*; 0 ≤ *a*; *a* ≤ (1::'*a*::{*ordered-semidom*,*recpower*})|]
==> *a* ^ *N* ≤ *a* ^ *n*
⟨*proof*⟩

lemma *power-Suc-less-one*:
[| 0 < *a*; *a* < (1::'*a*::{*ordered-semidom*,*recpower*}) |] ==> *a* ^ *Suc n* < 1
⟨*proof*⟩

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing*:
[|*n* ≤ *N*; (1::'*a*::{*ordered-semidom*,*recpower*}) ≤ *a*|] ==> *a* ^ *n* ≤ *a* ^ *N*
⟨*proof*⟩

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
(1::'*a*::{*ordered-semidom*,*recpower*}) < *a* ==> *a* ^ *n* < *a* ∗ *a* ^ *n*
⟨*proof*⟩

lemma *power-strict-increasing*:
[|*n* < *N*; (1::'*a*::{*ordered-semidom*,*recpower*}) < *a*|] ==> *a* ^ *n* < *a* ^ *N*
⟨*proof*⟩

lemma *power-increasing-iff* [*simp*]:
1 < (*b*::'*a*::{*ordered-semidom*,*recpower*}) ==> (*b* ^ *x* ≤ *b* ^ *y*) = (*x* ≤ *y*)
⟨*proof*⟩

lemma *power-strict-increasing-iff* [*simp*]:
1 < (b::'a::{ordered-semidom,recpower}) ==> (b ^ x < b ^ y) = (x < y)
⟨*proof*⟩

lemma *power-le-imp-le-base*:
assumes *le*: *a ^ Suc n ≤ b ^ Suc n*
and *ynonneg*: *(0::'a::{ordered-semidom,recpower}) ≤ b*
shows *a ≤ b*
⟨*proof*⟩

lemma *power-less-imp-less-base*:
fixes *a b :: 'a::{ordered-semidom,recpower}*
assumes *less*: *a ^ n < b ^ n*
assumes *nonneg*: *0 ≤ b*
shows *a < b*
⟨*proof*⟩

lemma *power-inject-base*:
[| a ^ Suc n = b ^ Suc n; 0 ≤ a; 0 ≤ b |]
==> a = (b::'a::{ordered-semidom,recpower})
⟨*proof*⟩

lemma *power-eq-imp-eq-base*:
fixes *a b :: 'a::{ordered-semidom,recpower}*
shows *⟦a ^ n = b ^ n; 0 ≤ a; 0 ≤ b; 0 < n⟧ ⟹ a = b*
⟨*proof*⟩

15.2 Exponentiation for the Natural Numbers

instantiation *nat* :: *recpower*
begin

primrec *power-nat* **where**
p ^ 0 = (1::nat)
*| p ^ (Suc n) = (p::nat) * (p ^ n)*

instance ⟨*proof*⟩

end

lemma *of-nat-power*:
of-nat (m ^ n) = (of-nat m::'a::{semiring-1,recpower}) ^ n
⟨*proof*⟩

lemma *nat-one-le-power* [*simp*]: *1 ≤ i ==> Suc 0 ≤ i^n*
⟨*proof*⟩

lemma *nat-zero-less-power-iff* [*simp*]: *(x^n > 0) = (x > (0::nat) | n=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*: $i\hat{}m < i\hat{}n$
shows $m < n$
⟨*proof*⟩

lemma *power-diff*:
assumes *nz*: $a \sim= 0$
shows $n <= m ==> (a::'a::\{recpower, field\})\ \hat{}\ (m-n) = (a\hat{}m)\ /\ (a\hat{}n)$
⟨*proof*⟩

ML bindings for the general exponentiation theorems

⟨*ML*⟩

ML bindings for the remaining theorems

⟨*ML*⟩

end

16 Divides: The division operators div, mod and the divides relation dvd

theory *Divides*
imports *Nat Power Product-Type*
uses ~~/*src/Provers/Arith/cancel-div-mod.ML*
begin

16.1 Syntactic division operations

class *div* = *times* +
fixes *div* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *div 70*)
fixes *mod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *mod 70*)
begin

definition
dvd :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** *dvd 50*)
where
[*code func del*]: $m\ dvd\ n \longleftrightarrow (\exists k.\ n = m * k)$

end

16.2 Abstract divisibility in commutative semirings.

class *semiring-div* = *comm-semiring-1-cancel* + *div* +

assumes *mod-div-equality*: $a\ div\ b * b + a\ mod\ b = a$
and *div-by-0*: $a\ div\ 0 = 0$
and *mult-div*: $b \neq 0 \Longrightarrow a * b\ div\ b = a$
begin

op div and *op mod*

lemma *div-by-1*: $a\ div\ 1 = a$
⟨*proof*⟩

lemma *mod-by-1*: $a\ mod\ 1 = 0$
⟨*proof*⟩

lemma *mod-by-0*: $a\ mod\ 0 = a$
⟨*proof*⟩

lemma *mult-mod*: $a * b\ mod\ b = 0$
⟨*proof*⟩

lemma *mod-self*: $a\ mod\ a = 0$
⟨*proof*⟩

lemma *div-self*: $a \neq 0 \Longrightarrow a\ div\ a = 1$
⟨*proof*⟩

lemma *div-0*: $0\ div\ a = 0$
⟨*proof*⟩

lemma *mod-0*: $0\ mod\ a = 0$
⟨*proof*⟩

lemma *mod-div-equality2*: $b * (a\ div\ b) + a\ mod\ b = a$
⟨*proof*⟩

lemma *div-mod-equality*: $((a\ div\ b) * b + a\ mod\ b) + c = a + c$
⟨*proof*⟩

lemma *div-mod-equality2*: $(b * (a\ div\ b) + a\ mod\ b) + c = a + c$
⟨*proof*⟩

The *op dvd* relation

lemma *dvdI* [*intro?*]: $a = b * c \Longrightarrow b\ dvd\ a$
⟨*proof*⟩

lemma *dvdE* [*elim?*]: $b\ dvd\ a \Longrightarrow (\bigwedge c.\ a = b * c \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *dvd-def-mod* [*code func*]: $a\ dvd\ b \longleftrightarrow b\ mod\ a = 0$
⟨*proof*⟩

lemma *dvd-refl*: *a dvd a*
⟨*proof*⟩

lemma *dvd-trans*:
assumes *a dvd b* **and** *b dvd c*
shows *a dvd c*
⟨*proof*⟩

lemma *zero-dvd-iff* [*noatp*]: *0 dvd a* ⟷ *a = 0*
⟨*proof*⟩

lemma *dvd-0*: *a dvd 0*
⟨*proof*⟩

lemma *one-dvd*: *1 dvd a*
⟨*proof*⟩

lemma *dvd-mult*: *a dvd c* ⟹ *a dvd* (*b* ∗ *c*)
⟨*proof*⟩

lemma *dvd-mult2*: *a dvd b* ⟹ *a dvd* (*b* ∗ *c*)
⟨*proof*⟩

lemma *dvd-triv-right*: *a dvd b* ∗ *a*
⟨*proof*⟩

lemma *dvd-triv-left*: *a dvd a* ∗ *b*
⟨*proof*⟩

lemma *mult-dvd-mono*: *a dvd c* ⟹ *b dvd d* ⟹ *a* ∗ *b dvd c* ∗ *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*⟩

end

16.3 Division on *nat*

We define *op div* and *op mod* on *nat* by means of a characteristic relation with two input arguments *m*, *n* and two output arguments *q*(uotient) and *r*(emainder).

definition *divmod-rel* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
divmod-rel m n q r ⟷ *m = q* ∗ *n* + *r* ∧ (*if n* > *0 then 0* ≤ *r* ∧ *r* < *n else q* = *0*)

divmod-rel is total:

lemma *divmod-rel-ex*:
obtains $q\ r$ **where** *divmod-rel* $m\ n\ q\ r$
⟨*proof*⟩

divmod-rel is injective:

lemma *divmod-rel-unique-div*:
assumes *divmod-rel* $m\ n\ q\ r$
and *divmod-rel* $m\ n\ q'\ r'$
shows $q = q'$
⟨*proof*⟩

lemma *divmod-rel-unique-mod*:
assumes *divmod-rel* $m\ n\ q\ r$
and *divmod-rel* $m\ n\ q'\ r'$
shows $r = r'$
⟨*proof*⟩

We instantiate divisibility on the natural numbers by means of *divmod-rel*:

instantiation *nat* :: *semiring-div*
begin

definition *divmod* :: $nat \Rightarrow nat \Rightarrow nat \times nat$ **where**
[*code func del*]: $divmod\ m\ n = (THE\ (q,\ r).\ divmod\text{-}rel\ m\ n\ q\ r)$

definition *div-nat* **where**
$m\ div\ n = fst\ (divmod\ m\ n)$

definition *mod-nat* **where**
$m\ mod\ n = snd\ (divmod\ m\ n)$

lemma *divmod-div-mod*:
$divmod\ m\ n = (m\ div\ n,\ m\ mod\ n)$
⟨*proof*⟩

lemma *divmod-eq*:
assumes *divmod-rel* $m\ n\ q\ r$
shows $divmod\ m\ n = (q,\ r)$
⟨*proof*⟩

lemma *div-eq*:
assumes *divmod-rel* $m\ n\ q\ r$
shows $m\ div\ n = q$
⟨*proof*⟩

lemma *mod-eq*:
assumes *divmod-rel* $m\ n\ q\ r$
shows $m\ mod\ n = r$
⟨*proof*⟩

lemma *divmod-rel*: *divmod-rel m n (m div n) (m mod n)*
⟨*proof*⟩

lemma *divmod-zero*:
divmod m 0 = (0, m)
⟨*proof*⟩

lemma *divmod-base*:
assumes *m < n*
shows *divmod m n = (0, m)*
⟨*proof*⟩

lemma *divmod-step*:
assumes *0 < n* **and** *n ≤ m*
shows *divmod m n = (Suc ((m − n) div n), (m − n) mod n)*
⟨*proof*⟩

The "recursion" equations for *op div* and *op mod*

lemma *div-less* [*simp*]:
fixes *m n* :: *nat*
assumes *m < n*
shows *m div n = 0*
⟨*proof*⟩

lemma *le-div-geq*:
fixes *m n* :: *nat*
assumes *0 < n* **and** *n ≤ m*
shows *m div n = Suc ((m − n) div n)*
⟨*proof*⟩

lemma *mod-less* [*simp*]:
fixes *m n* :: *nat*
assumes *m < n*
shows *m mod n = m*
⟨*proof*⟩

lemma *le-mod-geq*:
fixes *m n* :: *nat*
assumes *n ≤ m*
shows *m mod n = (m − n) mod n*
⟨*proof*⟩

instance ⟨*proof*⟩

end

Simproc for cancelling *op div* and *op mod*

lemmas *mod-div-equality = semiring-div-class.times-div-mod-plus-zero-one.mod-div-equality*

[*of m*::*nat n*, *standard*]
lemmas *mod-div-equality2* = *mod-div-equality2* [*of n*::*nat m*, *standard*]
lemmas *div-mod-equality* = *div-mod-equality* [*of m*::*nat n k*, *standard*]
lemmas *div-mod-equality2* = *div-mod-equality2* [*of m*::*nat n k*, *standard*]

⟨*ML*⟩

code generator setup

lemma *divmod-if* [*code*]: *divmod m n* = (*if n* = *0* ∨ *m* < *n then* (*0*, *m*) *else*
let (*q*, *r*) = *divmod* (*m* − *n*) *n in* (*Suc q*, *r*))
⟨*proof*⟩

code-modulename *SML*
Divides Nat

code-modulename *OCaml*
Divides Nat

code-modulename *Haskell*
Divides Nat

16.3.1 Quotient

lemmas *DIVISION-BY-ZERO-DIV* [*simp*] = *div-by-0* [*of a*::*nat*, *standard*]
lemmas *div-0* [*simp*] = *semiring-div-class.div-0* [*of n*::*nat*, *standard*]

lemma *div-geq*: *0* < *n* ⟹ ¬ *m* < *n* ⟹ *m div n* = *Suc* ((*m* − *n*) *div n*)
⟨*proof*⟩

lemma *div-if*: *0* < *n* ⟹ *m div n* = (*if m* < *n then 0 else Suc* ((*m* − *n*) *div n*))
⟨*proof*⟩

lemma *div-mult-self-is-m* [*simp*]: *0*<*n* ==> (*m*∗*n*) *div n* = (*m*::*nat*)
⟨*proof*⟩

lemma *div-mult-self1-is-m* [*simp*]: *0*<*n* ==> (*n*∗*m*) *div n* = (*m*::*nat*)
⟨*proof*⟩

16.3.2 Remainder

lemmas *DIVISION-BY-ZERO-MOD* [*simp*] = *mod-by-0* [*of a*::*nat*, *standard*]
lemmas *mod-0* [*simp*] = *semiring-div-class.mod-0* [*of n*::*nat*, *standard*]

lemma *mod-less-divisor* [*simp*]:
fixes *m n* :: *nat*
assumes *n* > *0*
shows *m mod n* < (*n*::*nat*)
⟨*proof*⟩

lemma *mod-less-eq-dividend* [*simp*]:

fixes $m\ n :: nat$
shows $m\ mod\ n \leq m$
⟨*proof*⟩

lemma *mod-geq*: $\neg\ m < (n::nat) \Longrightarrow m\ mod\ n = (m - n)\ mod\ n$
⟨*proof*⟩

lemma *mod-if*: $m\ mod\ (n::nat) = (if\ m < n\ then\ m\ else\ (m - n)\ mod\ n)$
⟨*proof*⟩

lemma *mod-1* [*simp*]: $m\ mod\ Suc\ 0 = 0$
⟨*proof*⟩

lemmas *mod-self* [*simp*] = *semiring-div-class.mod-self* [*of* $n::nat$, *standard*]

lemma *mod-add-self2* [*simp*]: $(m+n)\ mod\ n = m\ mod\ (n::nat)$
⟨*proof*⟩

lemma *mod-add-self1* [*simp*]: $(n+m)\ mod\ n = m\ mod\ (n::nat)$
⟨*proof*⟩

lemma *mod-mult-self1* [*simp*]: $(m + k*n)\ mod\ n = m\ mod\ (n::nat)$
⟨*proof*⟩

lemma *mod-mult-self2* [*simp*]: $(m + n*k)\ mod\ n = m\ mod\ (n::nat)$
⟨*proof*⟩

lemma *mod-mult-distrib*: $(m\ mod\ n) * (k::nat) = (m * k)\ mod\ (n * k)$
⟨*proof*⟩

lemma *mod-mult-distrib2*: $(k::nat) * (m\ mod\ n) = (k*m)\ mod\ (k*n)$
⟨*proof*⟩

lemma *mod-mult-self-is-0* [*simp*]: $(m*n)\ mod\ n = (0::nat)$
⟨*proof*⟩

lemma *mod-mult-self1-is-0* [*simp*]: $(n*m)\ mod\ n = (0::nat)$
⟨*proof*⟩

lemma *mult-div-cancel*: $(n::nat) * (m\ div\ n) = m - (m\ mod\ n)$
⟨*proof*⟩

lemma *mod-le-divisor*[*simp*]: $0 < n \Longrightarrow m\ mod\ n \leq (n::nat)$
⟨*proof*⟩

16.3.3 Quotient and Remainder

lemma *mod-div-decomp*:

fixes $n\ k :: nat$
obtains $m\ q$ **where** $m = n\ div\ k$ **and** $q = n\ mod\ k$
and $n = m * k + q$
⟨*proof*⟩

lemma *divmod-rel-mult1-eq*:
$[|\ divmod\text{-}rel\ b\ c\ q\ r;\ c > 0\ |]$
$==>\ divmod\text{-}rel\ (a*b)\ c\ (a*q + a*r\ div\ c)\ (a*r\ mod\ c)$
⟨*proof*⟩

lemma *div-mult1-eq*: $(a*b)\ div\ c = a*(b\ div\ c) + a*(b\ mod\ c)\ div\ (c::nat)$
⟨*proof*⟩

lemma *mod-mult1-eq*: $(a*b)\ mod\ c = a*(b\ mod\ c)\ mod\ (c::nat)$
⟨*proof*⟩

lemma *mod-mult1-eq'*: $(a*b)\ mod\ (c::nat) = ((a\ mod\ c) * b)\ mod\ c$
⟨*proof*⟩

lemma *mod-mult-distrib-mod*:
$(a*b)\ mod\ (c::nat) = ((a\ mod\ c) * (b\ mod\ c))\ mod\ c$
⟨*proof*⟩

lemma *divmod-rel-add1-eq*:
$[|\ divmod\text{-}rel\ a\ c\ aq\ ar;\ divmod\text{-}rel\ b\ c\ bq\ br;\ \ c > 0\ |]$
$==>\ divmod\text{-}rel\ (a + b)\ c\ (aq + bq + (ar+br)\ div\ c)\ ((ar + br)\ mod\ c)$
⟨*proof*⟩

lemma *div-add1-eq*:
$(a+b)\ div\ (c::nat) = a\ div\ c + b\ div\ c + ((a\ mod\ c + b\ mod\ c)\ div\ c)$
⟨*proof*⟩

lemma *mod-add1-eq*: $(a+b)\ mod\ (c::nat) = (a\ mod\ c + b\ mod\ c)\ mod\ c$
⟨*proof*⟩

lemma *mod-lemma*: $[|\ (0::nat) < c;\ r < b\ |] ==> b * (q\ mod\ c) + r < b * c$
⟨*proof*⟩

lemma *divmod-rel-mult2-eq*: $[|\ divmod\text{-}rel\ a\ b\ q\ r;\ \ 0 < b;\ \ 0 < c\ |]$
$==>\ divmod\text{-}rel\ a\ (b*c)\ (q\ div\ c)\ (b*(q\ mod\ c) + r)$
⟨*proof*⟩

lemma *div-mult2-eq*: $a\ div\ (b*c) = (a\ div\ b)\ div\ (c::nat)$
⟨*proof*⟩

lemma *mod-mult2-eq*: $a\ mod\ (b*c) = b*(a\ div\ b\ mod\ c) + a\ mod\ (b::nat)$
⟨*proof*⟩

16.3.4 Cancellation of Common Factors in Division

lemma *div-mult-mult-lemma*:
[| (0::*nat*) < *b*; 0 < *c* |] ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
⟨*proof*⟩

lemma *div-mult-mult1* [*simp*]: (0::*nat*) < *c* ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
⟨*proof*⟩

lemma *div-mult-mult2* [*simp*]: (0::*nat*) < *c* ==> (*a*∗*c*) *div* (*b*∗*c*) = *a div b*
⟨*proof*⟩

16.3.5 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m div Suc* 0 = *m*
⟨*proof*⟩

lemmas *div-self* [*simp*] = *semiring-div-class.div-self* [*of n*::*nat*, *standard*]

lemma *div-add-self2*: 0<*n* ==> (*m*+*n*) *div n* = *Suc* (*m div n*)
⟨*proof*⟩

lemma *div-add-self1*: 0<*n* ==> (*n*+*m*) *div n* = *Suc* (*m div n*)
⟨*proof*⟩

lemma *div-mult-self1* [*simp*]: !!*n*::*nat*. 0<*n* ==> (*m* + *k*∗*n*) *div n* = *k* + *m div n*
⟨*proof*⟩

lemma *div-mult-self2* [*simp*]: 0<*n* ==> (*m* + *n*∗*k*) *div n* = *k* + *m div* (*n*::*nat*)
⟨*proof*⟩

lemma *div-le-mono* [*rule-format* (*no-asm*)]:
∀ *m*::*nat*. *m* ≤ *n* −−> (*m div k*) ≤ (*n div k*)
⟨*proof*⟩

lemma *div-le-mono2*: !!*m*::*nat*. [| 0<*m*; *m*≤*n* |] ==> (*k div n*) ≤ (*k div m*)
⟨*proof*⟩

lemma *div-le-dividend* [*simp*]: *m div n* ≤ (*m*::*nat*)
⟨*proof*⟩

lemma *div-less-dividend* [*rule-format*]:
!!*n*::*nat*. 1<*n* ==> 0 < *m* −−> *m div n* < *m*
⟨*proof*⟩

declare *div-less-dividend* [*simp*]

A fact for the mutilated chess board

lemma *mod-Suc*: *Suc*(*m*) *mod n* = (*if Suc*(*m mod n*) = *n then 0 else Suc*(*m mod n*))
⟨*proof*⟩

lemma *nat-mod-div-trivial* [*simp*]: *m mod n div n* = (*0* :: *nat*)
⟨*proof*⟩

lemma *nat-mod-mod-trivial* [*simp*]: *m mod n mod n* = (*m mod n* :: *nat*)
⟨*proof*⟩

16.3.6 The Divides Relation

lemma *dvdI* [*intro?*]: *n* = *m* ∗ *k* ==> *m dvd n*
⟨*proof*⟩

lemma *dvdE* [*elim?*]: !!*P*. [|*m dvd n*; !!*k*. *n* = *m*∗*k* ==> *P*|] ==> *P*
⟨*proof*⟩

lemma *dvd-0-right* [*iff*]: *m dvd* (*0*::*nat*)
⟨*proof*⟩

lemma *dvd-0-left*: *0 dvd m* ==> *m* = (*0*::*nat*)
⟨*proof*⟩

lemma *dvd-0-left-iff* [*iff*]: (*0 dvd* (*m*::*nat*)) = (*m* = *0*)
⟨*proof*⟩

declare *dvd-0-left-iff* [*noatp*]

lemma *dvd-1-left* [*iff*]: *Suc 0 dvd k*
⟨*proof*⟩

lemma *dvd-1-iff-1* [*simp*]: (*m dvd Suc 0*) = (*m* = *Suc 0*)
⟨*proof*⟩

lemmas *dvd-refl* [*simp*] = *semiring-div-class.dvd-refl* [*of m*::*nat*, *standard*]
lemmas *dvd-trans* [*trans*] = *semiring-div-class.dvd-trans* [*of m*::*nat n p*, *standard*]

lemma *dvd-anti-sym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
⟨*proof*⟩

op dvd is a partial order

interpretation *dvd*: *order* [*op dvd* λ*n m* :: *nat*. *n dvd m* ∧ *n* ≠ *m*]
⟨*proof*⟩

lemma *dvd-add*: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m*+*n* :: *nat*)

⟨*proof*⟩

lemma *dvd-diff*: [| *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-mult*: *k dvd n* ==> *k dvd* (*m*∗*n* :: *nat*)
⟨*proof*⟩

lemma *dvd-mult2*: *k dvd m* ==> *k dvd* (*m*∗*n* :: *nat*)
⟨*proof*⟩

lemma *dvd-triv-right* [*iff*]: *k dvd* (*m*∗*k* :: *nat*)
⟨*proof*⟩

lemma *dvd-triv-left* [*iff*]: *k dvd* (*k*∗*m* :: *nat*)
⟨*proof*⟩

lemma *dvd-reduce*: (*k dvd n* + *k*) = (*k dvd* (*n*::*nat*))
⟨*proof*⟩

lemma *dvd-mod*: !!*n*::*nat*. [| *f dvd m*; *f dvd n* |] ==> *f dvd m mod n*
⟨*proof*⟩

lemma *dvd-mod-imp-dvd*: [| (*k*::*nat*) *dvd m mod n*; *k dvd n* |] ==> *k dvd m*
⟨*proof*⟩

lemma *dvd-mod-iff*: *k dvd n* ==> ((*k*::*nat*) *dvd m mod n*) = (*k dvd m*)
⟨*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 *mult-dvd-mono*: [| *i dvd m*; *j dvd n*|] ==> *i*∗*j dvd* (*m*∗*n* :: *nat*)
⟨*proof*⟩

lemma *dvd-mult-left*: (*i*∗*j* :: *nat*) *dvd k* ==> *i dvd k*
⟨*proof*⟩

lemma *dvd-mult-right*: $(i*j :: nat)\ dvd\ k ==> j\ dvd\ k$
⟨*proof*⟩

lemma *dvd-imp-le*: $[|\ k\ dvd\ n;\ 0 < n\ |] ==> k \leq (n::nat)$
⟨*proof*⟩

lemmas *dvd-eq-mod-eq-0* = *dvd-def-mod* [*of k*::*nat n*, *standard*]

lemma *dvd-mult-div-cancel*: $n\ dvd\ m ==> n * (m\ div\ n) = (m::nat)$
⟨*proof*⟩

lemma *le-imp-power-dvd*: $!!i::nat.\ m \leq n ==> i\hat{}m\ dvd\ i\hat{}n$
⟨*proof*⟩

lemma *mod-add-left-eq*: $((a::nat) + b)\ mod\ c = (a\ mod\ c + b)\ mod\ c$
⟨*proof*⟩

lemma *mod-add-right-eq*: $(a+b)\ mod\ (c::nat) = (a + (b\ mod\ c))\ mod\ c$
⟨*proof*⟩

lemma *nat-zero-less-power-iff* [*simp*]: $(x\hat{}n > 0) = (x > (0::nat) \mid n=0)$
⟨*proof*⟩

lemma *power-le-dvd* [*rule-format*]: $k\hat{}j\ dvd\ n --> i \leq j --> k\hat{}i\ dvd\ (n::nat)$
⟨*proof*⟩

lemma *power-dvd-imp-le*: $[|i\hat{}m\ dvd\ i\hat{}n;\ (1::nat) < i|] ==> m \leq n$
⟨*proof*⟩

lemma *mod-eq-0-iff*: $(m\ mod\ d = 0) = (\exists\, q::nat.\ m = d*q)$
⟨*proof*⟩

lemmas *mod-eq-0D* [*dest*!] = *mod-eq-0-iff* [*THEN iffD1*]

lemma *mod-eqD*: $(m\ mod\ d = r) ==> \exists\, q::nat.\ m = r + q*d$
⟨*proof*⟩

lemma *split-div*:
$P(n\ div\ k :: nat) =$
$((k = 0 \longrightarrow P\ 0) \wedge (k \neq 0 \longrightarrow (!i.\ !j<k.\ n = k*i + j \longrightarrow P\ i)))$
(**is** *?P* = *?Q* **is** - = (- ∧ (- ⟶ *?R*)))
⟨*proof*⟩

lemma *split-div-lemma*:
assumes $0 < n$
shows $n * q \leq m \wedge m < n * Suc\ q \longleftrightarrow q = ((m::nat)\ div\ n)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

theorem *split-div'*:
$P\ ((m::nat)\ div\ n) = ((n = 0 \wedge P\ 0) \vee$
$(\exists q.\ (n * q \leq m \wedge m < n * (Suc\ q)) \wedge P\ q))$
⟨*proof*⟩

lemma *split-mod*:
$P(n\ mod\ k :: nat) =$
$((k = 0 \longrightarrow P\ n) \wedge (k \neq 0 \longrightarrow (!i.\ !j{<}k.\ n = k{*}i + j \longrightarrow P\ j)))$
(**is** *?P = ?Q* **is** - = (- ∧ (- ⟶ *?R*)))
⟨*proof*⟩

theorem *mod-div-equality'*: $(m::nat)\ mod\ n = m - (m\ div\ n) * n$
⟨*proof*⟩

lemma *div-mod-equality'*:
fixes *m n* :: *nat*
shows $m\ div\ n * n = m - m\ mod\ n$
⟨*proof*⟩

16.3.7 An "induction" law for modulus arithmetic.

lemma *mod-induct-0*:
assumes *step*: $\forall i{<}p.\ P\ i \longrightarrow P\ ((Suc\ i)\ mod\ p)$
and *base*: *P i* **and** *i*: $i{<}p$
shows *P 0*
⟨*proof*⟩

lemma *mod-induct*:
assumes *step*: $\forall i{<}p.\ P\ i \longrightarrow P\ ((Suc\ i)\ mod\ p)$
and *base*: *P i* **and** *i*: $i{<}p$ **and** *j*: $j{<}p$
shows *P j*
⟨*proof*⟩

end

17 Relation: Relations

theory *Relation*
imports *Product-Type*
begin

17.1 Definitions

definition
$converse :: ('a * 'b)\ set => ('b * 'a)\ set$
$((-\hat{}{-}1)\ [1000]\ 999)$ **where**
$r\hat{}{-}1 == \{(y,\ x).\ (x,\ y) : r\}$

notation (*xsymbols*)
converse ((-$^{-1}$) [*1000*] *999*)

definition
rel-comp :: [('*b* * '*c*) *set*, ('*a* * '*b*) *set*] => ('*a* * '*c*) *set*
(**infixr** *O 75*) **where**
r O s == {(*x*,*z*). *EX y*. (*x*, *y*) : *s* & (*y*, *z*) : *r*}

definition
Image :: [('*a* * '*b*) *set*, '*a set*] => '*b set*
(**infixl** `` *90*) **where**
r `` s == {*y*. *EX x*:*s*. (*x*,*y*):*r*}

definition
Id :: ('*a* * '*a*) *set* **where** — the identity relation
Id == {*p*. *EX x*. *p* = (*x*,*x*)}

definition
diag :: '*a set* => ('*a* * '*a*) *set* **where** — diagonal: identity over a set
diag A == $\bigcup x \in A$. {(*x*,*x*)}

definition
Domain :: ('*a* * '*b*) *set* => '*a set* **where**
Domain r == {*x*. *EX y*. (*x*,*y*):*r*}

definition
Range :: ('*a* * '*b*) *set* => '*b set* **where**
Range r == *Domain*(*r*^−*1*)

definition
Field :: ('*a* * '*a*) *set* => '*a set* **where**
Field r == *Domain r* ∪ *Range r*

definition
refl :: ['*a set*, ('*a* * '*a*) *set*] => *bool* **where** — reflexivity over a set
refl A r == *r* ⊆ *A* × *A* & (*ALL x*: *A*. (*x*,*x*) : *r*)

abbreviation
reflexive :: ('*a* * '*a*) *set* => *bool* **where** — reflexivity over a type
reflexive == *refl UNIV*

definition
sym :: ('*a* * '*a*) *set* => *bool* **where** — symmetry predicate
sym r == *ALL x y*. (*x*,*y*): *r* −−> (*y*,*x*): *r*

definition
antisym :: ('*a* * '*a*) *set* => *bool* **where** — antisymmetry predicate
antisym r == *ALL x y*. (*x*,*y*):*r* −−> (*y*,*x*):*r* −−> *x*=*y*

definition
trans :: $('a * 'a)$ *set* => *bool* **where** — transitivity predicate
trans r == $(ALL\ x\ y\ z.\ (x,y):r\ -->\ (y,z):r\ -->\ (x,z):r)$

definition
single-valued :: $('a * 'b)$ *set* => *bool* **where**
single-valued r == $ALL\ x\ y.\ (x,y):r\ -->\ (ALL\ z.\ (x,z):r\ -->\ y=z)$

definition
inv-image :: $('b * 'b)$ *set* => $('a => 'b)$ => $('a * 'a)$ *set* **where**
inv-image r f == $\{(x,\ y).\ (f\ x,\ f\ y) : r\}$

17.2 The identity relation

lemma *IdI* [*intro*]: $(a,\ a) : Id$
⟨*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 *reflexive-Id*: *reflexive 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*⟩

17.3 Diagonal: identity over a set

lemma *diag-empty* [*simp*]: *diag* {} = {}
⟨*proof*⟩

lemma *diag-eqI*: $a = b$ ==> $a : A$ ==> $(a,\ b) : diag\ A$
⟨*proof*⟩

lemma *diagI* [*intro*!,*noatp*]: $a : A$ ==> $(a,\ a) : diag\ A$
⟨*proof*⟩

lemma *diagE* [*elim*!]:
$c : diag\ A$ ==> $(!!x.\ x : A$ ==> $c = (x,\ x)$ ==> $P)$ ==> P

— The general elimination rule.
⟨*proof*⟩

lemma *diag-iff*: $((x,\ y) : diag\ A) = (x = y\ \&\ x : A)$
⟨*proof*⟩

lemma *diag-subset-Times*: $diag\ A \subseteq A \times A$
⟨*proof*⟩

17.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a,\ b) : s \Longrightarrow (b,\ c) : r \Longrightarrow (a,\ c) : r\ O\ s$
⟨*proof*⟩

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s \Longrightarrow$
$(!!x\ y\ z.\ xz = (x,\ z) \Longrightarrow (x,\ y) : s \Longrightarrow (y,\ z) : r\ \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-compEpair*:
$(a,\ c) : r\ O\ s \Longrightarrow (!!y.\ (a,\ y) : s \Longrightarrow (y,\ c) : r \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *R-O-Id* [*simp*]: $R\ O\ Id = R$
⟨*proof*⟩

lemma *Id-O-R* [*simp*]: $Id\ O\ R = R$
⟨*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*: $trans\ r \Longrightarrow r\ O\ r \subseteq r$
⟨*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*:
$s \subseteq A \times B \Longrightarrow r \subseteq B \times C \Longrightarrow (r\ O\ s) \subseteq A \times C$
⟨*proof*⟩

17.5 Reflexivity

lemma *reflI*: $r \subseteq A \times A$ ==> (!!x. $x : A$ ==> $(x, x) : r$) ==> *refl* A r
⟨*proof*⟩

lemma *reflD*: *refl* A r ==> $a : A$ ==> $(a, a) : r$
⟨*proof*⟩

lemma *reflD1*: *refl* A r ==> $(x, y) : r$ ==> $x : A$
⟨*proof*⟩

lemma *reflD2*: *refl* A r ==> $(x, y) : r$ ==> $y : A$
⟨*proof*⟩

lemma *refl-Int*: *refl* A r ==> *refl* B s ==> *refl* $(A \cap B)$ $(r \cap s)$
⟨*proof*⟩

lemma *refl-Un*: *refl* A r ==> *refl* B s ==> *refl* $(A \cup B)$ $(r \cup s)$
⟨*proof*⟩

lemma *refl-INTER*:
ALL x:S. *refl* $(A\ x)$ $(r\ x)$ ==> *refl* $(INTER\ S\ A)$ $(INTER\ S\ r)$
⟨*proof*⟩

lemma *refl-UNION*:
ALL x:S. *refl* $(A\ x)$ $(r\ x)$ ⟹ *refl* $(UNION\ S\ A)$ $(UNION\ S\ r)$
⟨*proof*⟩

lemma *refl-empty*[*simp*]: *refl* {} {}
⟨*proof*⟩

lemma *refl-diag*: *refl* A $(diag\ A)$
⟨*proof*⟩

17.6 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*: $r \subseteq s$ ==> *antisym* s ==> *antisym* r
⟨*proof*⟩

lemma *antisym-empty* [*simp*]: *antisym* {}
⟨*proof*⟩

lemma *antisym-diag* [*simp*]: *antisym* $(diag\ A)$

⟨*proof*⟩

17.7 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-diag* [*simp*]: *sym* (*diag A*)
⟨*proof*⟩

17.8 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* ∩ *s*)
⟨*proof*⟩

lemma *trans-INTER*: *ALL x*:*S*. *trans* (*r x*) ==> *trans* (*INTER S r*)
⟨*proof*⟩

lemma *trans-diag* [*simp*]: *trans* (*diag A*)
⟨*proof*⟩

17.9 Converse

lemma *converse-iff* [*iff*]: ((*a*,*b*): *r*^−1) = ((*b*,*a*) : *r*)
⟨*proof*⟩

lemma *converseI*[*sym*]: (*a*, *b*) : *r* ==> (*b*, *a*) : *r*^−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-diag* [*simp*]: $(diag\ A)\hat{}-1 = diag\ A$
⟨*proof*⟩

lemma *refl-converse* [*simp*]: $refl\ A\ (converse\ r) = refl\ A\ r$
⟨*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*⟩

17.10 Domain

declare *Domain-def* [*noatp*]

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-insert*: $Domain\ (insert\ (a,\ b)\ r) = insert\ a\ (Domain\ r)$
⟨*proof*⟩

lemma *Domain-Id* [*simp*]: $Domain\ Id = UNIV$
⟨*proof*⟩

lemma *Domain-diag* [*simp*]: $Domain\ (diag\ A) = A$
⟨*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*⟩

17.11 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-insert*: $Range\ (insert\ (a,\ b)\ r) = insert\ b\ (Range\ r)$
⟨*proof*⟩

lemma *Range-Id* [*simp*]: $Range\ Id = UNIV$
⟨*proof*⟩

lemma *Range-diag* [*simp*]: $Range\ (diag\ A) = A$
⟨*proof*⟩

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

lemma *Range-Int-subset*: $Range(A \cap B) \subseteq Range(A) \cap Range(B)$
⟨*proof*⟩

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

lemma *Range-Union*: $Range\ (Union\ S) = (\bigcup A{\in}S.\ Range\ A)$
⟨*proof*⟩

lemma *Range-converse*[*simp*]: $Range(r\hat{}{-}1) = Domain\ r$
⟨*proof*⟩

lemma *snd-eq-Range*: $snd\ `\ R = Range\ R$
⟨*proof*⟩

17.12 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*} ∪ *Field r*
⟨*proof*⟩

lemma *Field-Un*[*simp*]: *Field* (*r* ∪ *s*) = *Field r* ∪ *Field s*
⟨*proof*⟩

lemma *Field-Union*[*simp*]: *Field* (⋃ *R*) = ⋃ (*Field* ‘ *R*)
⟨*proof*⟩

lemma *Field-converse*[*simp*]: *Field*(*r*^−*1*) = *Field r*
⟨*proof*⟩

17.13 Image of a set under a relation

declare *Image-def* [*noatp*]

lemma *Image-iff*: (*b* : *r*‘‘*A*) = (*EX x*:*A*. (*x*, *b*) : *r*)
⟨*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*,*noatp*]: (*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-diag* [*simp*]: *diag 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 `` $(A \cap B) = R$ `` $A \cap R$ `` B
⟨*proof*⟩

lemma *Image-Un*: R `` $(A \cup B) = R$ `` $A \cup R$ `` B
⟨*proof*⟩

lemma *Un-Image*: $(R \cup S)$ `` $A = R$ `` $A \cup S$ `` A
⟨*proof*⟩

lemma *Image-subset*: $r \subseteq A \times B$ ==> $r``C \subseteq B$
⟨*proof*⟩

lemma *Image-eq-UN*: $r``B = (\bigcup y \in B.\ r``\{y\})$
— NOT suitable for rewriting
⟨*proof*⟩

lemma *Image-mono*: $r' \subseteq r$ ==> $A' \subseteq A$ ==> $(r'$ `` $A') \subseteq (r$ `` $A)$
⟨*proof*⟩

lemma *Image-UN*: $(r$ `` $(UNION\ A\ B)) = (\bigcup x \in A.\ r$ `` $(B\ x))$
⟨*proof*⟩

lemma *Image-INT-subset*: $(r$ `` $INTER\ A\ B) \subseteq (\bigcap x \in A.\ r$ `` $(B\ x))$
⟨*proof*⟩

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
$[|$*single-valued* (r^{-1}); $A \neq \{\}|]$ ==> r `` $INTER\ A\ B = (\bigcap x \in A.\ r$ `` $B\ x)$
⟨*proof*⟩

lemma *Image-subset-eq*: $(r``A \subseteq B) = (A \subseteq -((r\hat{}-1)$ `` $(-B)))$
⟨*proof*⟩

17.14 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*:
$r \subseteq s \Longrightarrow$ *single-valued* $s \Longrightarrow$ *single-valued* r
⟨*proof*⟩

lemma *single-valued-Id* [*simp*]: *single-valued Id*
⟨*proof*⟩

lemma *single-valued-diag* [*simp*]: *single-valued* (*diag* A)
⟨*proof*⟩

17.15 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*⟩

17.16 Inverse image

lemma *sym-inv-image*: *sym* $r \Longrightarrow$ *sym* (*inv-image* $r\ f$)
⟨*proof*⟩

lemma *trans-inv-image*: *trans* $r \Longrightarrow$ *trans* (*inv-image* $r\ f$)
⟨*proof*⟩

17.17 Version of *lfp-induct* for binary relations

lemmas *lfp-induct2* =
lfp-induct-set [*of* (a, b), *split-format* (*complete*)]

end

18 Predicate: Predicates

theory *Predicate*
imports *Inductive Relation*
begin

18.1 Equality and Subsets

lemma *pred-equals-eq*: $((\lambda x.\ x \in R) = (\lambda x.\ x \in S)) = (R = S)$
⟨*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*⟩

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*⟩

18.2 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*⟩

18.3 The empty set

lemma *bot-empty-eq*: *bot* $= (\lambda x.\ x \in \{\})$
⟨*proof*⟩

lemma *bot-empty-eq2*: *bot* $= (\lambda x\ y.\ (x,\ y) \in \{\})$
⟨*proof*⟩

18.4 Binary union

lemma *sup1-iff* [*simp*]: *sup A B x* $\longleftrightarrow$ *A x* | *B x*
⟨*proof*⟩

lemma *sup2-iff* [*simp*]: *sup A B x y* $\longleftrightarrow$ *A x y* | *B x y*
⟨*proof*⟩

lemma *sup-Un-eq* [*pred-set-conv*]: *sup* $(\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
⟨*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*⟩

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*⟩

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI* [*intro*!]: $(\sim B\ x \Longrightarrow A\ x) \Longrightarrow sup\ A\ B\ x$
⟨*proof*⟩

lemma *sup2CI* [*intro*!]: $(\sim B\ x\ y \Longrightarrow A\ x\ y) \Longrightarrow sup\ A\ B\ x\ y$
⟨*proof*⟩

lemma *sup1E* [*elim*!]: $sup\ A\ B\ x \Longrightarrow (A\ x \Longrightarrow P) \Longrightarrow (B\ x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *sup2E* [*elim*!]: $sup\ A\ B\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow P) \Longrightarrow (B\ x\ y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

18.5 Binary intersection

lemma *inf1-iff* [*simp*]: $inf\ A\ B\ x \longleftrightarrow A\ x \wedge B\ x$
⟨*proof*⟩

lemma *inf2-iff* [*simp*]: $inf\ A\ B\ x\ y \longleftrightarrow A\ x\ y \wedge B\ x\ y$
⟨*proof*⟩

lemma *inf-Int-eq* [*pred-set-conv*]: $inf\ (\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cap S)$
⟨*proof*⟩

lemma *inf-Int-eq2* [*pred-set-conv*]: $inf\ (\lambda x\ y.\ (x,\ y) \in R)\ (\lambda x\ y.\ (x,\ y) \in S) = (\lambda x\ y.\ (x,\ y) \in R \cap S)$
⟨*proof*⟩

lemma *inf1I* [*intro*!]: $A\ x \Longrightarrow B\ x \Longrightarrow inf\ A\ B\ x$
⟨*proof*⟩

lemma *inf2I* [*intro*!]: $A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow inf\ A\ B\ x\ y$
⟨*proof*⟩

lemma *inf1D1*: $inf\ A\ B\ x \Longrightarrow A\ x$
⟨*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 *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*⟩

18.6 Unions of families

lemma *SUP1-iff* [*simp*]: *(SUP x:A. B x) b = (EX x:A. B x b)*
⟨*proof*⟩

lemma *SUP2-iff* [*simp*]: *(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. (λx. x ∈ r i)) = (λx. x ∈ (UN i. r i))*
⟨*proof*⟩

lemma *SUP-UN-eq2*: *(SUP i. (λx y. (x, y) ∈ r i)) = (λx y. (x, y) ∈ (UN i. r i))*
⟨*proof*⟩

18.7 Intersections of families

lemma *INF1-iff* [*simp*]: *(INF x:A. B x) b = (ALL x:A. B x b)*
⟨*proof*⟩

lemma *INF2-iff* [*simp*]: *(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*. (λ*x*. *x* ∈ *r i*)) = (λ*x*. *x* ∈ (*INT i*. *r i*))
⟨*proof*⟩

lemma *INF-INT-eq2*: (*INF i*. (λ*x y*. (*x*, *y*) ∈ *r i*)) = (λ*x y*. (*x*, *y*) ∈ (*INT i*. *r i*))
⟨*proof*⟩

18.8 Composition of two relations

inductive
pred-comp :: ['*b* => '*c* => *bool*, '*a* => '*b* => *bool*] => '*a* => '*c* => *bool*
(**infixr** *OO* 75)
for *r* :: '*b* => '*c* => *bool* **and** *s* :: '*a* => '*b* => *bool*
where
pred-compI [*intro*]: *s a b* ==> *r b c* ==> (*r OO s*) *a c*

inductive-cases *pred-compE* [*elim*!]: (*r OO s*) *a c*

lemma *pred-comp-rel-comp-eq* [*pred-set-conv*]:
((λ*x y*. (*x*, *y*) ∈ *r*) *OO* (λ*x y*. (*x*, *y*) ∈ *s*)) = (λ*x y*. (*x*, *y*) ∈ *r O s*)
⟨*proof*⟩

18.9 Converse

inductive
conversep :: ('*a* => '*b* => *bool*) => '*b* => '*a* => *bool*
((-^--1) [1000] 1000)
for *r* :: '*a* => '*b* => *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*⟩

18.10 Domain

inductive
DomainP :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$
for $r :: 'a \Rightarrow 'b \Rightarrow bool$
where
DomainPI [*intro*]: $r\ a\ b \Longrightarrow DomainP\ r\ a$

inductive-cases *DomainPE* [*elim*!]: $DomainP\ r\ a$

lemma *DomainP-Domain-eq* [*pred-set-conv*]: $DomainP\ (\lambda x\ y.\ (x,\ y) \in r) = (\lambda x.\ x \in Domain\ r)$
⟨*proof*⟩

18.11 Range

inductive
RangeP :: (*'a* => *'b* => *bool*) => *'b* => *bool*
for *r* :: *'a* => *'b* => *bool*
where
RangePI [*intro*]: *r a b* ==> *RangeP r b*

inductive-cases *RangePE* [*elim!*]: *RangeP r b*

lemma *RangeP-Range-eq* [*pred-set-conv*]: *RangeP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Range r*)
⟨*proof*⟩

18.12 Inverse image

definition
inv-imagep :: (*'b* => *'b* => *bool*) => (*'a* => *'b*) => *'a* => *'a* => *bool* **where**
inv-imagep r f == %*x y*. *r* (*f x*) (*f y*)

lemma [*pred-set-conv*]: *inv-imagep* (λ*x y*. (*x*, *y*) ∈ *r*) *f* = (λ*x y*. (*x*, *y*) ∈ *inv-image r f*)
⟨*proof*⟩

lemma *in-inv-imagep* [*simp*]: *inv-imagep r f x y* = *r* (*f x*) (*f y*)
⟨*proof*⟩

18.13 The Powerset operator

definition *Powp* :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *bool* **where**
Powp A == λ*B*. ∀ *x* ∈ *B*. *A x*

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp* (λ*x*. *x* ∈ *A*) = (λ*x*. *x* ∈ *Pow A*)
⟨*proof*⟩

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred pred-subset-eq*]

18.14 Properties of relations - predicate versions

abbreviation *antisymP* :: (*'a* => *'a* => *bool*) => *bool* **where**
antisymP r == *antisym* {(*x*, *y*). *r x y*}

abbreviation *transP* :: (*'a* => *'a* => *bool*) => *bool* **where**
transP r == *trans* {(*x*, *y*). *r x y*}

abbreviation *single-valuedP* :: (*'a* => *'b* => *bool*) => *bool* **where**
single-valuedP r == *single-valued* {(*x*, *y*). *r x y*}

end

19 Transitive-Closure: Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Predicate*
uses ~~/*src*/*Provers*/*trancl.ML*
begin

rtrancl is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

inductive-set
  rtrancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ ((-^*) [*1000*] *999*)
  for $r :: ('a \times 'a)\ set$
where
    rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: $(a, a) : r\hat{}*$
  | *rtrancl-into-rtrancl* [*Pure.intro*]: $(a, b) : r\hat{}* \Longrightarrow (b, c) : r \Longrightarrow (a, c) : r\hat{}*$

inductive-set
  trancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ ((-^+) [*1000*] *999*)
  for $r :: ('a \times 'a)\ set$
where
    r-into-trancl [*intro*, *Pure.intro*]: $(a, b) : r \Longrightarrow (a, b) : r\hat{}+$
  | *trancl-into-trancl* [*Pure.intro*]: $(a, b) : r\hat{}+ \Longrightarrow (b, c) : r \Longrightarrow (a, c) : r\hat{}+$

notation
  rtranclp ((-^**) [*1000*] *1000*) **and**
  tranclp ((-^++) [*1000*] *1000*)

abbreviation
  reflclp :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ ((-^==) [*1000*] *1000*)
where
  $r\hat{}== == sup\ r\ op =$

abbreviation
  reflcl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ ((-^=) [*1000*] *999*) **where**
  $r\hat{}= == r \cup Id$

notation (*xsymbols*)
  rtranclp ((-**) [*1000*] *1000*) **and**
  tranclp ((-$^{++}$) [*1000*] *1000*) **and**
  reflclp ((-$^{==}$) [*1000*] *1000*) **and**
  rtrancl ((-*) [*1000*] *999*) **and**
  trancl ((-$^{+}$) [*1000*] *999*) **and**
  reflcl ((-$^{=}$) [*1000*] *999*)

notation (*HTML* **output**)

rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-$^{++}$) [*1000*] *1000*) **and**
reflclp ((-$^{==}$) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-$^{+}$) [*1000*] *999*) **and**
reflcl ((-$^{=}$) [*1000*] *999*)

19.1 Reflexive closure

lemma *reflexive-reflcl*[*simp*]: *reflexive*($r\hat{}=$)
⟨*proof*⟩

lemma *antisym-reflcl*[*simp*]: *antisym*($r\hat{}=$) = *antisym r*
⟨*proof*⟩

lemma *trans-reflclI*[*simp*]: *trans r* ⟹ *trans*($r\hat{}=$)
⟨*proof*⟩

19.2 Reflexive-transitive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: (*sup* (λ*x y*. (*x*, *y*) ∈ *r*) *op* =) = (λ*x y*. (*x*, *y*) ∈ *r Un Id*)
⟨*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 *reflexive-rtrancl*: *reflexive* ($r\hat{}*$)
⟨*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*: [| $Id \subseteq s$; $r\ O\ (r\hat{}* \cap s) \subseteq s$|] ==> $r\hat{}* \subseteq s$
⟨*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* ^**) (*S* ^**)) ^** = (*sup R S*) ^**
⟨*proof*⟩

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflcl* [*simp*]: (*R* ^==) ^** = *R* ^**
⟨*proof*⟩

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: (*r* − *Id*) ^* = *r* ^*
⟨*proof*⟩

lemma *rtranclp-r-diff-Id*: (*inf r op* ~=) ^** = *r* ^**
⟨*proof*⟩

theorem *rtranclp-converseD*:
assumes *r*: (*r* ^−−1) ^** *x y*
shows *r* ^** *y x*
⟨*proof*⟩

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

theorem *rtranclp-converseI*:
assumes *r* ^** *y x*
shows (*r* ^−−1) ^** *x y*
⟨*proof*⟩

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

lemma *rtrancl-converse*: (*r* ^−1) ^* = (*r* ^*) ^−1
⟨*proof*⟩

lemma *sym-rtrancl*: *sym r* ==> *sym* (*r* ^*)
⟨*proof*⟩

theorem *converse-rtranclp-induct*[*consumes 1*]:
assumes *major*: *r* ^** *a b*
and *cases*: *P b* !!*y z*. [| *r y z*; *r* ^** *z b*; *P z* |] ==> *P y*
shows *P a*
⟨*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*:
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*: $r\hat{}* = Id\ Un\ r\ O\ r\hat{}*$
⟨*proof*⟩

19.3 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 $r^{++}\ a\ b$
  and *cases*: $!!y.\ r\ a\ y \Longrightarrow P\ y$
    $!!y\ z.\ r^{++}\ a\ y \Longrightarrow r\ y\ z \Longrightarrow P\ y \Longrightarrow P\ z$
  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*: $r^{++}\ x\ y$
    and *cases*: $!!x\ y.\ r\ x\ y \Longrightarrow P\ x\ y$
      $!!x\ y\ z.\ [|\ r^{++}\ x\ y;\ P\ x\ y;\ r^{++}\ y\ z;\ P\ y\ z\ |] \Longrightarrow P\ x\ z$
  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 $(a,\ b) : r^{+}$
  obtains
    (*base*) $(a,\ b) : r$
  | (*step*) c **where** $(a,\ c) : r^{+}$ **and** $(c,\ b) : r$
  ⟨*proof*⟩

lemma *trancl-Int-subset*: $[|\ r \subseteq s;\ r\ O\ (r^{+} \cap s) \subseteq s|] \Longrightarrow r^{+} \subseteq s$
  ⟨*proof*⟩

lemma *trancl-unfold*: $r^{+} = r\ Un\ r\ O\ r^{+}$
  ⟨*proof*⟩

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* (r^{+})
⟨*proof*⟩

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*, *standard*]

lemma *tranclp-trans*:
  assumes *xy*: $r^{++}\ x\ y$

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*:
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 *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*: $R\hat{}{*}{*}\ a\ b \Longrightarrow a = b \vee a \neq b \wedge R\hat{}{+}{+}\ a\ b$
⟨*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*⟩

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*⟩

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
⟦ *single-valued* r; $(x,y) \in r\hat{}*$; $(x,z) \in r\hat{}*$ ⟧
$\Longrightarrow (y,z) \in r\hat{}* \vee (z,y) \in r\hat{}*$
⟨*proof*⟩

lemma *r-r-into-trancl*: $(a,\ b) \in R ==> (b,\ c) \in R ==> (a,\ c) \in R\hat{}+$
⟨*proof*⟩

lemma *trancl-into-trancl* [*rule-format*]:
$(a,\ b) \in r^{+} ==> (b,\ c) \in r --> (a,c) \in r^{+}$
⟨*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*]

19.4 Setup of transitivity reasoner

⟨*ML*⟩

end

20 Finite-Set: Finite sets

theory *Finite-Set*
imports *Divides Transitive-Closure*
begin

20.1 Definition and basic properties

inductive *finite* :: *'a set* => *bool*
  where
    emptyI [*simp*, *intro*!]: *finite* {}
  | *insertI* [*simp*, *intro*!]: *finite A* ==> *finite* (*insert a A*)

lemma *ex-new-if-finite*: — does not depend on def of finite at all
  assumes ¬ *finite* (*UNIV* :: *'a set*) **and** *finite A*
  shows ∃ *a*::*'a*. *a* ∉ *A*
⟨*proof*⟩

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* ⊆ *A*
    and *empty*: *P* {}
    and *insert*: !!*a F*. *finite F* ==> *a* ∈ *A* ==> *a* ∉ *F* ==> *P F* ==> *P* (*insert*
a F)
  shows *P F*
⟨*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*} ⟹ *finite A*
⟨*proof*⟩

lemma *finite-conv-nat-seg-image*:
finite A = (∃ (*n*::*nat*) *f*. *A* = *f* ‘ {*i*::*nat*. *i*<*n*})
⟨*proof*⟩

20.1.1 Finiteness and set theoretic constructions

lemma *finite-UnI*: *finite F* ==> *finite G* ==> *finite* (*F Un G*)
— The union of two finite sets is finite.
⟨*proof*⟩

lemma *finite-subset*: *A* ⊆ *B* ==> *finite B* ==> *finite A*
— Every subset of a finite set is finite.
⟨*proof*⟩

lemma *finite-Collect-subset*[*simp*]: *finite A* ⟹ *finite*{*x* ∈ *A*. *P x*}
⟨*proof*⟩

lemma *finite-Un* [*iff*]: *finite* (*F Un G*) = (*finite F* & *finite G*)
⟨*proof*⟩

lemma *finite-Int* [*simp*, *intro*]: *finite F* | *finite G* ==> *finite* (*F Int G*)
— The converse obviously fails.
⟨*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-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 B* ==> *finite* (*B* − *Ba*)

⟨*proof*⟩

lemma *finite-Diff-insert* [*iff*]: *finite* (*A* − *insert a B*) = *finite* (*A* − *B*)
⟨*proof*⟩

lemma *finite-Diff-singleton* [*simp*]: *finite* (*A* − {*a*}) = *finite A*
⟨*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-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*⟩

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-Plus*: [| *finite A*; *finite B* |] ==> *finite* (*A* <+> *B*)
⟨*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-UnionD*: *finite*(⋃ *A*) ⟹ *finite A*
⟨*proof*⟩

lemma *finite-converse* [*iff*]: *finite* (*r*^*−1*) = *finite r*
⟨*proof*⟩

Finiteness of transitive closure (Thanks to Sidi Ehmety)

lemma *finite-Field*: *finite r* ==> *finite* (*Field r*)
— A finite relation has a finite field (= *domain* ∪ *range*.
⟨*proof*⟩

lemma *trancl-subset-Field2*: *r*^+ <= *Field r* × *Field r*
⟨*proof*⟩

lemma *finite-trancl*: *finite* (*r*^+) = *finite r*
⟨*proof*⟩

20.2 Class *finite*

⟨*ML*⟩

class *finite* = *itself* +
  assumes *finite-UNIV*: *finite* (*UNIV* :: *'a set*)
⟨*ML*⟩
hide *const finite*

lemma *finite* [*simp*]: *finite* (*A* :: *'a*::*finite set*)
  ⟨*proof*⟩

lemma *UNIV-unit* [*noatp*]:
  UNIV = {()} ⟨*proof*⟩

instance *unit* :: *finite*
  ⟨*proof*⟩

lemma *UNIV-bool* [*noatp*]:
  UNIV = {*False*, *True*} ⟨*proof*⟩

instance *bool* :: *finite*
  ⟨*proof*⟩

instance ∗ :: (*finite*, *finite*) *finite*
  ⟨*proof*⟩

instance + :: (*finite*, *finite*) *finite*
  ⟨*proof*⟩

lemma *inj-graph*: *inj* (%*f*. {(*x*, *y*). *y* = *f x*})
  ⟨*proof*⟩

instance *fun* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

20.3 A fold functional for finite sets

The intended behaviour is *fold f g z* {x_1, ..., x_n} = *f* (*g* x_1) (... (*f* (*g* x_n) *z*)...) if *f* is associative-commutative. For an application of *fold* se the definitions of sums and products over finite sets.

inductive
  foldSet :: (*'a* => *'a* => *'a*) => (*'b* => *'a*) => *'a* => *'b set* => *'a* => *bool*
  for *f* :: *'a* => *'a* => *'a*
  and *g* :: *'b* => *'a*
  and *z* :: *'a*
where
  emptyI [*intro*]: *foldSet f g z* {} *z*
| *insertI* [*intro*]:
    ⟦ *x* ∉ *A*; *foldSet f g z A y* ⟧
    ⟹ *foldSet f g z* (*insert x A*) (*f* (*g x*) *y*)

inductive-cases *empty-foldSetE* [*elim*!]: *foldSet f g z* {} *x*

constdefs
fold :: ('*a* => '*a* => '*a*) => ('*b* => '*a*) => '*a* => '*b set* => '*a*
fold f g z A == *THE x. foldSet f g z A x*

A tempting alternative for the definiens is *if finite A then THE x. foldSet f g e A x else e*. It allows the removal of finiteness assumptions from the theorems *fold-commute*, *fold-reindex* and *fold-distrib*. The proofs become ugly, with *rule-format*. It is not worth the effort.

lemma *Diff1-foldSet*:
foldSet f g z (*A* − {*x*}) *y* ==> *x*: *A* ==> *foldSet f g z A* (*f* (*g x*) *y*)
⟨*proof*⟩

lemma *foldSet-imp-finite*: *foldSet f g z A x*==> *finite A*
⟨*proof*⟩

lemma *finite-imp-foldSet*: *finite A* ==> *EX x. foldSet f g z A x*
⟨*proof*⟩

20.3.1 From *foldSet* to *fold*

lemma *image-less-Suc*: *h* ‘ {*i*. *i* < *Suc m*} = *insert* (*h m*) (*h* ‘ {*i*. *i* < *m*})
⟨*proof*⟩

lemma *insert-image-inj-on-eq*:
[|*insert* (*h m*) *A* = *h* ‘ {*i*. *i* < *Suc m*}; *h m* ∉ *A*;
inj-on h {*i*. *i* < *Suc m*}|]
==> *A* = *h* ‘ {*i*. *i* < *m*}
⟨*proof*⟩

lemma *insert-inj-onE*:
assumes *aA*: *insert a A* = *h*‘{*i*::*nat*. *i*<*n*} **and** *anot*: *a* ∉ *A*
and *inj-on*: *inj-on h* {*i*::*nat*. *i*<*n*}
shows ∃*hm m*. *inj-on hm* {*i*::*nat*. *i*<*m*} & *A* = *hm* ‘ {*i*. *i*<*m*} & *m* < *n*
⟨*proof*⟩

context *ab-semigroup-mult*
begin

lemma *foldSet-determ-aux*:
!!*A x x*′ *h*. ⟦ *A* = *h*‘{*i*::*nat*. *i*<*n*}; *inj-on h* {*i*. *i*<*n*};
foldSet times g z A x; *foldSet times g z A x*′ ⟧
⟹ *x*′ = *x*
⟨*proof*⟩

lemma *foldSet-determ*:
foldSet times g z A x ==> *foldSet times g z A y* ==> *y* = *x*
⟨*proof*⟩

lemma *fold-equality*: *foldSet times g z A y ==> fold times g z A = y*
⟨*proof*⟩

The base case for *fold*:

lemma (**in** −) *fold-empty* [*simp*]: *fold f g z {} = z*
⟨*proof*⟩

lemma *fold-insert-aux*: *x ∉ A ==>*
(foldSet times g z (insert x A) v) =
(EX y. foldSet times g z A y & v = g x ∗ y)
⟨*proof*⟩

The recursion equation for *fold*:

lemma *fold-insert* [*simp*]:
finite A ==> x ∉ A ==> fold times g z (insert x A) = g x ∗ fold times g z A
⟨*proof*⟩

lemma *fold-rec*:
assumes *fin*: *finite A* **and** *a*: *a*:*A*
shows *fold times g z A = g a ∗ fold times g z (A − {a})*
⟨*proof*⟩

end

A simplified version for idempotent functions:

context *ab-semigroup-idem-mult*
begin

lemma *fold-insert-idem*:
assumes *finA*: *finite A*
shows *fold times g z (insert a A) = g a ∗ fold times g z A*
⟨*proof*⟩

lemma *foldI-conv-id*:
finite A ⟹ fold times g z A = fold times id z (g ‘ A)
⟨*proof*⟩

end

20.3.2 Lemmas about *fold*

context *ab-semigroup-mult*
begin

lemma *fold-commute*:
finite A ==> (!!z. x ∗ (fold times g z A) = fold times g (x ∗ z) A)
⟨*proof*⟩

lemma *fold-nest-Un-Int*:

finite A ==> finite B
==> fold times g (fold times g z B) A = fold times g (fold times g z (A Int B)) (A Un B)
⟨*proof*⟩

lemma *fold-nest-Un-disjoint*:
finite A ==> finite B ==> A Int B = {}
==> fold times g z (A Un B) = fold times g (fold times g z B) A
⟨*proof*⟩

lemma *fold-reindex*:
assumes *fin*: *finite A*
shows *inj-on h A ⟹ fold times g z (h ' A) = fold times (g ∘ h) z A*
⟨*proof*⟩

Fusion theorem, as described in Graham Hutton's paper, A Tutorial on the Universality and Expressiveness of Fold, JFP 9:4 (355-372), 1999.

lemma *fold-fusion*:
includes *ab-semigroup-mult g*
assumes *fin*: *finite A*
and *hyp*: ⋀*x y*. *h (g x y) = times x (h y)*
shows *h (fold g j w A) = fold times j (h w) A*
⟨*proof*⟩

lemma *fold-cong*:
finite A ⟹ (!!x. x:A ==> g x = h x) ==> fold times g z A = fold times h z A
⟨*proof*⟩

end

context *comm-monoid-mult*
begin

lemma *fold-Un-Int*:
finite A ==> finite B ==>
*fold times g 1 A * fold times g 1 B =*
*fold times g 1 (A Un B) * fold times g 1 (A Int B)*
⟨*proof*⟩

corollary *fold-Un-disjoint*:
finite A ==> finite B ==> A Int B = {} ==>
*fold times g 1 (A Un B) = fold times g 1 A * fold times g 1 B*
⟨*proof*⟩

lemma *fold-UN-disjoint*:
⟦ *finite I*; *ALL i:I. finite (A i)*;
ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {} ⟧
⟹ fold times g 1 (UNION I A) =
fold times (%i. fold times g 1 (A i)) 1 I

⟨*proof*⟩

lemma *fold-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
fold times (%x. fold times (g x) 1 (B x)) 1 A =
fold times (split g) 1 (SIGMA x:A. B x)
⟨*proof*⟩

lemma *fold-distrib*: *finite A ⟹*
*fold times (%x. g x * h x) 1 A = fold times g 1 A * fold times h 1 A*
⟨*proof*⟩

end

20.4 Generalized summation over a set

interpretation *comm-monoid-add*: *comm-monoid-mult* [*0*::'*a*::*comm-monoid-add op +*]
⟨*proof*⟩

constdefs
setsum :: ('*a* => '*b*) => '*a set* => '*b*::*comm-monoid-add*
setsum f A == *if finite A then fold (op +) f 0 A else 0*

abbreviation
Setsum (∑ - [*1000*] *999*) **where**
∑ *A* == *setsum* (%*x*. *x*) *A*

Now: lot's of fancy syntax. First, *setsum* (λ*x*. *e*) *A* is written ∑ *x*∈*A*. *e*.

syntax
-setsum :: *pttrn* => '*a set* => '*b* => '*b*::*comm-monoid-add* ((*3SUM* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setsum :: *pttrn* => '*a set* => '*b* => '*b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setsum :: *pttrn* => '*a set* => '*b* => '*b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM i:*A*. *b* == *setsum* (%*i*. *b*) *A*
∑ *i*∈*A*. *b* == *setsum* (%*i*. *b*) *A*

Instead of ∑ *x*∈{*x*. *P*}. *e* we introduce the shorter ∑ *x*|*P*. *e*.

syntax
-qsetsum :: *pttrn* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3SUM* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3*∑ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3*∑ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
SUM x|P. t => setsum (%x. t) {x. P}
$\sum x|P.\ t$ => *setsum (%x. t) {x. P}*

⟨*ML*⟩

lemma *setsum-empty* [*simp*]: *setsum f {} = 0*
⟨*proof*⟩

lemma *setsum-insert* [*simp*]:
finite F ==> a ∉ F ==> setsum f (insert a F) = f a + setsum f F
⟨*proof*⟩

lemma *setsum-infinite* [*simp*]: ~ *finite A ==> setsum f A = 0*
⟨*proof*⟩

lemma *setsum-reindex*:
inj-on f B ==> setsum h (f ' B) = setsum (h ∘ f) B
⟨*proof*⟩

lemma *setsum-reindex-id*:
inj-on f B ==> setsum f B = setsum id (f ' B)
⟨*proof*⟩

lemma *setsum-cong*:
A = B ==> (!!x. x:B ==> f x = g x) ==> setsum f A = setsum g B
⟨*proof*⟩

lemma *strong-setsum-cong*[*cong*]:
A = B ==> (!!x. x:B =simp=> f x = g x)
==> setsum (%x. f x) A = setsum (%x. g x) B
⟨*proof*⟩

lemma *setsum-cong2*: $[\![\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x]\!] \Longrightarrow$ *setsum f A = setsum g A*
⟨*proof*⟩

lemma *setsum-reindex-cong*:
[|inj-on f A; B = f ' A; !!a. a:A ⟹ g a = h (f a)|]
==> setsum h B = setsum g A
⟨*proof*⟩

lemma *setsum-0*[*simp*]: *setsum (%i. 0) A = 0*
⟨*proof*⟩

lemma *setsum-0'*: *ALL a:A. f a = 0 ==> setsum f A = 0*
⟨*proof*⟩

lemma *setsum-Un-Int*: *finite A ==> finite B ==>*
setsum g (A Un B) + setsum g (A Int B) = setsum g A + setsum g B
— The reversed orientation looks more natural, but LOOPS as a simprule!
⟨*proof*⟩

lemma *setsum-Un-disjoint*: *finite A ==> finite B*
==> A Int B = {} ==> setsum g (A Un B) = setsum g A + setsum g B
⟨*proof*⟩

lemma *setsum-UN-disjoint*:
finite I ==> (ALL i:I. finite (A i)) ==>
(ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {}) ==>
setsum f (UNION I A) = (∑ i∈I. setsum f (A i))
⟨*proof*⟩

No need to assume that C is finite. If infinite, the rhs is directly 0, and $\bigcup C$ is also infinite, hence the lhs is also 0.

lemma *setsum-Union-disjoint*:
[| (ALL A:C. finite A);
(ALL A:C. ALL B:C. A ≠ B --> A Int B = {}) |]
==> setsum f (Union C) = setsum (setsum f) C
⟨*proof*⟩

lemma *setsum-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
(∑ x∈A. (∑ y∈B x. f x y)) = (∑ (x,y)∈(SIGMA x:A. B x). f x y)
⟨*proof*⟩

Here we can eliminate the finiteness assumptions, by cases.

lemma *setsum-cartesian-product*:
(∑ x∈A. (∑ y∈B. f x y)) = (∑ (x,y) ∈ A <> B. f x y)*
⟨*proof*⟩

lemma *setsum-addf*: *setsum (%x. f x + g x) A = (setsum f A + setsum g A)*
⟨*proof*⟩

20.4.1 Properties in more restricted classes of structures

lemma *setsum-SucD*: *setsum f A = Suc n ==> EX a:A. 0 < f a*
⟨*proof*⟩

lemma *setsum-eq-0-iff* [*simp*]:
finite F ==> (setsum f F = 0) = (ALL a:F. f a = (0::nat))
⟨*proof*⟩

lemma *setsum-Un-nat*: *finite A ==> finite B ==>*
(setsum f (A Un B) :: nat) = setsum f A + setsum f B - setsum f (A Int B)
— For the natural numbers, we have subtraction.

⟨*proof*⟩

lemma *setsum-Un*: *finite A* ==> *finite B* ==>
(*setsum f* (*A Un B*) :: *'a* :: *ab-group-add*) =
setsum f A + *setsum f B* − *setsum f* (*A Int B*)
⟨*proof*⟩

lemma *setsum-diff1-nat*: (*setsum f* (*A* − {*a*}) :: *nat*) =
(*if a*:*A then setsum f A* − *f a else setsum f A*)
⟨*proof*⟩

lemma *setsum-diff1*: *finite A* ⟹
(*setsum f* (*A* − {*a*}) :: (*'a*::*ab-group-add*)) =
(*if a*:*A then setsum f A* − *f a else setsum f A*)
⟨*proof*⟩

lemma *setsum-diff1'*[*rule-format*]: *finite A* ⟹ *a* ∈ *A* ⟶ (∑ *x* ∈ *A*. *f x*) = *f a*
+ (∑ *x* ∈ (*A* − {*a*}). *f x*)
⟨*proof*⟩

lemma *setsum-diff-nat*:
assumes *finite B*
and *B* ⊆ *A*
shows (*setsum f* (*A* − *B*) :: *nat*) = (*setsum f A*) − (*setsum f B*)
⟨*proof*⟩

lemma *setsum-diff*:
assumes *le*: *finite A B* ⊆ *A*
shows *setsum f* (*A* − *B*) = *setsum f A* − ((*setsum f B*)::(*'a*::*ab-group-add*))
⟨*proof*⟩

lemma *setsum-mono*:
assumes *le*: ⋀*i*. *i*∈*K* ⟹ *f* (*i*::*'a*) ≤ ((*g i*)::(*'b*::{*comm-monoid-add*, *pordered-ab-semigroup-add*}))
shows (∑ *i*∈*K*. *f i*) ≤ (∑ *i*∈*K*. *g i*)
⟨*proof*⟩

lemma *setsum-strict-mono*:
fixes *f* :: *'a* ⇒ *'b*::{*pordered-cancel-ab-semigroup-add*,*comm-monoid-add*}
assumes *finite A* *A* ≠ {}
and !!*x*. *x*:*A* ⟹ *f x* < *g x*
shows *setsum f A* < *setsum g A*
⟨*proof*⟩

lemma *setsum-negf*:
setsum (%*x*. − (*f x*)::*'a*::*ab-group-add*) *A* = − *setsum f A*
⟨*proof*⟩

lemma *setsum-subtractf*:
setsum (%*x*. ((*f x*)::'*a*::*ab-group-add*) − *g x*) *A* =
setsum f A − *setsum g A*
⟨*proof*⟩

lemma *setsum-nonneg*:
assumes *nn*: ∀ *x*∈*A*. (*0*::'*a*::{*pordered-ab-semigroup-add*,*comm-monoid-add*}) ≤ *f x*
shows *0* ≤ *setsum f A*
⟨*proof*⟩

lemma *setsum-nonpos*:
assumes *np*: ∀ *x*∈*A*. *f x* ≤ (*0*::'*a*::{*pordered-ab-semigroup-add*,*comm-monoid-add*})
shows *setsum f A* ≤ *0*
⟨*proof*⟩

lemma *setsum-mono2*:
fixes *f* :: '*a* ⇒ '*b* :: {*pordered-ab-semigroup-add-imp-le*,*comm-monoid-add*}
assumes *fin*: *finite B* **and** *sub*: *A* ⊆ *B* **and** *nn*: ⋀*b*. *b* ∈ *B*−*A* ⟹ *0* ≤ *f b*
shows *setsum f A* ≤ *setsum f B*
⟨*proof*⟩

lemma *setsum-mono3*: *finite B* ==> *A* <= *B* ==>
ALL x: *B* − *A*.
0 <= ((*f x*)::'*a*::{*comm-monoid-add*,*pordered-ab-semigroup-add*}) ==>
setsum f A <= *setsum f B*
⟨*proof*⟩

lemma *setsum-right-distrib*:
fixes *f* :: '*a* => ('*b*::*semiring-0*)
shows *r* * *setsum f A* = *setsum* (%*n*. *r* * *f n*) *A*
⟨*proof*⟩

lemma *setsum-left-distrib*:
setsum f A * (*r*::'*a*::*semiring-0*) = (∑ *n*∈*A*. *f n* * *r*)
⟨*proof*⟩

lemma *setsum-divide-distrib*:
setsum f A / (*r*::'*a*::*field*) = (∑ *n*∈*A*. *f n* / *r*)
⟨*proof*⟩

lemma *setsum-abs*[*iff*]:
fixes *f* :: '*a* => ('*b*::*pordered-ab-group-add-abs*)
shows *abs* (*setsum f A*) ≤ *setsum* (%*i*. *abs*(*f i*)) *A*
⟨*proof*⟩

lemma *setsum-abs-ge-zero*[*iff*]:
fixes *f* :: '*a* => ('*b*::*pordered-ab-group-add-abs*)
shows *0* ≤ *setsum* (%*i*. *abs*(*f i*)) *A*

⟨*proof*⟩

lemma *abs-setsum-abs*[*simp*]:
fixes *f* :: *'a* => (*'b*::*pordered-ab-group-add-abs*)
shows *abs* (∑ *a*∈*A*. *abs*(*f a*)) = (∑ *a*∈*A*. *abs*(*f a*))
⟨*proof*⟩

Commuting outer and inner summation

lemma *swap-inj-on*:
inj-on (%(*i*, *j*). (*j*, *i*)) (*A* × *B*)
⟨*proof*⟩

lemma *swap-product*:
(%(*i*, *j*). (*j*, *i*)) ' (*A* × *B*) = *B* × *A*
⟨*proof*⟩

lemma *setsum-commute*:
(∑ *i*∈*A*. ∑ *j*∈*B*. *f i j*) = (∑ *j*∈*B*. ∑ *i*∈*A*. *f i j*)
⟨*proof*⟩

lemma *setsum-product*:
fixes *f* :: *'a* => (*'b*::*semiring-0*)
shows *setsum f A* * *setsum g B* = (∑ *i*∈*A*. ∑ *j*∈*B*. *f i* * *g j*)
⟨*proof*⟩

20.5 Generalized product over a set

constdefs
setprod :: (*'a* => *'b*) => *'a set* => *'b*::*comm-monoid-mult*
setprod f A == *if finite A then fold* (*op* *) *f 1 A else 1*

abbreviation
Setprod (∏ - [*1000*] *999*) **where**
∏ *A* == *setprod* (%*x*. *x*) *A*

syntax
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3PROD* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*∏ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*∏ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
PROD i:*A*. *b* == *setprod* (%*i*. *b*) *A*
∏ *i*∈*A*. *b* == *setprod* (%*i*. *b*) *A*

Instead of ∏ *x*∈{*x*. *P*}. *e* we introduce the shorter ∏ *x*|*P*. *e*.

syntax
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3PROD* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*∏ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*∏ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
PROD x|P. t => *setprod* (%*x. t*) {*x. P*}
∏ *x|P. t* => *setprod* (%*x. t*) {*x. P*}

lemma *setprod-empty* [*simp*]: *setprod f* {} = *1*
⟨*proof*⟩

lemma *setprod-insert* [*simp*]: [| *finite A*; *a* ∉ *A* |] ==>
setprod f (*insert a A*) = *f a* ∗ *setprod f A*
⟨*proof*⟩

lemma *setprod-infinite* [*simp*]: ~ *finite A* ==> *setprod f A* = *1*
⟨*proof*⟩

lemma *setprod-reindex*:
inj-on f B ==> *setprod h* (*f* ' *B*) = *setprod* (*h* ∘ *f*) *B*
⟨*proof*⟩

lemma *setprod-reindex-id*: *inj-on f B* ==> *setprod f B* = *setprod id* (*f* ' *B*)
⟨*proof*⟩

lemma *setprod-cong*:
A = *B* ==> (!!*x. x*:*B* ==> *f x* = *g x*) ==> *setprod f A* = *setprod g B*
⟨*proof*⟩

lemma *strong-setprod-cong*:
A = *B* ==> (!!*x. x*:*B* =*simp*=> *f x* = *g x*) ==> *setprod f A* = *setprod g B*
⟨*proof*⟩

lemma *setprod-reindex-cong*: *inj-on f A* ==>
B = *f* ' *A* ==> *g* = *h* ∘ *f* ==> *setprod h B* = *setprod g A*
⟨*proof*⟩

lemma *setprod-1*: *setprod* (%*i. 1*) *A* = *1*
⟨*proof*⟩

lemma *setprod-1'*: *ALL a*:*F. f a* = *1* ==> *setprod f F* = *1*
⟨*proof*⟩

lemma *setprod-Un-Int*: *finite A* ==> *finite B*

$==>$ *setprod g* $(A$ *Un* $B) *$ *setprod g* $(A$ *Int* $B) =$ *setprod g* $A *$ *setprod g* B
⟨*proof*⟩

lemma *setprod-Un-disjoint*: *finite* $A ==>$ *finite* B
$==> A$ *Int* $B = \{\} ==>$ *setprod g* $(A$ *Un* $B) =$ *setprod g* $A *$ *setprod g* B
⟨*proof*⟩

lemma *setprod-UN-disjoint*:
finite $I ==> (ALL\ i{:}I.$ *finite* $(A\ i)) ==>$
$(ALL\ i{:}I.\ ALL\ j{:}I.\ i \neq j \dashrightarrow A\ i$ *Int* $A\ j = \{\}) ==>$
setprod f $(UNION\ I\ A) =$ *setprod* $(\%i.$ *setprod f* $(A\ i))\ I$
⟨*proof*⟩

lemma *setprod-Union-disjoint*:
$[|\ (ALL\ A{:}C.$ *finite* $A);$
$(ALL\ A{:}C.\ ALL\ B{:}C.\ A \neq B \dashrightarrow A$ *Int* $B = \{\})\ |]$
$==>$ *setprod f* $(Union\ C) =$ *setprod* $(setprod\ f)\ C$
⟨*proof*⟩

lemma *setprod-Sigma*: *finite* $A ==> ALL\ x{:}A.$ *finite* $(B\ x) ==>$
$(\prod x{\in}A.\ (\prod y{\in} B\ x.\ f\ x\ y)) =$
$(\prod (x,y){\in}(SIGMA\ x{:}A.\ B\ x).\ f\ x\ y)$
⟨*proof*⟩

Here we can eliminate the finiteness assumptions, by cases.

lemma *setprod-cartesian-product*:
$(\prod x{\in}A.\ (\prod y{\in} B.\ f\ x\ y)) = (\prod (x,y){\in}(A <*> B).\ f\ x\ y)$
⟨*proof*⟩

lemma *setprod-timesf*:
setprod $(\%x.\ f\ x * g\ x)\ A =$ (*setprod f A* $*$ *setprod g A*)
⟨*proof*⟩

20.5.1 Properties in more restricted classes of structures

lemma *setprod-eq-1-iff* [*simp*]:
finite $F ==>$ (*setprod f F* $= 1) = (ALL\ a{:}F.\ f\ a = (1{::}nat))$
⟨*proof*⟩

lemma *setprod-zero*:
finite $A ==> EX\ x{:}\ A.\ f\ x = (0{::}'a{::}$*comm-semiring-1*$) ==>$ *setprod f A* $= 0$
⟨*proof*⟩

lemma *setprod-nonneg* [*rule-format*]:
$(ALL\ x{:}\ A.\ (0{::}'a{::}$*ordered-idom*$) \leq f\ x) \dashrightarrow 0 \leq$ *setprod f A*
⟨*proof*⟩

lemma *setprod-pos* [*rule-format*]: $(ALL\ x{:}\ A.\ (0{::}'a{::}$*ordered-idom*$) < f\ x)$
$\dashrightarrow 0 <$ *setprod f A*

⟨*proof*⟩

lemma *setprod-nonzero* [*rule-format*]:
(*ALL x y.* (*x*::'*a*::*comm-semiring-1*) * *y* = *0* −−> *x* = *0* | *y* = *0*) ==>
finite A ==> (*ALL x*: *A. f x* ≠ (*0*::'*a*)) −−> *setprod f A* ≠ *0*
⟨*proof*⟩

lemma *setprod-zero-eq*:
(*ALL x y.* (*x*::'*a*::*comm-semiring-1*) * *y* = *0* −−> *x* = *0* | *y* = *0*) ==>
finite A ==> (*setprod f A* = (*0*::'*a*)) = (*EX x*: *A. f x* = *0*)
⟨*proof*⟩

lemma *setprod-nonzero-field*:
finite A ==> (*ALL x*: *A. f x* ≠ (*0*::'*a*::*idom*)) ==> *setprod f A* ≠ *0*
⟨*proof*⟩

lemma *setprod-zero-eq-field*:
finite A ==> (*setprod f A* = (*0*::'*a*::*idom*)) = (*EX x*: *A. f x* = *0*)
⟨*proof*⟩

lemma *setprod-Un*: *finite A* ==> *finite B* ==> (*ALL x*: *A Int B. f x* ≠ *0*) ==>
(*setprod f* (*A Un B*) :: '*a* ::{*field*})
= *setprod f A* * *setprod f B* / *setprod f* (*A Int B*)
⟨*proof*⟩

lemma *setprod-diff1*: *finite A* ==> *f a* ≠ *0* ==>
(*setprod f* (*A* − {*a*}) :: '*a* :: {*field*}) =
(*if a*:*A then setprod f A* / *f a else setprod f A*)
⟨*proof*⟩

lemma *setprod-inversef*: *finite A* ==>
ALL x: *A. f x* ≠ (*0*::'*a*::{*field*,*division-by-zero*}) ==>
setprod (*inverse* ∘ *f*) *A* = *inverse* (*setprod f A*)
⟨*proof*⟩

lemma *setprod-dividef*:
[|*finite A*;
∀ *x* ∈ *A. g x* ≠ (*0*::'*a*::{*field*,*division-by-zero*})|]
==> *setprod* (%*x. f x* / *g x*) *A* = *setprod f A* / *setprod g A*
⟨*proof*⟩

20.6 Finite cardinality

This definition, although traditional, is ugly to work with: *card A* == *LEAST n. EX f. A* = {*f i* | *i. i* < *n*}. But now that we have *setsum* things are easy:

definition
card :: '*a set* ⇒ *nat*
where

card A = setsum (λx. 1) A

lemma *card-empty* [*simp*]: *card {} = 0*
⟨*proof*⟩

lemma *card-infinite* [*simp*]: ~ *finite A ==> card A = 0*
⟨*proof*⟩

lemma *card-eq-setsum*: *card A = setsum (%x. 1) A*
⟨*proof*⟩

lemma *card-insert-disjoint* [*simp*]:
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-0-eq* [*simp,noatp*]: *finite A ==> (card A = 0) = (A = {})*
⟨*proof*⟩

lemma *card-eq-0-iff*: *(card A = 0) = (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*: ⟦ *finite B*; *A ⊆ B* ⟧ ⟹ *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*:
finite B ==> *B* <= *A* ==> *card* (*A* − *B*) = *card A* − *card 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* ⊆ *B* ==> *card A* < *card B* ==> *A* < *B*
⟨*proof*⟩

lemma *insert-partition*:
⟦ *x* ∉ *F*; ∀ *c1* ∈ *insert x F*. ∀ *c2* ∈ *insert x F*. *c1* ≠ *c2* ⟶ *c1* ∩ *c2* = {} ⟧
⟹ *x* ∩ ⋃ *F* = {}
⟨*proof*⟩

main cardinality theorem

lemma *card-partition* [*rule-format*]:
finite C ==>
finite (⋃ *C*) −−>
(∀ *c*∈*C*. *card c* = *k*) −−>
(∀ *c1* ∈ *C*. ∀ *c2* ∈ *C*. *c1* ≠ *c2* −−> *c1* ∩ *c2* = {}) −−>
k ∗ *card*(*C*) = *card* (⋃ *C*)
⟨*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 *setsum-constant* [*simp*]: $(\sum x \in A.\ y) = of\text{-}nat(card\ A) * y$
⟨*proof*⟩

lemma *setprod-constant*: *finite A* ==> $(\prod x{\in} A.\ (y{::}'a{::}\{recpower, comm\text{-}monoid\text{-}mult\}))$
$= y\ \hat{}\ (card\ A)$
⟨*proof*⟩

lemma *setsum-bounded*:
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\ i \leq (K{::}'a{::}\{semiring\text{-}1, pordered\text{-}ab\text{-}semigroup\text{-}add\}) $
shows *setsum f A* $\leq$ *of-nat*(*card A*) $*$ *K*
⟨*proof*⟩

20.6.1 Cardinality of unions

lemma *card-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. *i* $\neq$ *j* −−> *A i Int A j* = {}) ==>
card (*UNION I A*) = $(\sum i{\in}I.\ card(A\ i))$
⟨*proof*⟩

lemma *card-Union-disjoint*:
finite C ==> (*ALL A*:*C*. *finite A*) ==>
(*ALL A*:*C*. *ALL B*:*C*. *A* $\neq$ *B* −−> *A Int B* = {}) ==>
card (*Union C*) = *setsum card C*
⟨*proof*⟩

20.6.2 Cardinality of image

The image of a finite set can be expressed using *fold*.

lemma *image-eq-fold*: *finite A* ==> *f* ‘ *A* = *fold* (*op Un*) (%*x*. {*f x*}) {} *A*
⟨*proof*⟩

lemma *card-image-le*: *finite A* ==> *card* (*f* ‘ *A*) <= *card A*
⟨*proof*⟩

lemma *card-image*: *inj-on f A* ==> *card* (*f* ‘ *A*) = *card A*
⟨*proof*⟩

lemma *endo-inj-surj*: *finite A* ==> *f* ‘ *A* $\subseteq$ *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*⟩

20.6.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
⟦ *finite A*; *ALL a*:*A*. *finite* (*B a*) ⟧
⟹ *card* (*SIGMA x*: *A*. *B x*) = (∑ *a*∈*A*. *card* (*B a*))
⟨*proof*⟩

lemma *card-cartesian-product*: *card* (*A* <*> *B*) = *card*(*A*) * *card*(*B*)
⟨*proof*⟩

lemma *card-cartesian-product-singleton*: *card*({*x*} <*> *A*) = *card*(*A*)
⟨*proof*⟩

20.6.4 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. Kammüller.

lemma *dvd-partition*:
finite (*Union C*) ==>
ALL c : *C*. *k dvd card c* ==>
(*ALL c1*: *C*. *ALL c2*: *C*. *c1* ≠ *c2* --> *c1 Int c2* = {}) ==>
k dvd card (*Union C*)
⟨*proof*⟩

20.6.5 Relating injectivity and surjectivity

lemma *finite-surj-inj*: *finite*(*A*) ⟹ *A* <= *f*'*A* ⟹ *inj-on f A*
⟨*proof*⟩

lemma *finite-UNIV-surj-inj*: **fixes** $f :: 'a \Rightarrow 'a$
shows *finite*(*UNIV*:: *'a set*) $\Longrightarrow$ *surj f* $\Longrightarrow$ *inj f*
⟨*proof*⟩

lemma *finite-UNIV-inj-surj*: **fixes** $f :: 'a \Rightarrow 'a$
shows *finite*(*UNIV*:: *'a set*) $\Longrightarrow$ *inj f* $\Longrightarrow$ *surj f*
⟨*proof*⟩

corollary *infinite-UNIV-nat*: ~*finite*(*UNIV*::*nat set*)
⟨*proof*⟩

20.7 A fold functional for non-empty sets

Does not require start value.

inductive
fold1Set :: (*'a => 'a => 'a*) => *'a set* => *'a* => *bool*
for *f* :: *'a => 'a => 'a*
where
fold1Set-insertI [*intro*]:
⟦ *foldSet f id a A x*; $a \notin A$ ⟧ $\Longrightarrow$ *fold1Set f* (*insert a A*) *x*

constdefs
fold1 :: (*'a => 'a => 'a*) => *'a set* => *'a*
fold1 f A == *THE x*. *fold1Set f A x*

lemma *fold1Set-nonempty*:
fold1Set f A x $\Longrightarrow$ $A \neq \{\}$
⟨*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 *foldSet*.

context *ab-semigroup-mult*
begin

lemma *foldSet-insert-swap*:
assumes *fold*: *foldSet times id b A y*
shows $b \notin A \Longrightarrow$ *foldSet times id z* (*insert b A*) ($z * y$)
⟨*proof*⟩

lemma *foldSet-permute-diff*:
assumes *fold*: *foldSet times id b A x*
shows !!a. $[\![a \in A;\ b \notin A]\!] \Longrightarrow$ *foldSet times id a* (*insert b* ($A - \{a\}$)) *x*
⟨*proof*⟩

lemma *fold1-eq-fold*:
[|*finite A*; $a \notin A$|] ==> *fold1 times* (*insert a A*) = *fold times id a A*
⟨*proof*⟩

lemma *nonempty-iff*: ($A \neq \{\}$) = ($\exists x\ B.\ A$ = *insert x B* & $x \notin B$)
⟨*proof*⟩

lemma *fold1-insert*:
assumes *nonempty*: $A \neq \{\}$ **and** *A*: *finite A* $x \notin A$
shows *fold1 times* (*insert x A*) = $x *$ *fold1 times A*
⟨*proof*⟩

end

context *ab-semigroup-idem-mult*
begin

lemma *fold1-insert-idem* [*simp*]:
assumes *nonempty*: $A \neq \{\}$ **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* $N \neq \{\}$ **shows** h (*fold1 times N*) = *fold1 times* (h ‘ *N*)
⟨*proof*⟩

end

Now the recursion rules for definitions:

lemma *fold1-singleton-def*: g = *fold1 f* $\Longrightarrow$ $g\ \{a\} = a$
⟨*proof*⟩

lemma (**in** *ab-semigroup-mult*) *fold1-insert-def*:
$[\![\ g$ = *fold1 times*; *finite A*; $x \notin A$; $A \neq \{\}\ ]\!] \Longrightarrow g$ (*insert x A*) = $x * g\ A$
⟨*proof*⟩

lemma (**in** *ab-semigroup-idem-mult*) *fold1-insert-idem-def*:
$[\![\ g$ = *fold1 times*; *finite A*; $A \neq \{\}\ ]\!] \Longrightarrow g$ (*insert x A*) = $x * g\ A$

⟨*proof*⟩

20.7.1 Determinacy for *fold1Set*

Not actually used!!

context *ab-semigroup-mult*
begin

lemma *foldSet-permute*:
[|*foldSet times id b* (*insert a A*) *x*; *a* ∉ *A*; *b* ∉ *A*|]
==> *foldSet times id a* (*insert b A*) *x*
⟨*proof*⟩

lemma *fold1Set-determ*:
fold1Set times A x ==> *fold1Set times A y* ==> *y* = *x*
⟨*proof*⟩

lemma *fold1Set-equality*: *fold1Set times A y* ==> *fold1 times A* = *y*
⟨*proof*⟩

end

declare
empty-foldSetE [*rule del*] *foldSet.intros* [*rule del*]
empty-fold1SetE [*rule del*] *insert-fold1SetE* [*rule del*]
— No more proofs involve these relations.

20.7.2 Lemmas about *fold1*

context *ab-semigroup-mult*
begin

lemma *fold1-Un*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹ *A Int B* = {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
⟨*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*⟩

20.7.3 Fold1 in lattices with *inf* and *sup*

As an application of *fold1* we define infimum and supremum in (not necessarily complete!) lattices over (non-empty) sets by means of *fold1*.

context *lower-semilattice*
begin

lemma *ab-semigroup-idem-mult-inf*:
ab-semigroup-idem-mult inf
⟨*proof*⟩

lemma *below-fold1-iff*:
assumes *finite A* $A \neq \{\}$
shows $x \leq$ *fold1 inf A* $\longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
⟨*proof*⟩

lemma *fold1-belowI*:
assumes *finite A*
and $a \in A$
shows *fold1 inf A* $\leq a$
⟨*proof*⟩

end

lemma (**in** *upper-semilattice*) *ab-semigroup-idem-mult-sup*:
ab-semigroup-idem-mult sup
⟨*proof*⟩

context *lattice*
begin

definition
Inf-fin :: *'a set* ⇒ *'a* ($\bigsqcap_{fin}$- [*900*] *900*)
where
Inf-fin = *fold1 inf*

definition
Sup-fin :: *'a set* ⇒ *'a* ($\bigsqcup_{fin}$- [*900*] *900*)
where
Sup-fin = *fold1 sup*

lemma *Inf-le-Sup* [*simp*]: ⟦ *finite A*; $A \neq \{\}$ ⟧ ⟹ $\bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
⟨*proof*⟩

lemma *sup-Inf-absorb* [*simp*]:
finite A ⟹ $a \in A$ ⟹ *sup a* ($\bigsqcap_{fin} A$) = *a*

⟨*proof*⟩

lemma *inf-Sup-absorb* [*simp*]:
finite $A \Longrightarrow a \in A \Longrightarrow inf\ a\ (\bigsqcup_{fin} A) = a$
⟨*proof*⟩

end

context *distrib-lattice*
begin

lemma *sup-Inf1-distrib*:
assumes *finite* A
and $A \neq \{\}$
shows $sup\ x\ (\bigsqcap_{fin} A) = \bigsqcap_{fin} \{sup\ x\ a | a.\ a \in A\}$
⟨*proof*⟩

lemma *sup-Inf2-distrib*:
assumes A: *finite* A $A \neq \{\}$ **and** B: *finite* B $B \neq \{\}$
shows $sup\ (\bigsqcap_{fin} A)\ (\bigsqcap_{fin} B) = \bigsqcap_{fin} \{sup\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
⟨*proof*⟩

lemma *inf-Sup1-distrib*:
assumes *finite* A **and** $A \neq \{\}$
shows $inf\ x\ (\bigsqcup_{fin} A) = \bigsqcup_{fin} \{inf\ x\ a | a.\ a \in A\}$
⟨*proof*⟩

lemma *inf-Sup2-distrib*:
assumes A: *finite* A $A \neq \{\}$ **and** B: *finite* B $B \neq \{\}$
shows $inf\ (\bigsqcup_{fin} A)\ (\bigsqcup_{fin} B) = \bigsqcup_{fin} \{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
⟨*proof*⟩

end

context *complete-lattice*
begin

Coincidence on finite sets in complete lattices:

lemma *Inf-fin-Inf*:
assumes *finite* A **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A = Inf\ A$
⟨*proof*⟩

lemma *Sup-fin-Sup*:
assumes *finite* A **and** $A \neq \{\}$
shows $\bigsqcup_{fin} A = Sup\ A$
⟨*proof*⟩

end

20.7.4 Fold1 in linear orders with *min* and *max*

As an application of *fold1* we define minimum and maximum in (not necessarily complete!) linear orders over (non-empty) sets by means of *fold1*.

context *linorder*
begin

lemma *ab-semigroup-idem-mult-min*:
ab-semigroup-idem-mult min
⟨*proof*⟩

lemma *ab-semigroup-idem-mult-max*:
ab-semigroup-idem-mult max
⟨*proof*⟩

lemma *min-lattice*:
lower-semilattice (*op* ≤) (*op* <) *min*
⟨*proof*⟩

lemma *max-lattice*:
lower-semilattice (*op* ≥) (*op* >) *max*
⟨*proof*⟩

lemma *dual-max*:
ord.max (*op* ≥) = *min*
⟨*proof*⟩

lemma *dual-min*:
ord.min (*op* ≥) = *max*
⟨*proof*⟩

lemma *strict-below-fold1-iff*:
assumes *finite A* **and** *A* ≠ {}
shows *x* < *fold1 min A* ⟷ (∀ *a*∈*A*. *x* < *a*)
⟨*proof*⟩

lemma *fold1-below-iff*:
assumes *finite A* **and** *A* ≠ {}
shows *fold1 min A* ≤ *x* ⟷ (∃ *a*∈*A*. *a* ≤ *x*)
⟨*proof*⟩

lemma *fold1-strict-below-iff*:
assumes *finite A* **and** *A* ≠ {}
shows *fold1 min A* < *x* ⟷ (∃ *a*∈*A*. *a* < *x*)
⟨*proof*⟩

lemma *fold1-antimono*:
assumes *A* ≠ {} **and** *A* ⊆ *B* **and** *finite B*
shows *fold1 min B* ≤ *fold1 min A*

⟨*proof*⟩

definition
  Min :: *'a set* ⇒ *'a*
where
  Min = *fold1 min*

definition
  Max :: *'a set* ⇒ *'a*
where
  Max = *fold1 max*

lemmas *Min-singleton* [*simp*] = *fold1-singleton-def* [*OF Min-def*]
lemmas *Max-singleton* [*simp*] = *fold1-singleton-def* [*OF Max-def*]

lemma *Min-insert* [*simp*]:
  assumes *finite A* **and** *A* ≠ {}
  shows *Min* (*insert x A*) = *min x* (*Min A*)
⟨*proof*⟩

lemma *Max-insert* [*simp*]:
  assumes *finite A* **and** *A* ≠ {}
  shows *Max* (*insert x A*) = *max x* (*Max A*)
⟨*proof*⟩

lemma *Min-in* [*simp*]:
  assumes *finite A* **and** *A* ≠ {}
  shows *Min A* ∈ *A*
⟨*proof*⟩

lemma *Max-in* [*simp*]:
  assumes *finite A* **and** *A* ≠ {}
  shows *Max A* ∈ *A*
⟨*proof*⟩

lemma *Min-Un*:
  assumes *finite A* **and** *A* ≠ {} **and** *finite B* **and** *B* ≠ {}
  shows *Min* (*A* ∪ *B*) = *min* (*Min A*) (*Min B*)
⟨*proof*⟩

lemma *Max-Un*:
  assumes *finite A* **and** *A* ≠ {} **and** *finite B* **and** *B* ≠ {}
  shows *Max* (*A* ∪ *B*) = *max* (*Max A*) (*Max B*)
⟨*proof*⟩

lemma *hom-Min-commute*:
  assumes ⋀*x y*. *h* (*min x y*) = *min* (*h x*) (*h y*)
    and *finite N* **and** *N* ≠ {}
  shows *h* (*Min N*) = *Min* (*h* ‘ *N*)

⟨*proof*⟩

lemma *hom-Max-commute*:
assumes $\bigwedge x\ y.\ h\ (max\ x\ y) = max\ (h\ x)\ (h\ y)$
and *finite* N **and** $N \neq \{\}$
shows $h\ (Max\ N) = Max\ (h\ `\ N)$
⟨*proof*⟩

lemma *Min-le* [*simp*]:
assumes *finite* A **and** $x \in A$
shows $Min\ A \leq x$
⟨*proof*⟩

lemma *Max-ge* [*simp*]:
assumes *finite* A **and** $x \in A$
shows $x \leq Max\ A$
⟨*proof*⟩

lemma *Min-ge-iff* [*simp*, *noatp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $x \leq Min\ A \longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
⟨*proof*⟩

lemma *Max-le-iff* [*simp*, *noatp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $Max\ A \leq x \longleftrightarrow (\forall\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *Min-gr-iff* [*simp*, *noatp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $x < Min\ A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
⟨*proof*⟩

lemma *Max-less-iff* [*simp*, *noatp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $Max\ A < x \longleftrightarrow (\forall\, a{\in}A.\ a < x)$
⟨*proof*⟩

lemma *Min-le-iff* [*noatp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $Min\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *Max-ge-iff* [*noatp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $x \leq Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$
⟨*proof*⟩

lemma *Min-less-iff* [*noatp*]:

assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
⟨*proof*⟩

lemma *Max-gr-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x < Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$
⟨*proof*⟩

lemma *Min-antimono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Min\ N \leq Min\ M$
⟨*proof*⟩

lemma *Max-mono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Max\ M \leq Max\ N$
⟨*proof*⟩

lemma *finite-linorder-induct*[*consumes 1*, *case-names empty insert*]:
finite A $\Longrightarrow$ $P\ \{\} \Longrightarrow$
(!!$A\ b$. *finite A* $\Longrightarrow$ $ALL\ a{:}A.\ a < b \Longrightarrow P\ A \Longrightarrow P(insert\ b\ A)$)
$\Longrightarrow P\ A$
⟨*proof*⟩

end

context *ordered-ab-semigroup-add*
begin

lemma *add-Min-commute*:
fixes *k*
assumes *finite N* **and** $N \neq \{\}$
shows $k + Min\ N = Min\ \{k + m \mid m.\ m \in N\}$
⟨*proof*⟩

lemma *add-Max-commute*:
fixes *k*
assumes *finite N* **and** $N \neq \{\}$
shows $k + Max\ N = Max\ \{k + m \mid m.\ m \in N\}$
⟨*proof*⟩

end

end

21 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *Finite-Set Relation*
begin

21.1 Equivalence relations

locale *equiv* =
  fixes A **and** r
  assumes *refl*: *refl* A r
    and *sym*: *sym* r
    and *trans*: *trans* r

Suppes, Theorem 70: r is an equiv relation iff $r^{-1}\ O\ r = r$.

First half: *equiv* A r ==> $r^{-1}\ O\ r = r$.

lemma *sym-trans-comp-subset*:
    sym r ==> *trans* r ==> $r^{-1}\ O\ r \subseteq r$
  ⟨*proof*⟩

lemma *refl-comp-subset*: *refl* A r ==> $r \subseteq r^{-1}\ O\ r$
  ⟨*proof*⟩

lemma *equiv-comp-eq*: *equiv* A r ==> $r^{-1}\ O\ r = r$
  ⟨*proof*⟩

Second half.

lemma *comp-equivI*:
    $r^{-1}\ O\ r = r$ ==> *Domain* $r = A$ ==> *equiv* A r
  ⟨*proof*⟩

21.2 Equivalence classes

lemma *equiv-class-subset*:
  equiv A r ==> $(a, b) \in r$ ==> $r``\{a\} \subseteq r``\{b\}$
  — lemma for the next result
  ⟨*proof*⟩

theorem *equiv-class-eq*: *equiv* A r ==> $(a, b) \in r$ ==> $r``\{a\} = r``\{b\}$
  ⟨*proof*⟩

lemma *equiv-class-self*: *equiv* A r ==> $a \in A$ ==> $a \in r``\{a\}$
  ⟨*proof*⟩

lemma *subset-equiv-class*:
    equiv A r ==> $r``\{b\} \subseteq r``\{a\}$ ==> $b \in A$ ==> $(a,b) \in r$
  — lemma for the next result
  ⟨*proof*⟩

lemma *eq-equiv-class*:
$r``\{a\} = r``\{b\} ==> equiv\ A\ r ==> b \in A ==> (a, b) \in r$
⟨*proof*⟩

lemma *equiv-class-nondisjoint*:
$equiv\ A\ r ==> x \in (r``\{a\} \cap r``\{b\}) ==> (a, b) \in r$
⟨*proof*⟩

lemma *equiv-type*: $equiv\ A\ r ==> r \subseteq A \times A$
⟨*proof*⟩

theorem *equiv-class-eq-iff*:
$equiv\ A\ r ==> ((x, y) \in r) = (r``\{x\} = r``\{y\} \ \&\ x \in A \ \&\ y \in A)$
⟨*proof*⟩

theorem *eq-equiv-class-iff*:
$equiv\ A\ r ==> x \in A ==> y \in A ==> (r``\{x\} = r``\{y\}) = ((x, y) \in r)$
⟨*proof*⟩

21.3 Quotients

constdefs
quotient :: ['*a set*, ('*a*∗'*a*) *set*] => '*a set set* (**infixl** '//' *90*)
$A//r == \bigcup x \in A.\ \{r``\{x\}\}$ — set of equiv classes

lemma *quotientI*: $x \in A ==> r``\{x\} \in A//r$
⟨*proof*⟩

lemma *quotientE*:
$X \in A//r ==> (!!x.\ X = r``\{x\} ==> x \in A ==> P) ==> P$
⟨*proof*⟩

lemma *Union-quotient*: $equiv\ A\ r ==> Union\ (A//r) = A$
⟨*proof*⟩

lemma *quotient-disj*:
$equiv\ A\ r ==> X \in A//r ==> Y \in A//r ==> X = Y \mid (X \cap Y = \{\})$
⟨*proof*⟩

lemma *quotient-eqI*:
$[|equiv\ A\ r;\ X \in A//r;\ Y \in A//r;\ x \in X;\ y \in Y;\ (x,y) \in r|] ==> X = Y$
⟨*proof*⟩

lemma *quotient-eq-iff*:
$[|equiv\ A\ r;\ X \in A//r;\ Y \in A//r;\ x \in X;\ y \in Y|] ==> (X = Y) = ((x,y) \in r)$
⟨*proof*⟩

lemma *eq-equiv-class-iff2*:
⟦ *equiv A r*; $x \in A$; $y \in A$ ⟧ ⟹ $(\{x\}//r = \{y\}//r) = ((x,y) : r)$
⟨*proof*⟩

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
⟨*proof*⟩

lemma *quotient-is-empty* [*iff*]: $(A//r = \{\}) = (A = \{\})$
⟨*proof*⟩

lemma *quotient-is-empty2* [*iff*]: $(\{\} = A//r) = (A = \{\})$
⟨*proof*⟩

lemma *singleton-quotient*: $\{x\}//r = \{r \text{ `` } \{x\}\}$
⟨*proof*⟩

lemma *quotient-diff1*:
⟦ *inj-on* (%*a*. $\{a\}//r$) *A*; $a \in A$ ⟧ ⟹ $(A - \{a\})//r = A//r - \{a\}//r$
⟨*proof*⟩

21.4 Defining unary operations upon equivalence classes

A congruence-preserving function

locale *congruent* =
fixes *r* **and** *f*
assumes *congruent*: $(y,z) \in r$ ==> $f\ y = f\ z$

abbreviation
RESPECTS :: $('a$ => $'b)$ => $('a * 'a)$ *set* => *bool*
(**infixr** *respects 80*) **where**
f respects r == *congruent r f*

lemma *UN-constant-eq*: $a \in A$ ==> $\forall y \in A.\ f\ y = c$ ==> $(\bigcup y \in A.\ f(y))=c$
— lemma required to prove *UN-equiv-class*
⟨*proof*⟩

lemma *UN-equiv-class*:
equiv A r ==> *f respects r* ==> $a \in A$
==> $(\bigcup x \in r\text{``}\{a\}.\ f\ x) = f\ a$
— Conversion rule
⟨*proof*⟩

lemma *UN-equiv-class-type*:
equiv A r ==> *f respects r* ==> $X \in A//r$ ==>
(!!x. $x \in A$ ==> $f\ x \in B$) ==> $(\bigcup x \in X.\ f\ x) \in B$
⟨*proof*⟩

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be $!!y.\ y \in A \Longrightarrow f\ y \in B$.

lemma *UN-equiv-class-inject*:
equiv A r ==> f respects r ==>
$(\bigcup x \in X.\ f\ x) = (\bigcup y \in Y.\ f\ y)$ *==> X ∈ A//r ==> Y ∈ A//r*
==> $(!!x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow (x,\ y) \in r)$
==> X = Y
⟨*proof*⟩

21.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

locale *congruent2* =
fixes *r1* **and** *r2* **and** *f*
assumes *congruent2*:
(y1,z1) ∈ r1 ==> (y2,z2) ∈ r2 ==> f y1 y2 = f z1 z2

Abbreviation for the common case where the relations are identical

abbreviation
RESPECTS2:: $['a \Rightarrow 'a \Rightarrow 'b,\ ('a * 'a)\ set] \Rightarrow bool$
(**infixr** *respects2 80*) **where**
f respects2 r == congruent2 r r f

lemma *congruent2-implies-congruent*:
equiv A r1 ==> congruent2 r1 r2 f ==> a ∈ A ==> congruent r2 (f a)
⟨*proof*⟩

lemma *congruent2-implies-congruent-UN*:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a ∈ A2 ==>
congruent r1 $(\lambda x1.\ \bigcup x2 \in r2``\{a\}.\ f\ x1\ x2)$
⟨*proof*⟩

lemma *UN-equiv-class2*:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a1 ∈ A1 ==> a2 ∈ A2
$\Longrightarrow (\bigcup x1 \in r1``\{a1\}.\ \bigcup x2 \in r2``\{a2\}.\ f\ x1\ x2) = f\ a1\ a2$
⟨*proof*⟩

lemma *UN-equiv-class-type2*:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f
==> X1 ∈ A1//r1 ==> X2 ∈ A2//r2
$\Longrightarrow (!!x1\ x2.\ x1 \in A1 \Longrightarrow x2 \in A2 \Longrightarrow f\ x1\ x2 \in B)$
$\Longrightarrow (\bigcup x1 \in X1.\ \bigcup x2 \in X2.\ f\ x1\ x2) \in B$
⟨*proof*⟩

lemma *UN-UN-split-split-eq*:
$(\bigcup (x1,\ x2) \in X.\ \bigcup (y1,\ y2) \in Y.\ A\ x1\ x2\ y1\ y2) =$

$(\bigcup x \in X.\ \bigcup y \in Y.\ (\lambda(x1,\ x2).\ (\lambda(y1,\ y2).\ A\ x1\ x2\ y1\ y2)\ y)\ x)$
— Allows a natural expression of binary operators,
— without explicit calls to *split*
⟨*proof*⟩

lemma *congruent2I*:
equiv A1 r1 ==> equiv A2 r2
==> (!!y z w. w ∈ A2 ==> (y,z) ∈ r1 ==> f y w = f z w)
==> (!!y z w. w ∈ A1 ==> (y,z) ∈ r2 ==> f w y = f w z)
==> congruent2 r1 r2 f
— Suggested by John Harrison – the two subproofs may be
— *much* simpler than the direct proof.
⟨*proof*⟩

lemma *congruent2-commuteI*:
assumes *equivA*: *equiv A r*
and *commute*: *!!y z. y ∈ A ==> z ∈ A ==> f y z = f z y*
and *congt*: *!!y z w. w ∈ A ==> (y,z) ∈ r ==> f w y = f w z*
shows *f respects2 r*
⟨*proof*⟩

21.6 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*: *finite A ==> r ⊆ A × A ==> finite (A//r)*
— recall *equiv ?A ?r ⟹ ?r ⊆ ?A × ?A*
⟨*proof*⟩

lemma *finite-equiv-class*:
finite A ==> r ⊆ A × A ==> X ∈ A//r ==> finite X
⟨*proof*⟩

lemma *equiv-imp-dvd-card*:
finite A ==> equiv A r ==> ∀ X ∈ A//r. k dvd card X
==> k dvd card A
⟨*proof*⟩

lemma *card-quotient-disjoint*:
⟦ *finite A*; *inj-on (λx. {x} // r) A* ⟧ ⟹ *card(A//r) = card A*
⟨*proof*⟩

end

22 Wellfounded: Well-founded Recursion

theory *Wellfounded*
imports *Finite-Set Nat*

uses (*Tools/function-package/size.ML*)
begin

22.1 Basic Definitions

inductive
wfrec-rel :: (*'a* ∗ *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *'a* => *'b* => *bool*
for *R* :: (*'a* ∗ *'a*) *set*
and *F* :: (*'a* => *'b*) => *'a* => *'b*
where
wfrecI: *ALL z*. (*z*, *x*) : *R* −−> *wfrec-rel R F z* (*g z*) ==>
wfrec-rel R F x (*F g x*)

constdefs
wf :: (*'a* ∗ *'a*)*set* => *bool*
wf(*r*) == (!*P*. (!*x*. (!*y*. (*y*,*x*):*r* −−> *P*(*y*)) −−> *P*(*x*)) −−> (!*x*. *P*(*x*)))

wfP :: (*'a* => *'a* => *bool*) => *bool*
wfP r == *wf* {(*x*, *y*). *r x y*}

acyclic :: (*'a*∗*'a*)*set* => *bool*
acyclic r == !*x*. (*x*,*x*) ~: *r*^+

cut :: (*'a* => *'b*) => (*'a* ∗ *'a*)*set* => *'a* => *'a* => *'b*
cut f r x == (%*y*. *if* (*y*,*x*):*r then f y else arbitrary*)

adm-wf :: (*'a* ∗ *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *bool*
adm-wf R F == *ALL f g x*.
(*ALL z*. (*z*, *x*) : *R* −−> *f z* = *g z*) −−> *F f x* = *F g x*

wfrec :: (*'a* ∗ *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *'a* => *'b*
[*code func del*]: *wfrec R F* == %*x*. *THE y*. *wfrec-rel R* (%*f x*. *F* (*cut f R x*) *x*) *x y*

abbreviation *acyclicP* :: (*'a* => *'a* => *bool*) => *bool* **where**
acyclicP r == *acyclic* {(*x*, *y*). *r x y*}

class *wellorder* = *linorder* +
assumes *wf*: *wf* {(*x*, *y*). *x* < *y*}

lemma *wfP-wf-eq* [*pred-set-conv*]: *wfP* (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*:
[| $r \subseteq A$ <*> B;
!!x P. [|$\forall x.\ (\forall y.\ (y,x) : r$ −−> $P\ y)$ −−> $P\ x$; $x : A$; $x : B$ |] ==> $P\ x$ |]
==> *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*⟩

lemmas *wf-asym* = *wf-not-sym* [*elim-format*]

lemma *wf-not-refl* [*simp*]: *wf r* ==> (a, a) ~: r
⟨*proof*⟩

lemmas *wf-irrefl* = *wf-not-refl* [*elim-format*]

22.2 Basic Results

transitive closure of a well-founded relation is well-founded!

lemma *wf-trancl*:
assumes *wf r*
shows *wf* (r^+)
⟨*proof*⟩

lemmas *wfP-trancl* = *wf-trancl* [*to-pred*]

lemma *wf-converse-trancl*: *wf* (r^−1) ==> *wf* ((r^+)^−1)
⟨*proof*⟩

Minimal-element characterization of well-foundedness

lemma *wf-eq-minimal*: *wf r = (∀ Q x. x∈Q --> (∃ z∈Q. ∀ y. (y,z)∈r --> y∉Q))*
⟨*proof*⟩

lemma *wfE-min*:
assumes *wf R x ∈ Q*
obtains *z* **where** *z ∈ Q ⋀y. (y, z) ∈ R ⟹ y ∉ Q*
⟨*proof*⟩

lemma *wfI-min*:
(⋀x Q. x ∈ Q ⟹ ∃ z∈Q. ∀ y. (y, z) ∈ R ⟶ y ∉ Q)
⟹ wf R
⟨*proof*⟩

lemmas *wfP-eq-minimal = wf-eq-minimal [to-pred]*

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*⟩

lemmas *wfP-empty [iff] =*
wf-empty [to-pred bot-empty-eq2, simplified bot-fun-eq bot-bool-eq]

lemma *wf-Int1*: *wf r ==> wf (r Int r')*
⟨*proof*⟩

lemma *wf-Int2*: *wf r ==> wf (r' Int 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*⟩

22.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
assumes *wf R wf S*
assumes *S O R ⊆ R*
shows *wf (R ∪ S)*

⟨*proof*⟩

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*⟩

lemmas *wfP-SUP* = *wf-UN* [**where** *I=UNIV* **and** *r=λi. {(x, y). r i x y}*,
to-pred SUP-UN-eq2 bot-empty-eq pred-equals-eq, *simplified*, *standard*]

lemma *wf-Union*:
[| *ALL r:R. wf r*;
ALL r:R. ALL s:R. r ~= s --> Domain r Int Range s = {}
|] ==> *wf(Union R)*
⟨*proof*⟩

lemma *wf-Un*:
[| *wf r*; *wf s*; *Domain r Int Range s = {}* |] ==> *wf(r Un s)*
⟨*proof*⟩

lemma *wf-union-merge*:
wf (R ∪ S) = wf (R O R ∪ R O S ∪ S) (**is** *wf ?A = wf ?B*)
⟨*proof*⟩

lemma *wf-comp-self*: *wf R = wf (R O R)* — special case
⟨*proof*⟩

22.3.1 acyclic

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*⟩

22.4 Well-Founded Recursion

cut

lemma *cuts-eq*: (*cut f r x* = *cut g r x*) = (*ALL y*. (*y*,*x*):*r* −−> *f*(*y*)=*g*(*y*))
⟨*proof*⟩

lemma *cut-apply*: (*x*,*a*):*r* ==> (*cut f r a*)(*x*) = *f*(*x*)
⟨*proof*⟩

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"

lemma *wfrec-unique*: [| *adm-wf R F*; *wf R* |] ==> *EX*! *y*. *wfrec-rel R F x y*
⟨*proof*⟩

lemma *adm-lemma*: *adm-wf R* (%*f x*. *F* (*cut f R x*) *x*)
⟨*proof*⟩

lemma *wfrec*: *wf*(*r*) ==> *wfrec r H a* = *H* (*cut* (*wfrec r H*) *r a*) *a*
⟨*proof*⟩

22.5 Code generator setup

consts-code
  wfrec (⟨**module**⟩*wfrec?*)
attach ⟪
fun wfrec f x = *f* (*wfrec f*) *x*;
⟫

22.6 LEAST and wellorderings

See also *wf-linord-ex-has-least* and its consequences in *Wellfounded-Relations.ML*

lemma *wellorder-Least-lemma* [*rule-format*]:
  $P\ (k::'a::wellorder) \longrightarrow P\ (LEAST\ x.\ P(x))\ \&\ (LEAST\ x.\ P(x)) <= k$
⟨*proof*⟩

lemmas *LeastI* = *wellorder-Least-lemma* [*THEN conjunct1*, *standard*]
lemmas *Least-le* = *wellorder-Least-lemma* [*THEN conjunct2*, *standard*]

— The following 3 lemmas are due to Brian Huffman
lemma *LeastI-ex*: $EX\ x::'a::wellorder.\ P\ x \Longrightarrow P\ (Least\ P)$
⟨*proof*⟩

lemma *LeastI2*:
  $[|\ P\ (a::'a::wellorder);\ !!x.\ P\ x \Longrightarrow Q\ x\ |] \Longrightarrow Q\ (Least\ P)$
⟨*proof*⟩

lemma *LeastI2-ex*:
  $[|\ EX\ a::'a::wellorder.\ P\ a;\ !!x.\ P\ x \Longrightarrow Q\ x\ |] \Longrightarrow Q\ (Least\ P)$
⟨*proof*⟩

lemma *not-less-Least*: $[|\ k < (LEAST\ x.\ P\ x)\ |] \Longrightarrow \sim P\ (k::'a::wellorder)$
⟨*proof*⟩

22.7 *nat* is well-founded

lemma *less-nat-rel*: $op < = (\lambda m\ n.\ n = Suc\ m)\hat{}++$
⟨*proof*⟩

definition
  pred-nat :: $(nat * nat)\ set$ **where**
  $pred\text{-}nat = \{(m,\ n).\ n = Suc\ m\}$

definition
  less-than :: $(nat * nat)\ set$ **where**
  $less\text{-}than = pred\text{-}nat\hat{}+$

lemma *less-eq*: $(m,\ n) \in pred\text{-}nat\hat{}+ \longleftrightarrow m < n$
  ⟨*proof*⟩

lemma *pred-nat-trancl-eq-le*:
  $(m,\ n) \in pred\text{-}nat\hat{}* \longleftrightarrow m \leq n$
  ⟨*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*⟩

Type *nat* is a wellfounded order

instance *nat* :: *wellorder*
⟨*proof*⟩

LEAST theorems for type *nat*

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*⟩

22.8 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [**?**].

inductive-set
acc :: ('*a* ∗ '*a*) *set* => '*a set*
for *r* :: ('*a* ∗ '*a*) *set*
where
accI: (!!*y*. (*y*, *x*) : *r* ==> *y* : *acc r*) ==> *x* : *acc r*

abbreviation
termip :: ('*a* => '*a* => *bool*) => '*a* => *bool* **where**
termip r == *accp* (r^{-1-1})

abbreviation
termi :: ('*a* ∗ '*a*) *set* => '*a set* **where**
termi r == *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

Induction rules

theorem *accp-induct*:

assumes *major*: *accp r a*
assumes *hyp*: !!*x*. *accp r x* ==> ∀ *y*. *r y x* −−> *P y* ==> *P x*
shows *P a*
⟨*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** ¬ *accp R z*
⟨*proof*⟩

lemma *accp-downwards-aux*: r^{**} *b a* ==> *accp r a* −−> *accp r b*
⟨*proof*⟩

theorem *accp-downwards*: *accp r a* ==> r^{**} *b a* ==> *accp r b*
⟨*proof*⟩

theorem *accp-wfPI*: ∀ *x*. *accp r x* ==> *wfP r*
⟨*proof*⟩

theorem *accp-wfPD*: *wfP r* ==> *accp r x*
⟨*proof*⟩

theorem *wfP-accp-iff*: *wfP r* = (∀ *x*. *accp r x*)
⟨*proof*⟩

Smaller relations have bigger accessible parts:

lemma *accp-subset*:
assumes *sub*: *R1* ≤ *R2*
shows *accp R2* ≤ *accp R1*
⟨*proof*⟩

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
assumes *subset*: *D* ≤ *accp R*
and *dcl*: ⋀*x z*. ⟦*D x*; *R z x*⟧ ⟹ *D z*
and *D x*
and *istep*: ⋀*x*. ⟦*D x*; (⋀*z*. *R z x* ⟹ *P z*)⟧ ⟹ *P x*
shows *P x*
⟨*proof*⟩

Set versions of the above theorems

lemmas *acc-induct* = *accp-induct* [*to-set*]

lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

lemmas *acc-downward* = *accp-downward* [*to-set*]

lemmas *not-acc-down* = *not-accp-down* [*to-set*]

lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]

lemmas *acc-downwards* = *accp-downwards* [*to-set*]

lemmas *acc-wfI* = *accp-wfPI* [*to-set*]

lemmas *acc-wfD* = *accp-wfPD* [*to-set*]

lemmas *wf-acc-iff* = *wfP-accp-iff* [*to-set*]

lemmas *acc-subset* = *accp-subset* [*to-set pred-subset-eq*]

lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set pred-subset-eq*]

22.9 Tools for building wellfounded relations

Inverse Image

lemma *wf-inv-image* [*simp*,*intro*!]: *wf*(*r*) ==> *wf*(*inv-image r* (*f*::'*a*=>'*b*))
⟨*proof*⟩

lemma *in-inv-image*[*simp*]: ((*x*,*y*) : *inv-image r f*) = ((*f x*, *f y*) : *r*)
⟨*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* ⟹ *wf* (*f* <∗*mlex*∗> *R*)
⟨*proof*⟩

lemma *mlex-less*: *f x* < *f y* ⟹ (*x*, *y*) ∈ *f* <∗*mlex*∗> *R*
⟨*proof*⟩

lemma *mlex-leq*: *f x* ≤ *f y* ⟹ (*x*, *y*) ∈ *R* ⟹ (*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*: *wf*(*finite-psubset*)
⟨*proof*⟩

lemma *trans-finite-psubset*: *trans finite-psubset*
⟨*proof*⟩

Wellfoundedness of *same-fst*

definition
same-fst :: ('*a* => *bool*) => ('*a* => ('*b* ∗ '*b*)*set*) => (('*a*∗'*b*)∗('*a*∗'*b*))*set*
where
same-fst P R == {((*x'*,*y'*),(*x*,*y*)) . *x'*=*x* & *P x* & (*y'*,*y*) : *R x*}
— For *rec-def* declarations where the first n parameters stay unchanged in the recursive call. See *Library/While-Combinator.thy* for an application.

lemma *same-fstI* [*intro*!]:
[| *P x*; (*y'*,*y*) : *R x* |] ==> ((*x*,*y'*),(*x*,*y*)) : *same-fst P R*
⟨*proof*⟩

lemma *wf-same-fst*:

assumes *prem*: $(!!x.\ P\ x ==> wf(R\ x))$
shows $wf(same\text{-}fst\ P\ R)$
⟨*proof*⟩

22.10 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

lemma *lemma1*: $[|\ ALL\ i.\ (f\ (Suc\ i), f\ i) : r\hat{}*\ |] ==> (f\ (i+k), f\ i) : r\hat{}*$
⟨*proof*⟩

lemma *lemma2*: $[|\ ALL\ i.\ (f\ (Suc\ i), f\ i) : r\hat{}*;\ wf\ (r\hat{}+)\ |]$
$==>\ ALL\ m.\ f\ m = x\ --> (EX\ i.\ ALL\ k.\ f\ (m+i+k) = f\ (m+i))$
⟨*proof*⟩

lemma *wf-weak-decr-stable*: $[|\ ALL\ i.\ (f\ (Suc\ i), f\ i) : r\hat{}*;\ wf\ (r\hat{}+)\ |]$
$==>\ EX\ i.\ ALL\ k.\ f\ (i+k) = f\ i$
⟨*proof*⟩

lemma *weak-decr-stable*:
$ALL\ i.\ f\ (Suc\ i) <= ((f\ i)::nat) ==>\ EX\ i.\ ALL\ k.\ f\ (i+k) = f\ i$
⟨*proof*⟩

22.11 size of a datatype value

⟨*ML*⟩

lemma *nat-size* [*simp*, *code func*]: $size\ (n::nat) = n$
⟨*proof*⟩

end

23 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *Int*
imports *Equiv-Relations Nat Wellfounded*
uses
(*Tools/numeral.ML*)
(*Tools/numeral-syntax.ML*)
(~~/*src/Provers/Arith/assoc-fold.ML*)
~~/*src/Provers/Arith/cancel-numerals.ML*
~~/*src/Provers/Arith/combine-numerals.ML*
(*int-arith1.ML*)
begin

23.1 The equivalence relation underlying the integers

definition
intrel :: $((nat \times nat) \times (nat \times nat))$ *set*
where
$intrel = \{((x, y), (u, v)) \mid x\ y\ u\ v.\ x + v = u + y\ \}$

typedef (*Integ*)
$int = UNIV//intrel$
⟨*proof*⟩

instantiation *int* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *ord*, *abs*, *sgn*}
begin

definition
Zero-int-def [*code func del*]: $0 = Abs\text{-}Integ\ (intrel\ ``\ \{(0, 0)\})$

definition
One-int-def [*code func del*]: $1 = Abs\text{-}Integ\ (intrel\ ``\ \{(1, 0)\})$

definition
add-int-def [*code func del*]: $z + w = Abs\text{-}Integ$
$(\bigcup(x, y) \in Rep\text{-}Integ\ z.\ \bigcup(u, v) \in Rep\text{-}Integ\ w.$
$intrel\ ``\ \{(x + u, y + v)\})$

definition
minus-int-def [*code func del*]:
$-z = Abs\text{-}Integ\ (\bigcup(x, y) \in Rep\text{-}Integ\ z.\ intrel\ ``\ \{(y, x)\})$

definition
diff-int-def [*code func del*]: $z - w = z + (-w :: int)$

definition
mult-int-def [*code func del*]: $z * w = Abs\text{-}Integ$
$(\bigcup(x, y) \in Rep\text{-}Integ\ z.\ \bigcup(u,v) \in Rep\text{-}Integ\ w.$
$intrel\ ``\ \{(x*u + y*v, x*v + y*u)\})$

definition
le-int-def [*code func del*]:
$z \leq w \longleftrightarrow (\exists x\ y\ u\ v.\ x+v \leq u+y \wedge (x, y) \in Rep\text{-}Integ\ z \wedge (u, v) \in Rep\text{-}Integ\ w)$

definition
less-int-def [*code func del*]: $(z::int) < w \longleftrightarrow z \leq w \wedge z \neq w$

definition
zabs-def: $|i::int| = (if\ i < 0\ then\ -i\ else\ i)$

definition
zsgn-def: $sgn\ (i::int) = (if\ i=0\ then\ 0\ else\ if\ 0<i\ then\ 1\ else\ -1)$

instance ⟨*proof*⟩

end

23.2 Construction of the Integers

lemma *intrel-iff* [*simp*]: $(((x,y),(u,v)) \in intrel) = (x+v = u+y)$
⟨*proof*⟩

lemma *equiv-intrel*: *equiv UNIV intrel*
⟨*proof*⟩

Reduces equality of equivalence classes to the *intrel* relation: $(intrel\ ``\ \{x\} = intrel\ ``\ \{y\}) = ((x,\ y) \in intrel)$

lemmas *equiv-intrel-iff* [*simp*] = *eq-equiv-class-iff* [*OF equiv-intrel UNIV-I UNIV-I*]

All equivalence classes belong to set of representatives

lemma [*simp*]: $intrel``\{(x,y)\} \in Integ$
⟨*proof*⟩

Reduces equality on abstractions to equality on representatives: $[\![x \in Integ; y \in Integ]\!] \Longrightarrow (Abs\text{-}Integ\ x = Abs\text{-}Integ\ y) = (x = y)$

declare *Abs-Integ-inject* [*simp*,*noatp*] *Abs-Integ-inverse* [*simp*,*noatp*]

Case analysis on the representation of an integer as an equivalence class of pairs of naturals.

lemma *eq-Abs-Integ* [*case-names Abs-Integ*, *cases type*: *int*]:
(!!$x\ y.\ z = Abs\text{-}Integ(intrel``\{(x,y)\}) ==> P) ==> P$
⟨*proof*⟩

23.3 Arithmetic Operations

lemma *minus*: $-\ Abs\text{-}Integ(intrel``\{(x,y)\}) = Abs\text{-}Integ(intrel\ ``\ \{(y,x)\})$
⟨*proof*⟩

lemma *add*:
$Abs\text{-}Integ\ (intrel``\{(x,y)\}) + Abs\text{-}Integ\ (intrel``\{(u,v)\}) =$
$Abs\text{-}Integ\ (intrel``\{(x+u,\ y+v)\})$
⟨*proof*⟩

Congruence property for multiplication

lemma *mult-congruent2*:
$(\%p1\ p2.\ (\%(x,y).\ (\%(u,v).\ intrel``\{(x*u + y*v,\ x*v + y*u)\})\ p2)\ p1)$
$respects2\ intrel$
⟨*proof*⟩

lemma *mult*:

Abs-Integ((*intrel*``{(*x*,*y*)})) ∗ *Abs-Integ*((*intrel*``{(*u*,*v*)})) =
Abs-Integ(*intrel* `` {(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)})
⟨*proof*⟩

The integers form a *comm-ring-1*

instance *int* :: *comm-ring-1*
⟨*proof*⟩

lemma *int-def*: *of-nat m* = *Abs-Integ* (*intrel* `` {(*m*, *0*)})
⟨*proof*⟩

23.4 The ≤ Ordering

lemma *le*:
(*Abs-Integ*(*intrel*``{(*x*,*y*)}) ≤ *Abs-Integ*(*intrel*``{(*u*,*v*)})) = (*x*+*v* ≤ *u*+*y*)
⟨*proof*⟩

lemma *less*:
(*Abs-Integ*(*intrel*``{(*x*,*y*)}) < *Abs-Integ*(*intrel*``{(*u*,*v*)})) = (*x*+*v* < *u*+*y*)
⟨*proof*⟩

instance *int* :: *linorder*
⟨*proof*⟩

instantiation *int* :: *distrib-lattice*
begin

definition
(*inf* :: *int* ⇒ *int* ⇒ *int*) = *min*

definition
(*sup* :: *int* ⇒ *int* ⇒ *int*) = *max*

instance
⟨*proof*⟩

end

instance *int* :: *pordered-cancel-ab-semigroup-add*
⟨*proof*⟩

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on k¿0

lemma *zmult-zless-mono2-lemma*:
(*i*::*int*)<*j* ==> *0*<*k* ==> *of-nat k* ∗ *i* < *of-nat k* ∗ *j*
⟨*proof*⟩

lemma *zero-le-imp-eq-int*: (*0*::*int*) ≤ *k* ==> ∃ *n*. *k* = *of-nat n*

⟨*proof*⟩

lemma *zero-less-imp-eq-int*: $(0::int) < k \Longrightarrow \exists n>0.\ k = of\text{-}nat\ n$
⟨*proof*⟩

lemma *zmult-zless-mono2*: $[|\ i<j;\ \ (0::int) < k\ |] \Longrightarrow k*i < k*j$
⟨*proof*⟩

The integers form an ordered integral domain

instance *int* :: *ordered-idom*
⟨*proof*⟩

instance *int* :: *lordered-ring*
⟨*proof*⟩

lemma *zless-imp-add1-zle*: $w < z \Longrightarrow w + (1::int) \leq z$
⟨*proof*⟩

lemma *zless-iff-Suc-zadd*:
$(w :: int) < z \longleftrightarrow (\exists n.\ z = w + of\text{-}nat\ (Suc\ n))$
⟨*proof*⟩

lemmas *int-distrib* =
left-distrib [*of z1*::*int z2 w*, *standard*]
right-distrib [*of w*::*int z1 z2*, *standard*]
left-diff-distrib [*of z1*::*int z2 w*, *standard*]
right-diff-distrib [*of w*::*int z1 z2*, *standard*]

23.5 Embedding of the Integers into any *ring-1*: *of-int*

context *ring-1*
begin

definition
of-int :: *int* ⇒ *'a*
where
[*code func del*]: *of-int z* = *contents* ($\bigcup(i, j) \in Rep\text{-}Integ\ z.$ { *of-nat i* − *of-nat j* })

lemma *of-int*: *of-int* (*Abs-Integ* (*intrel* `` {(*i*,*j*)})) = *of-nat i* − *of-nat j*
⟨*proof*⟩

lemma *of-int-0* [*simp*]: *of-int 0* = *0*
⟨*proof*⟩

lemma *of-int-1* [*simp*]: *of-int 1* = *1*
⟨*proof*⟩

lemma *of-int-add* [*simp*]: *of-int* (*w*+*z*) = *of-int w* + *of-int z*

⟨*proof*⟩

lemma *of-int-minus* [*simp*]: *of-int* (−*z*) = − (*of-int z*)
⟨*proof*⟩

lemma *of-int-diff* [*simp*]: *of-int* (*w* − *z*) = *of-int w* − *of-int z*
⟨*proof*⟩

lemma *of-int-mult* [*simp*]: *of-int* (*w*∗*z*) = *of-int w* ∗ *of-int z*
⟨*proof*⟩

Collapse nested embeddings

lemma *of-int-of-nat-eq* [*simp*]: *of-int* (*of-nat n*) = *of-nat n*
⟨*proof*⟩

end

context *ordered-idom*
begin

lemma *of-int-le-iff* [*simp*]:
of-int w ≤ *of-int z* ⟷ *w* ≤ *z*
⟨*proof*⟩

Special cases where either operand is zero

lemmas *of-int-0-le-iff* [*simp*] = *of-int-le-iff* [*of 0*, *simplified*]
lemmas *of-int-le-0-iff* [*simp*] = *of-int-le-iff* [*of - 0*, *simplified*]

lemma *of-int-less-iff* [*simp*]:
of-int w < *of-int z* ⟷ *w* < *z*
⟨*proof*⟩

Special cases where either operand is zero

lemmas *of-int-0-less-iff* [*simp*] = *of-int-less-iff* [*of 0*, *simplified*]
lemmas *of-int-less-0-iff* [*simp*] = *of-int-less-iff* [*of - 0*, *simplified*]

end

Class for unital rings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *ring-char-0* = *ring-1* + *semiring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]:
of-int w = *of-int z* ⟷ *w* = *z*
⟨*proof*⟩

Special cases where either operand is zero

lemmas *of-int-0-eq-iff* [*simp*] = *of-int-eq-iff* [*of 0*, *simplified*]
lemmas *of-int-eq-0-iff* [*simp*] = *of-int-eq-iff* [*of - 0*, *simplified*]

end

Every *ordered-idom* has characteristic zero.

subclass (**in** *ordered-idom*) *ring-char-0* ⟨*proof*⟩

lemma *of-int-eq-id* [*simp*]: *of-int* = *id*
⟨*proof*⟩

23.6 Magnitude of an Integer, as a Natural Number: *nat*

definition
nat :: *int* ⇒ *nat*
where
[*code func del*]: *nat z* = *contents* (⋃(*x*, *y*) ∈ *Rep-Integ z*. {*x*−*y*})

lemma *nat*: *nat* (*Abs-Integ* (*intrel*``{(*x*,*y*)})) = *x*−*y*
⟨*proof*⟩

lemma *nat-int* [*simp*]: *nat* (*of-nat n*) = *n*
⟨*proof*⟩

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

lemma *int-nat-eq* [*simp*]: *of-nat* (*nat z*) = (*if 0* ≤ *z then z else 0*)
⟨*proof*⟩

corollary *nat-0-le*: *0* ≤ *z* ==> *of-nat* (*nat z*) = *z*
⟨*proof*⟩

lemma *nat-le-0* [*simp*]: *z* ≤ *0* ==> *nat z* = *0*
⟨*proof*⟩

lemma *nat-le-eq-zle*: *0* < *w* | *0* ≤ *z* ==> (*nat w* ≤ *nat z*) = (*w*≤*z*)
⟨*proof*⟩

An alternative condition is (*0*::′*a*) ≤ *w*

corollary *nat-mono-iff*: *0* < *z* ==> (*nat w* < *nat z*) = (*w* < *z*)
⟨*proof*⟩

corollary *nat-less-eq-zless*: *0* ≤ *w* ==> (*nat w* < *nat z*) = (*w*<*z*)
⟨*proof*⟩

lemma *zless-nat-conj* [*simp*]: (*nat w* < *nat z*) = (*0* < *z* & *w* < *z*)
⟨*proof*⟩

lemma *nonneg-eq-int*:
fixes z :: *int*
assumes $0 \leq z$ **and** $\bigwedge m.\ z = \textit{of-nat}\ m \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *nat-eq-iff*: $(\textit{nat}\ w = m) = (\textit{if}\ 0 \leq w\ \textit{then}\ w = \textit{of-nat}\ m\ \textit{else}\ m{=}0)$
⟨*proof*⟩

corollary *nat-eq-iff2*: $(m = \textit{nat}\ w) = (\textit{if}\ 0 \leq w\ \textit{then}\ w = \textit{of-nat}\ m\ \textit{else}\ m{=}0)$
⟨*proof*⟩

lemma *nat-less-iff*: $0 \leq w \Longrightarrow (\textit{nat}\ w < m) = (w < \textit{of-nat}\ m)$
⟨*proof*⟩

lemma *int-eq-iff*: $(\textit{of-nat}\ m = z) = (m = \textit{nat}\ z\ \&\ 0 \leq z)$
⟨*proof*⟩

lemma *zero-less-nat-eq* [*simp*]: $(0 < \textit{nat}\ z) = (0 < z)$
⟨*proof*⟩

lemma *nat-add-distrib*:
$[\!|\ (0{::}\textit{int}) \leq z;\ \ 0 \leq z'\ |\!] \Longrightarrow \textit{nat}\ (z{+}z') = \textit{nat}\ z + \textit{nat}\ z'$
⟨*proof*⟩

lemma *nat-diff-distrib*:
$[\!|\ (0{::}\textit{int}) \leq z';\ \ z' \leq z\ |\!] \Longrightarrow \textit{nat}\ (z{-}z') = \textit{nat}\ z - \textit{nat}\ z'$
⟨*proof*⟩

lemma *nat-zminus-int* [*simp*]: $\textit{nat}\ (-\ (\textit{of-nat}\ n)) = 0$
⟨*proof*⟩

lemma *zless-nat-eq-int-zless*: $(m < \textit{nat}\ z) = (\textit{of-nat}\ m < z)$
⟨*proof*⟩

context *ring-1*
begin

lemma *of-nat-nat*: $0 \leq z \Longrightarrow \textit{of-nat}\ (\textit{nat}\ z) = \textit{of-int}\ z$
⟨*proof*⟩

end

23.7 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: $-\ (\textit{of-nat}\ (\textit{Suc}\ n)) < (0 :: \textit{int})$
⟨*proof*⟩

lemma *negative-zless* [*iff*]: $-\ (\textit{of-nat}\ (\textit{Suc}\ n)) < (\textit{of-nat}\ m :: \textit{int})$

⟨*proof*⟩

lemma *negative-zle-0*: − *of-nat n* ≤ (*0* :: *int*)
⟨*proof*⟩

lemma *negative-zle* [*iff*]: − *of-nat n* ≤ (*of-nat m* :: *int*)
⟨*proof*⟩

lemma *not-zle-0-negative* [*simp*]: ~ (*0* ≤ − (*of-nat* (*Suc n*) :: *int*))
⟨*proof*⟩

lemma *int-zle-neg*: ((*of-nat n* :: *int*) ≤ − *of-nat m*) = (*n* = *0* & *m* = *0*)
⟨*proof*⟩

lemma *not-int-zless-negative* [*simp*]: ~ ((*of-nat n* :: *int*) < − *of-nat m*)
⟨*proof*⟩

lemma *negative-eq-positive* [*simp*]: ((− *of-nat n* :: *int*) = *of-nat m*) = (*n* = *0* & *m* = *0*)
⟨*proof*⟩

lemma *zle-iff-zadd*: (*w*::*int*) ≤ *z* ⟷ (∃ *n*. *z* = *w* + *of-nat n*)
⟨*proof*⟩

lemma *zadd-int-left*: *of-nat m* + (*of-nat n* + *z*) = *of-nat* (*m* + *n*) + (*z*::*int*)
⟨*proof*⟩

lemma *int-Suc0-eq-1*: *of-nat* (*Suc 0*) = (*1*::*int*)
⟨*proof*⟩

This version is proved for all ordered rings, not just integers! It is proved here because attribute *arith-split* is not available in theory *Ring-and-Field*. But is it really better than just rewriting with *abs-if*?

lemma *abs-split* [*arith-split*,*noatp*]:
P(*abs*(*a*::'*a*::*ordered-idom*)) = ((*0* ≤ *a* −−> *P a*) & (*a* < *0* −−> *P*(−*a*)))
⟨*proof*⟩

lemma *negD*: (*x* :: *int*) < *0* ⟹ ∃ *n*. *x* = − (*of-nat* (*Suc n*))
⟨*proof*⟩

23.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

theorem *int-cases* [*cases type*: *int*, *case-names nonneg neg*]:
[|!! *n*. (*z* :: *int*) = *of-nat n* ==> *P*; !! *n*. *z* = − (*of-nat* (*Suc n*)) ==> *P* |] ==> *P*
⟨*proof*⟩

theorem *int-induct* [*induct type*: *int*, *case-names nonneg neg*]:
[|!! *n*. *P* (*of-nat n* :: *int*); !!*n*. *P* (− (*of-nat* (*Suc n*))) |] ==> *P z*
⟨*proof*⟩

Contributed by Brian Huffman

theorem *int-diff-cases*:
obtains (*diff*) *m n* **where** (*z*::*int*) = *of-nat m* − *of-nat n*
⟨*proof*⟩

23.9 Binary representation

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Tools/twos−compl.ML*, function *int-of-binary*, for the numerical interpretation.

The representation expects that (*m mod 2*) is 0 or 1, even if m is negative; For instance, *−5 div 2* = *−3* and *−5 mod 2* = *1*; thus *−5* = (*−3*)∗*2* + *1*.

This two's complement binary representation derives from the paper "An Efficient Representation of Arithmetic for Term Rewriting" by Dave Cohen and Phil Watson, Rewriting Techniques and Applications, Springer LNCS 488 (240-251), 1991.

definition
Pls :: *int* **where**
[*code func del*]: *Pls* = *0*

definition
Min :: *int* **where**
[*code func del*]: *Min* = − *1*

definition
Bit0 :: *int* ⇒ *int* **where**
[*code func del*]: *Bit0 k* = *k* + *k*

definition
Bit1 :: *int* ⇒ *int* **where**
[*code func del*]: *Bit1 k* = *1* + *k* + *k*

class *number* = *type* + — for numeric types: nat, int, real, ...
fixes *number-of* :: *int* ⇒ *'a*

⟨*ML*⟩

syntax
-Numeral :: *num-const* ⇒ *'a* (-)

⟨*ML*⟩

abbreviation
Numeral0 ≡ *number-of Pls*

abbreviation
Numeral1 ≡ *number-of* (*Bit1 Pls*)

lemma *Let-number-of* [*simp*]: *Let* (*number-of v*) *f* = *f* (*number-of v*)
— Unfold all *let*s involving constants
⟨*proof*⟩

definition
succ :: *int* ⇒ *int* **where**
[*code func del*]: *succ k* = *k* + *1*

definition
pred :: *int* ⇒ *int* **where**
[*code func del*]: *pred k* = *k* − *1*

lemmas
max-number-of [*simp*] = *max-def*
[*of number-of u number-of v*, *standard*, *simp*]
and
min-number-of [*simp*] = *min-def*
[*of number-of u number-of v*, *standard*, *simp*]
— unfolding *minx* and *max* on numerals

lemmas *numeral-simps* =
succ-def pred-def Pls-def Min-def Bit0-def Bit1-def

Removal of leading zeroes

lemma *Bit0-Pls* [*simp*, *code post*]:
Bit0 Pls = *Pls*
⟨*proof*⟩

lemma *Bit1-Min* [*simp*, *code post*]:
Bit1 Min = *Min*
⟨*proof*⟩

lemmas *normalize-bin-simps* =
Bit0-Pls Bit1-Min

23.10 The Functions *succ*, *pred* and *uminus*

lemma *succ-Pls* [*simp*]:
succ Pls = *Bit1 Pls*
⟨*proof*⟩

lemma *succ-Min* [*simp*]:

succ Min = Pls
⟨*proof*⟩

lemma *succ-Bit0* [*simp*]:
succ (Bit0 k) = Bit1 k
⟨*proof*⟩

lemma *succ-Bit1* [*simp*]:
succ (Bit1 k) = Bit0 (succ k)
⟨*proof*⟩

lemmas *succ-bin-simps =*
succ-Pls succ-Min succ-Bit0 succ-Bit1

lemma *pred-Pls* [*simp*]:
pred Pls = Min
⟨*proof*⟩

lemma *pred-Min* [*simp*]:
pred Min = Bit0 Min
⟨*proof*⟩

lemma *pred-Bit0* [*simp*]:
pred (Bit0 k) = Bit1 (pred k)
⟨*proof*⟩

lemma *pred-Bit1* [*simp*]:
pred (Bit1 k) = Bit0 k
⟨*proof*⟩

lemmas *pred-bin-simps =*
pred-Pls pred-Min pred-Bit0 pred-Bit1

lemma *minus-Pls* [*simp*]:
− Pls = Pls
⟨*proof*⟩

lemma *minus-Min* [*simp*]:
− Min = Bit1 Pls
⟨*proof*⟩

lemma *minus-Bit0* [*simp*]:
− (Bit0 k) = Bit0 (− k)
⟨*proof*⟩

lemma *minus-Bit1* [*simp*]:
− (Bit1 k) = Bit1 (pred (− k))
⟨*proof*⟩

lemmas *minus-bin-simps* =
minus-Pls minus-Min minus-Bit0 minus-Bit1

23.11 Binary Addition and Multiplication: *op* + and *op* ∗

lemma *add-Pls* [*simp*]:
$Pls + k = k$
⟨*proof*⟩

lemma *add-Min* [*simp*]:
$Min + k = pred\ k$
⟨*proof*⟩

lemma *add-Bit0-Bit0* [*simp*]:
$(Bit0\ k) + (Bit0\ l) = Bit0\ (k + l)$
⟨*proof*⟩

lemma *add-Bit0-Bit1* [*simp*]:
$(Bit0\ k) + (Bit1\ l) = Bit1\ (k + l)$
⟨*proof*⟩

lemma *add-Bit1-Bit0* [*simp*]:
$(Bit1\ k) + (Bit0\ l) = Bit1\ (k + l)$
⟨*proof*⟩

lemma *add-Bit1-Bit1* [*simp*]:
$(Bit1\ k) + (Bit1\ l) = Bit0\ (k + succ\ l)$
⟨*proof*⟩

lemma *add-Pls-right* [*simp*]:
$k + Pls = k$
⟨*proof*⟩

lemma *add-Min-right* [*simp*]:
$k + Min = pred\ k$
⟨*proof*⟩

lemmas *add-bin-simps* =
add-Pls add-Min add-Pls-right add-Min-right
add-Bit0-Bit0 add-Bit0-Bit1 add-Bit1-Bit0 add-Bit1-Bit1

lemma *mult-Pls* [*simp*]:
$Pls * w = Pls$
⟨*proof*⟩

lemma *mult-Min* [*simp*]:
$Min * k = - k$
⟨*proof*⟩

lemma *mult-Bit0* [*simp*]:
(*Bit0 k*) ∗ *l* = *Bit0* (*k* ∗ *l*)
⟨*proof*⟩

lemma *mult-Bit1* [*simp*]:
(*Bit1 k*) ∗ *l* = (*Bit0* (*k* ∗ *l*)) + *l*
⟨*proof*⟩

lemmas *mult-bin-simps* =
mult-Pls mult-Min mult-Bit0 mult-Bit1

23.12 Converting Numerals to Rings: *number-of*

class *number-ring* = *number* + *comm-ring-1* +
assumes *number-of-eq*: *number-of k* = *of-int k*

self-embedding of the integers

instantiation *int* :: *number-ring*
begin

definition
int-number-of-def [*code func del*]: *number-of w* = (*of-int w* :: *int*)

instance
⟨*proof*⟩

end

lemma *number-of-is-id*:
number-of (*k*::*int*) = *k*
⟨*proof*⟩

lemma *number-of-succ*:
number-of (*succ k*) = (*1* + *number-of k* ::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-pred*:
number-of (*pred w*) = (− *1* + *number-of w* ::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-minus*:
number-of (*uminus w*) = (− (*number-of w*)::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-add*:
number-of (*v* + *w*) = (*number-of v* + *number-of w*::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-mult*:

number-of $(v * w) = (\mathit{number\text{-}of}\ v * \mathit{number\text{-}of}\ w::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

lemma *double-number-of-Bit0*:
$(1 + 1) * \mathit{number\text{-}of}\ w = (\mathit{number\text{-}of}\ (\mathit{Bit0}\ w)::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

Converting numerals 0 and 1 to their abstract versions.

lemma *numeral-0-eq-0* [*simp*]:
$\mathit{Numeral0} = (0::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

lemma *numeral-1-eq-1* [*simp*]:
$\mathit{Numeral1} = (1::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

Special-case simplification for small constants.

Unary minus for the abstract constant 1. Cannot be inserted as a simprule until later: it is *number-of-Min* re-oriented!

lemma *numeral-m1-eq-minus-1*:
$(-1::'a::\mathit{number\text{-}ring}) = -\ 1$
⟨*proof*⟩

lemma *mult-minus1* [*simp*]:
$-1 * z = -(z::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

lemma *mult-minus1-right* [*simp*]:
$z * -1 = -(z::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

lemma *minus-number-of-mult* [*simp*]:
$-\ (\mathit{number\text{-}of}\ w) * z = \mathit{number\text{-}of}\ (\mathit{uminus}\ w) * (z::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

Subtraction

lemma *diff-number-of-eq*:
$\mathit{number\text{-}of}\ v - \mathit{number\text{-}of}\ w =$
$(\mathit{number\text{-}of}\ (v + \mathit{uminus}\ w)::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

lemma *number-of-Pls*:
$\mathit{number\text{-}of}\ \mathit{Pls} = (0::'a::\mathit{number\text{-}ring})$
⟨*proof*⟩

lemma *number-of-Min*:
number-of Min = (− *1*::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-Bit0*:
number-of (*Bit0 w*) = (*0*::'*a*::*number-ring*) + (*number-of w*) + (*number-of w*)
⟨*proof*⟩

lemma *number-of-Bit1*:
number-of (*Bit1 w*) = (*1*::'*a*::*number-ring*) + (*number-of w*) + (*number-of w*)
⟨*proof*⟩

23.13 Equality of Binary Numbers

First version by Norbert Voelker

definition
neg :: '*a*::*ordered-idom* ⇒ *bool*
where
neg Z ⟷ *Z* < *0*

definition
iszero :: '*a*::*semiring-1* ⇒ *bool*
where
iszero z ⟷ *z* = *0*

lemma *not-neg-int* [*simp*]: ~ *neg* (*of-nat n*)
⟨*proof*⟩

lemma *neg-zminus-int* [*simp*]: *neg* (− (*of-nat* (*Suc n*)))
⟨*proof*⟩

lemmas *neg-eq-less-0* = *neg-def*

lemma *not-neg-eq-ge-0*: (~*neg x*) = (*0* ≤ *x*)
⟨*proof*⟩

To simplify inequalities when Numeral1 can get simplified to 1

lemma *not-neg-0*: ~ *neg 0*
⟨*proof*⟩

lemma *not-neg-1*: ~ *neg 1*
⟨*proof*⟩

lemma *iszero-0*: *iszero 0*
⟨*proof*⟩

lemma *not-iszero-1*: ~ *iszero 1*
⟨*proof*⟩

lemma *neg-nat*: *neg z ==> nat z = 0*
⟨*proof*⟩

lemma *not-neg-nat*: ~ *neg z ==> of-nat (nat z) = z*
⟨*proof*⟩

lemma *eq-number-of-eq*:
((*number-of x*::'*a*::*number-ring*) = *number-of y*) =
iszero (*number-of* (*x* + *uminus y*) :: '*a*)
⟨*proof*⟩

lemma *iszero-number-of-Pls*:
iszero ((*number-of Pls*)::'*a*::*number-ring*)
⟨*proof*⟩

lemma *nonzero-number-of-Min*:
~ *iszero* ((*number-of Min*)::'*a*::*number-ring*)
⟨*proof*⟩

23.14 Comparisons, for Ordered Rings

lemmas *double-eq-0-iff* = *double-zero*

lemma *le-imp-0-less*:
assumes *le*: *0* ≤ *z*
shows (*0*::*int*) < *1* + *z*
⟨*proof*⟩

lemma *odd-nonzero*:
1 + *z* + *z* ≠ (*0*::*int*)
⟨*proof*⟩

lemma *iszero-number-of-Bit0*:
iszero (*number-of* (*Bit0 w*)::'*a*) =
iszero (*number-of w*::'*a*::{*ring-char-0*,*number-ring*})
⟨*proof*⟩

lemma *iszero-number-of-Bit1*:
~ *iszero* (*number-of* (*Bit1 w*)::'*a*::{*ring-char-0*,*number-ring*})
⟨*proof*⟩

23.15 The Less-Than Relation

lemma *less-number-of-eq-neg*:
((*number-of x*::'*a*::{*ordered-idom*,*number-ring*}) < *number-of y*)
= *neg* (*number-of* (*x* + *uminus y*) :: '*a*)
⟨*proof*⟩

If *Numeral0* is rewritten to 0 then this rule can't be applied: *Numeral0* IS

Numeral0

lemma *not-neg-number-of-Pls*:
~ *neg* (*number-of Pls* ::'*a*::{*ordered-idom*,*number-ring*})
⟨*proof*⟩

lemma *neg-number-of-Min*:
neg (*number-of Min* ::'*a*::{*ordered-idom*,*number-ring*})
⟨*proof*⟩

lemma *double-less-0-iff*:
(*a* + *a* < *0*) = (*a* < (*0*::'*a*::*ordered-idom*))
⟨*proof*⟩

lemma *odd-less-0*:
(*1* + *z* + *z* < *0*) = (*z* < (*0*::*int*))
⟨*proof*⟩

lemma *neg-number-of-Bit0*:
neg (*number-of* (*Bit0 w*)::'*a*) =
neg (*number-of w* :: '*a*::{*ordered-idom*,*number-ring*})
⟨*proof*⟩

lemma *neg-number-of-Bit1*:
neg (*number-of* (*Bit1 w*)::'*a*) =
neg (*number-of w* :: '*a*::{*ordered-idom*,*number-ring*})
⟨*proof*⟩

Less-Than or Equals

Reduces $a \leq b$ to $\neg\ b < a$ for ALL numerals.

lemmas *le-number-of-eq-not-less* =
linorder-not-less [*of number-of w number-of v*, *symmetric*,
standard]

lemma *le-number-of-eq*:
((*number-of x*::'*a*::{*ordered-idom*,*number-ring*}) ≤ *number-of y*)
= (~ (*neg* (*number-of* (*y* + *uminus x*) :: '*a*)))
⟨*proof*⟩

Absolute value (*abs*)

lemma *abs-number-of*:
abs(*number-of x*::'*a*::{*ordered-idom*,*number-ring*}) =
(*if number-of x* < (*0*::'*a*) *then* −*number-of x else number-of x*)
⟨*proof*⟩

Re-orientation of the equation nnn=x

lemma *number-of-reorient*:
(*number-of w* = *x*) = (*x* = *number-of w*)
⟨*proof*⟩

23.16 Simplification of arithmetic operations on integer constants.

lemmas *arith-extra-simps* [*standard*, *simp*] =
number-of-add [*symmetric*]
number-of-minus [*symmetric*] *numeral-m1-eq-minus-1* [*symmetric*]
number-of-mult [*symmetric*]
diff-number-of-eq abs-number-of

For making a minimal simpset, one must include these default simprules. Also include *simp-thms*.

lemmas *arith-simps* =
normalize-bin-simps pred-bin-simps succ-bin-simps
add-bin-simps minus-bin-simps mult-bin-simps
abs-zero abs-one arith-extra-simps

Simplification of relational operations

lemmas *rel-simps* [*simp*] =
eq-number-of-eq iszero-0 nonzero-number-of-Min
iszero-number-of-Bit0 iszero-number-of-Bit1
less-number-of-eq-neg
not-neg-number-of-Pls not-neg-0 not-neg-1 not-iszero-1
neg-number-of-Min neg-number-of-Bit0 neg-number-of-Bit1
le-number-of-eq

23.17 Simplification of arithmetic when nested to the right.

lemma *add-number-of-left* [*simp*]:
number-of v + (*number-of* w + z) =
(*number-of*(v + w) + z::$'a$::*number-ring*)
⟨*proof*⟩

lemma *mult-number-of-left* [*simp*]:
number-of v ∗ (*number-of* w ∗ z) =
(*number-of*(v ∗ w) ∗ z::$'a$::*number-ring*)
⟨*proof*⟩

lemma *add-number-of-diff1*:
number-of v + (*number-of* w − c) =
number-of(v + w) − (c::$'a$::*number-ring*)
⟨*proof*⟩

lemma *add-number-of-diff2* [*simp*]:
number-of v + (c − *number-of* w) =
number-of (v + *uminus* w) + (c::$'a$::*number-ring*)
⟨*proof*⟩

23.18 The Set of Integers

context *ring-1*

begin

definition
Ints :: *'a set*
where
Ints = *range of-int*

end

notation (*xsymbols*)
Ints ($\mathbb{Z}$)

context *ring-1*
begin

lemma *Ints-0* [*simp*]: $0 \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-1* [*simp*]: $1 \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-add* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a + b \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-minus* [*simp*]: $a \in \mathbb{Z} \Longrightarrow -a \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-mult* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a * b \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-cases* [*cases set*: *Ints*]:
assumes $q \in \mathbb{Z}$
obtains (*of-int*) *z* **where** q = *of-int z*
⟨*proof*⟩

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
$q \in \mathbb{Z} \Longrightarrow (\bigwedge z.\ P\ (\textit{of-int}\ z)) \Longrightarrow P\ q$
⟨*proof*⟩

end

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a-b \in \mathbb{Z}$
⟨*proof*⟩

The premise involving $\mathbb{Z}$ prevents $a = (1::'a)\ /\ (2::'a)$.

lemma *Ints-double-eq-0-iff*:
assumes *in-Ints*: $a \in$ *Ints*
shows $(a + a = 0) = (a = (0::'a::\textit{ring-char-0}))$
⟨*proof*⟩

lemma *Ints-odd-nonzero*:
assumes *in-Ints*: $a \in Ints$
shows $1 + a + a \neq (0::'a::ring\text{-}char\text{-}0)$
⟨*proof*⟩

lemma *Ints-number-of*:
$(number\text{-}of\ w :: 'a::number\text{-}ring) \in Ints$
⟨*proof*⟩

lemma *Ints-odd-less-0*:
assumes *in-Ints*: $a \in Ints$
shows $(1 + a + a < 0) = (a < (0::'a::ordered\text{-}idom))$
⟨*proof*⟩

23.19 *setsum* and *setprod*

By Jeremy Avigad

lemma *of-nat-setsum*: $of\text{-}nat\ (setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}nat(f\ x))$
⟨*proof*⟩

lemma *of-int-setsum*: $of\text{-}int\ (setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}int(f\ x))$
⟨*proof*⟩

lemma *of-nat-setprod*: $of\text{-}nat\ (setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}nat(f\ x))$
⟨*proof*⟩

lemma *of-int-setprod*: $of\text{-}int\ (setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}int(f\ x))$
⟨*proof*⟩

lemma *setprod-nonzero-nat*:
$finite\ A \Longrightarrow (\forall x \in A.\ f\ x \neq (0::nat)) \Longrightarrow setprod\ f\ A \neq 0$
⟨*proof*⟩

lemma *setprod-zero-eq-nat*:
$finite\ A \Longrightarrow (setprod\ f\ A = (0::nat)) = (\exists x \in A.\ f\ x = 0)$
⟨*proof*⟩

lemma *setprod-nonzero-int*:
$finite\ A \Longrightarrow (\forall x \in A.\ f\ x \neq (0::int)) \Longrightarrow setprod\ f\ A \neq 0$
⟨*proof*⟩

lemma *setprod-zero-eq-int*:
$finite\ A \Longrightarrow (setprod\ f\ A = (0::int)) = (\exists x \in A.\ f\ x = 0)$
⟨*proof*⟩

lemmas *int-setsum* = *of-nat-setsum* [**where** *'a=int*]
lemmas *int-setprod* = *of-nat-setprod* [**where** *'a=int*]

23.20 Inequality Reasoning for the Arithmetic Simproc

lemma *add-numeral-0*: *Numeral0 + a = (a::'a::number-ring)*
⟨*proof*⟩

lemma *add-numeral-0-right*: *a + Numeral0 = (a::'a::number-ring)*
⟨*proof*⟩

lemma *mult-numeral-1*: *Numeral1 * a = (a::'a::number-ring)*
⟨*proof*⟩

lemma *mult-numeral-1-right*: *a * Numeral1 = (a::'a::number-ring)*
⟨*proof*⟩

lemma *divide-numeral-1*: *a / Numeral1 = (a::'a::{number-ring,field})*
⟨*proof*⟩

lemma *inverse-numeral-1*:
inverse Numeral1 = (Numeral1::'a::{number-ring,field})
⟨*proof*⟩

Theorem lists for the cancellation simprocs. The use of binary numerals for 0 and 1 reduces the number of special cases.

lemmas *add-0s = add-numeral-0 add-numeral-0-right*
lemmas *mult-1s = mult-numeral-1 mult-numeral-1-right*
mult-minus1 mult-minus1-right

23.21 Special Arithmetic Rules for Abstract 0 and 1

Arithmetic computations are defined for binary literals, which leaves 0 and 1 as special cases. Addition already has rules for 0, but not 1. Multiplication and unary minus already have rules for both 0 and 1.

lemma *binop-eq*: *[|f x y = g x y; x = x'; y = y'|] ==> f x' y' = g x' y'*
⟨*proof*⟩

lemmas *add-number-of-eq = number-of-add [symmetric]*

Allow 1 on either or both sides

lemma *one-add-one-is-two*: *1 + 1 = (2::'a::number-ring)*
⟨*proof*⟩

lemmas *add-special =*
one-add-one-is-two
binop-eq [of op +, OF add-number-of-eq numeral-1-eq-1 refl, standard]
binop-eq [of op +, OF add-number-of-eq refl numeral-1-eq-1, standard]

Allow 1 on either or both sides (1-1 already simplifies to 0)

lemmas *diff-special* =
binop-eq [*of op* −, *OF diff-number-of-eq numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op* −, *OF diff-number-of-eq refl numeral-1-eq-1*, *standard*]

Allow 0 or 1 on either side with a binary numeral on the other

lemmas *eq-special* =
binop-eq [*of op* =, *OF eq-number-of-eq numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op* =, *OF eq-number-of-eq numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op* =, *OF eq-number-of-eq refl numeral-0-eq-0*, *standard*]
binop-eq [*of op* =, *OF eq-number-of-eq refl numeral-1-eq-1*, *standard*]

Allow 0 or 1 on either side with a binary numeral on the other

lemmas *less-special* =
binop-eq [*of op* <, *OF less-number-of-eq-neg numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op* <, *OF less-number-of-eq-neg numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op* <, *OF less-number-of-eq-neg refl numeral-0-eq-0*, *standard*]
binop-eq [*of op* <, *OF less-number-of-eq-neg refl numeral-1-eq-1*, *standard*]

Allow 0 or 1 on either side with a binary numeral on the other

lemmas *le-special* =
binop-eq [*of op* ≤, *OF le-number-of-eq numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op* ≤, *OF le-number-of-eq numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op* ≤, *OF le-number-of-eq refl numeral-0-eq-0*, *standard*]
binop-eq [*of op* ≤, *OF le-number-of-eq refl numeral-1-eq-1*, *standard*]

lemmas *arith-special*[*simp*] =
add-special diff-special eq-special less-special le-special

lemma *min-max-01*: *min* (*0*::*int*) *1* = *0* & *min* (*1*::*int*) *0* = *0* &
max (*0*::*int*) *1* = *1* & *max* (*1*::*int*) *0* = *1*
⟨*proof*⟩

lemmas *min-max-special*[*simp*] =
min-max-01
max-def[*of 0*::*int number-of v*, *standard*, *simp*]
min-def[*of 0*::*int number-of v*, *standard*, *simp*]
max-def[*of number-of u 0*::*int*, *standard*, *simp*]
min-def[*of number-of u 0*::*int*, *standard*, *simp*]
max-def[*of 1*::*int number-of v*, *standard*, *simp*]
min-def[*of 1*::*int number-of v*, *standard*, *simp*]
max-def[*of number-of u 1*::*int*, *standard*, *simp*]
min-def[*of number-of u 1*::*int*, *standard*, *simp*]

Legacy theorems

lemmas *zle-int* = *of-nat-le-iff* [**where** *'a*=*int*]
lemmas *int-int-eq* = *of-nat-eq-iff* [**where** *'a*=*int*]

⟨*ML*⟩

23.22 Lemmas About Small Numerals

lemma *of-int-m1* [*simp*]: *of-int* $-1 = (-1 :: 'a :: number\text{-}ring)$
⟨*proof*⟩

lemma *abs-minus-one* [*simp*]: *abs* $(-1) = (1::'a::\{ordered\text{-}idom,number\text{-}ring\})$
⟨*proof*⟩

lemma *abs-power-minus-one* [*simp*]:
$abs(-1 \hat{}\ n) = (1::'a::\{ordered\text{-}idom,number\text{-}ring,recpower\})$
⟨*proof*⟩

lemma *of-int-number-of-eq*:
of-int $(number\text{-}of\ v) = (number\text{-}of\ v :: 'a :: number\text{-}ring)$
⟨*proof*⟩

Lemmas for specialist use, NOT as default simprules

lemma *mult-2*: $2 * z = (z+z::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *mult-2-right*: $z * 2 = (z+z::'a::number\text{-}ring)$
⟨*proof*⟩

23.23 More Inequality Reasoning

lemma *zless-add1-eq*: $(w < z + (1::int)) = (w<z \mid w=z)$
⟨*proof*⟩

lemma *add1-zle-eq*: $(w + (1::int) \le z) = (w<z)$
⟨*proof*⟩

lemma *zle-diff1-eq* [*simp*]: $(w \le z - (1::int)) = (w<z)$
⟨*proof*⟩

lemma *zle-add1-eq-le* [*simp*]: $(w < z + (1::int)) = (w\le z)$
⟨*proof*⟩

lemma *int-one-le-iff-zero-less*: $((1::int) \le z) = (0 < z)$
⟨*proof*⟩

23.24 The Functions *nat* and *int*

Simplify the terms *int* $(0::'a)$, *int* $(Suc\ 0)$ and $w + - z$

declare *Zero-int-def* [*symmetric*, *simp*]
declare *One-int-def* [*symmetric*, *simp*]

lemmas *diff-int-def-symmetric* = *diff-int-def* [*symmetric*, *simp*]

lemma *nat-0*: *nat* $0 = 0$

⟨*proof*⟩

lemma *nat-1*: *nat 1 = Suc 0*
⟨*proof*⟩

lemma *nat-2*: *nat 2 = Suc (Suc 0)*
⟨*proof*⟩

lemma *one-less-nat-eq* [*simp*]: *(Suc 0 < nat z) = (1 < z)*
⟨*proof*⟩

This simplifies expressions of the form *int n = z* where z is an integer literal.

lemmas *int-eq-iff-number-of* [*simp*] *= int-eq-iff* [*of - number-of v, standard*]

lemma *split-nat* [*arith-split*]:
P(nat(i::int)) = ((∀ n. i = of-nat n ⟶ P n) & (i < 0 ⟶ P 0))
(is *?P = (?L & ?R)***)**
⟨*proof*⟩

context *ring-1*
begin

lemma *of-int-of-nat*:
of-int k = (if k < 0 then − of-nat (nat (− k)) else of-nat (nat k))
⟨*proof*⟩

end

lemma *nat-mult-distrib*:
fixes *z z′* :: *int*
assumes *0 ≤ z*
shows *nat (z * z′) = nat z * nat z′*
⟨*proof*⟩

lemma *nat-mult-distrib-neg*: *z ≤ (0::int) ==> nat(z*z′) = nat(−z) * nat(−z′)*
⟨*proof*⟩

lemma *nat-abs-mult-distrib*: *nat (abs (w * z)) = nat (abs w) * nat (abs z)*
⟨*proof*⟩

23.25 Induction principles for int

Well-founded segments of the integers

definition
int-ge-less-than :: *int => (int * int) set*
where
int-ge-less-than d = {(z′,z). d ≤ z′ & z′ < z}

theorem *wf-int-ge-less-than*: *wf (int-ge-less-than d)*

⟨*proof*⟩

This variant looks odd, but is typical of the relations suggested by RankFinder.

definition
int-ge-less-than2 :: *int* => (*int* ∗ *int*) *set*
where
int-ge-less-than2 d = {(*z'*,*z*). *d* ≤ *z* & *z'* < *z*}

theorem *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2 d*)
⟨*proof*⟩

abbreviation
int :: *nat* ⇒ *int*
where
int ≡ *of-nat*

theorem *int-ge-induct* [*case-names base step*, *induct set*: *int*]:
fixes *i* :: *int*
assumes *ge*: *k* ≤ *i* **and**
base: *P k* **and**
step: ⋀*i*. *k* ≤ *i* ⟹ *P i* ⟹ *P* (*i* + *1*)
shows *P i*
⟨*proof*⟩

theorem *int-gr-induct* [*case-names base step*, *induct set*: *int*]:
assumes *gr*: *k* < (*i*::*int*) **and**
base: *P*(*k*+*1*) **and**
step: ⋀*i*. ⟦*k* < *i*; *P i*⟧ ⟹ *P*(*i*+*1*)
shows *P i*
⟨*proof*⟩

theorem *int-le-induct*[*consumes 1*,*case-names base step*]:
assumes *le*: *i* ≤ (*k*::*int*) **and**
base: *P*(*k*) **and**
step: ⋀*i*. ⟦*i* ≤ *k*; *P i*⟧ ⟹ *P*(*i* − *1*)
shows *P i*
⟨*proof*⟩

theorem *int-less-induct* [*consumes 1*,*case-names base step*]:
assumes *less*: (*i*::*int*) < *k* **and**
base: *P*(*k* − *1*) **and**
step: ⋀*i*. ⟦*i* < *k*; *P i*⟧ ⟹ *P*(*i* − *1*)
shows *P i*
⟨*proof*⟩

23.26 Intermediate value theorems

lemma *int-val-lemma*:
(∀ i<n::nat. abs(f(i+1) − f i) ≤ 1) −−>
f 0 ≤ k −−> k ≤ f n −−> (∃ i ≤ n. f i = (k::int))
⟨*proof*⟩

lemmas *nat0-intermed-int-val* = *int-val-lemma* [*rule-format* (*no-asm*)]

lemma *nat-intermed-int-val*:
[| ∀ i. m ≤ i & i < n −−> abs(f(i + 1::nat) − f i) ≤ 1; m < n;
f m ≤ k; k ≤ f n |] ==> ? i. m ≤ i & i ≤ n & f i = (k::int)
⟨*proof*⟩

23.27 Products and 1, by T. M. Rasmussen

lemma *zabs-less-one-iff* [*simp*]: (|z| < 1) = (z = (0::int))
⟨*proof*⟩

lemma *abs-zmult-eq-1*: (|m * n| = 1) ==> |m| = (1::int)
⟨*proof*⟩

lemma *pos-zmult-eq-1-iff-lemma*: (m * n = 1) ==> m = (1::int) | m = −1
⟨*proof*⟩

lemma *pos-zmult-eq-1-iff*: 0 < (m::int) ==> (m * n = 1) = (m = 1 & n = 1)
⟨*proof*⟩

lemma *zmult-eq-1-iff*: (m*n = (1::int)) = ((m = 1 & n = 1) | (m = −1 & n = −1))
⟨*proof*⟩

lemma *infinite-UNIV-int*: ~*finite*(*UNIV*::*int set*)
⟨*proof*⟩

23.28 Integer Powers

instantiation *int* :: *recpower*
begin

primrec *power-int* **where**
p ^ 0 = (1::int)
| p ^ (Suc n) = (p::int) * (p ^ n)

instance ⟨*proof*⟩

end

lemma *zpower-zadd-distrib*: x ^ (y + z) = ((x ^ y) * (x ^ z)::int)

⟨*proof*⟩

lemma *zpower-zpower*: $(x \hat{} y) \hat{} z = (x \hat{} (y * z)::int)$
⟨*proof*⟩

lemma *zero-less-zpower-abs-iff* [*simp*]:
$(0 < abs\ x \hat{} n) \longleftrightarrow (x \neq (0::int) \mid n = 0)$
⟨*proof*⟩

lemma *zero-le-zpower-abs* [*simp*]: $(0::int) \leq abs\ x \hat{} n$
⟨*proof*⟩

lemma *of-int-power*:
of-int $(z \hat{} n) = (of\text{-}int\ z \hat{} n :: 'a::\{recpower, ring\text{-}1\})$
⟨*proof*⟩

lemma *int-power*: *int* $(m \hat{} n) = (int\ m) \hat{} n$
⟨*proof*⟩

lemmas *zpower-int* = *int-power* [*symmetric*]

23.29 Configuration of the code generator

code-datatype *Pls Min Bit0 Bit1 number-of* :: *int* ⇒ *int*

lemmas *pred-succ-numeral-code* [*code func*] =
pred-bin-simps succ-bin-simps

lemmas *plus-numeral-code* [*code func*] =
add-bin-simps
arith-extra-simps(*1*) [**where** $'a$ = *int*]

lemmas *minus-numeral-code* [*code func*] =
minus-bin-simps
arith-extra-simps(*2*) [**where** $'a$ = *int*]
arith-extra-simps(*5*) [**where** $'a$ = *int*]

lemmas *times-numeral-code* [*code func*] =
mult-bin-simps
arith-extra-simps(*4*) [**where** $'a$ = *int*]

instantiation *int* :: *eq*
begin

definition [*code func del*]: *eq-class.eq k l* $\longleftrightarrow k - l = (0::int)$

instance ⟨*proof*⟩

end

lemma *eq-number-of-int-code* [*code func*]:
eq-class.eq (*number-of k* :: *int*) (*number-of l*) ⟷ *eq-class.eq k l*
⟨*proof*⟩

lemma *eq-int-code* [*code func*]:
eq-class.eq Int.Pls Int.Pls ⟷ *True*
eq-class.eq Int.Pls Int.Min ⟷ *False*
eq-class.eq Int.Pls (*Int.Bit0 k2*) ⟷ *eq-class.eq Int.Pls k2*
eq-class.eq Int.Pls (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq Int.Min Int.Pls ⟷ *False*
eq-class.eq Int.Min Int.Min ⟷ *True*
eq-class.eq Int.Min (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq Int.Min (*Int.Bit1 k2*) ⟷ *eq-class.eq Int.Min k2*
eq-class.eq (*Int.Bit0 k1*) *Int.Pls* ⟷ *eq-class.eq Int.Pls k1*
eq-class.eq (*Int.Bit1 k1*) *Int.Pls* ⟷ *False*
eq-class.eq (*Int.Bit0 k1*) *Int.Min* ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) *Int.Min* ⟷ *eq-class.eq Int.Min k1*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit0 k2*) ⟷ *eq-class.eq k1 k2*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit1 k2*) ⟷ *eq-class.eq k1 k2*
⟨*proof*⟩

lemma *less-eq-number-of-int-code* [*code func*]:
(*number-of k* :: *int*) ≤ *number-of l* ⟷ *k* ≤ *l*
⟨*proof*⟩

lemma *less-eq-int-code* [*code func*]:
Int.Pls ≤ *Int.Pls* ⟷ *True*
Int.Pls ≤ *Int.Min* ⟷ *False*
Int.Pls ≤ *Int.Bit0 k* ⟷ *Int.Pls* ≤ *k*
Int.Pls ≤ *Int.Bit1 k* ⟷ *Int.Pls* ≤ *k*
Int.Min ≤ *Int.Pls* ⟷ *True*
Int.Min ≤ *Int.Min* ⟷ *True*
Int.Min ≤ *Int.Bit0 k* ⟷ *Int.Min* < *k*
Int.Min ≤ *Int.Bit1 k* ⟷ *Int.Min* ≤ *k*
Int.Bit0 k ≤ *Int.Pls* ⟷ *k* ≤ *Int.Pls*
Int.Bit1 k ≤ *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit0 k ≤ *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit1 k ≤ *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit0 k1 ≤ *Int.Bit0 k2* ⟷ *k1* ≤ *k2*
Int.Bit0 k1 ≤ *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
Int.Bit1 k1 ≤ *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit1 k1 ≤ *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
⟨*proof*⟩

lemma *less-number-of-int-code* [*code func*]:
(*number-of k* :: *int*) < *number-of l* ⟷ *k* < *l*

⟨*proof*⟩

lemma *less-int-code* [*code func*]:
Int.Pls < *Int.Pls* ⟷ *False*
Int.Pls < *Int.Min* ⟷ *False*
Int.Pls < *Int.Bit0 k* ⟷ *Int.Pls* < *k*
Int.Pls < *Int.Bit1 k* ⟷ *Int.Pls* ≤ *k*
Int.Min < *Int.Pls* ⟷ *True*
Int.Min < *Int.Min* ⟷ *False*
Int.Min < *Int.Bit0 k* ⟷ *Int.Min* < *k*
Int.Min < *Int.Bit1 k* ⟷ *Int.Min* < *k*
Int.Bit0 k < *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit1 k < *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit0 k < *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit1 k < *Int.Min* ⟷ *k* < *Int.Min*
Int.Bit0 k1 < *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit0 k1 < *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
Int.Bit1 k1 < *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit1 k1 < *Int.Bit1 k2* ⟷ *k1* < *k2*
⟨*proof*⟩

definition
int-aux :: *nat* ⇒ *int* ⇒ *int* **where**
[*code func del*]: *int-aux* = *of-nat-aux*

lemmas *int-aux-code* = *of-nat-aux-code* [**where** *?'a* = *int*, *simplified int-aux-def* [*symmetric*], *code*]

lemma [*code*, *code unfold*, *code inline del*]:
of-nat n = *int-aux n 0*
⟨*proof*⟩

definition
nat-aux :: *int* ⇒ *nat* ⇒ *nat* **where**
nat-aux i n = *nat i* + *n*

lemma [*code*]:
nat-aux i n = (*if i* ≤ *0 then n else nat-aux* (*i* − *1*) (*Suc n*)) — tail recursive
⟨*proof*⟩

lemma [*code*]: *nat i* = *nat-aux i 0*
⟨*proof*⟩

hide (**open**) *const int-aux nat-aux*

lemma *zero-is-num-zero* [*code func*, *code inline*, *symmetric*, *code post*]:
(*0*::*int*) = *Numeral0*
⟨*proof*⟩

lemma *one-is-num-one* [*code func*, *code inline*, *symmetric*, *code post*]:
(*1*::*int*) = *Numeral1*
⟨*proof*⟩

code-modulename *SML*
Int Integer

code-modulename *OCaml*
Int Integer

code-modulename *Haskell*
Int Integer

types-code
int (*int*)
attach (*term-of*) ⟪
val term-of-int = HOLogic.mk-number HOLogic.intT;
⟫
attach (*test*) ⟪
fun gen-int i =
let val j = one-of [~*1*, *1*] ∗ *random-range 0 i*
in (*j*, *fn* () => *term-of-int j*) *end*;
⟫

⟨*ML*⟩

consts-code
number-of :: *int* ⇒ *int* ((-))
0 :: *int* (*0*)
1 :: *int* (*1*)
uminus :: *int* => *int* (~)
op + :: *int* => *int* => *int* ((- +/ -))
op ∗ :: *int* => *int* => *int* ((- */ -))
op ≤ :: *int* => *int* => *bool* ((- <=/ -))
op < :: *int* => *int* => *bool* ((- </ -))

quickcheck-params [*default-type* = *int*]

hide (**open**) *const Pls Min Bit0 Bit1 succ pred*

23.30 Legacy theorems

lemmas *zminus-zminus* = *minus-minus* [*of z*::*int*, *standard*]
lemmas *zminus-0* = *minus-zero* [**where** *'a*=*int*]
lemmas *zminus-zadd-distrib* = *minus-add-distrib* [*of z*::*int w*, *standard*]
lemmas *zadd-commute* = *add-commute* [*of z*::*int w*, *standard*]
lemmas *zadd-assoc* = *add-assoc* [*of z1*::*int z2 z3*, *standard*]
lemmas *zadd-left-commute* = *add-left-commute* [*of x*::*int y z*, *standard*]
lemmas *zadd-ac* = *zadd-assoc zadd-commute zadd-left-commute*

lemmas *zmult-ac = OrderedGroup.mult-ac*
lemmas *zadd-0 = OrderedGroup.add-0-left* [*of z::int*, *standard*]
lemmas *zadd-0-right = OrderedGroup.add-0-left* [*of z::int*, *standard*]
lemmas *zadd-zminus-inverse2 = left-minus* [*of z::int*, *standard*]
lemmas *zmult-zminus = mult-minus-left* [*of z::int w*, *standard*]
lemmas *zmult-commute = mult-commute* [*of z::int w*, *standard*]
lemmas *zmult-assoc = mult-assoc* [*of z1::int z2 z3*, *standard*]
lemmas *zadd-zmult-distrib = left-distrib* [*of z1::int z2 w*, *standard*]
lemmas *zadd-zmult-distrib2 = right-distrib* [*of w::int z1 z2*, *standard*]
lemmas *zdiff-zmult-distrib = left-diff-distrib* [*of z1::int z2 w*, *standard*]
lemmas *zdiff-zmult-distrib2 = right-diff-distrib* [*of w::int z1 z2*, *standard*]

lemmas *zmult-1 = mult-1-left* [*of z::int*, *standard*]
lemmas *zmult-1-right = mult-1-right* [*of z::int*, *standard*]

lemmas *zle-refl = order-refl* [*of w::int*, *standard*]
lemmas *zle-trans = order-trans* [**where** *'a=int* **and** *x=i* **and** *y=j* **and** *z=k*, *standard*]
lemmas *zle-anti-sym = order-antisym* [*of z::int w*, *standard*]
lemmas *zle-linear = linorder-linear* [*of z::int w*, *standard*]
lemmas *zless-linear = linorder-less-linear* [**where** *'a = int*]

lemmas *zadd-left-mono = add-left-mono* [*of i::int j k*, *standard*]
lemmas *zadd-strict-right-mono = add-strict-right-mono* [*of i::int j k*, *standard*]
lemmas *zadd-zless-mono = add-less-le-mono* [*of w'::int w z' z*, *standard*]

lemmas *int-0-less-1 = zero-less-one* [**where** *'a=int*]
lemmas *int-0-neq-1 = zero-neq-one* [**where** *'a=int*]

lemmas *inj-int = inj-of-nat* [**where** *'a=int*]
lemmas *zadd-int = of-nat-add* [**where** *'a=int*, *symmetric*]
lemmas *int-mult = of-nat-mult* [**where** *'a=int*]
lemmas *zmult-int = of-nat-mult* [**where** *'a=int*, *symmetric*]
lemmas *int-eq-0-conv = of-nat-eq-0-iff* [**where** *'a=int* **and** *m=n*, *standard*]
lemmas *zless-int = of-nat-less-iff* [**where** *'a=int*]
lemmas *int-less-0-conv = of-nat-less-0-iff* [**where** *'a=int* **and** *m=k*, *standard*]
lemmas *zero-less-int-conv = of-nat-0-less-iff* [**where** *'a=int*]
lemmas *zero-zle-int = of-nat-0-le-iff* [**where** *'a=int*]
lemmas *int-le-0-conv = of-nat-le-0-iff* [**where** *'a=int* **and** *m=n*, *standard*]
lemmas *int-0 = of-nat-0* [**where** *'a=int*]
lemmas *int-1 = of-nat-1* [**where** *'a=int*]
lemmas *int-Suc = of-nat-Suc* [**where** *'a=int*]
lemmas *abs-int-eq = abs-of-nat* [**where** *'a=int* **and** *n=m*, *standard*]
lemmas *of-int-int-eq = of-int-of-nat-eq* [**where** *'a=int*]
lemmas *zdiff-int = of-nat-diff* [**where** *'a=int*, *symmetric*]
lemmas *zless-le = less-int-def*
lemmas *int-eq-of-nat = TrueI*

end

24 FunDef: General recursive function definitions

theory *FunDef*
imports *Wellfounded*
uses
(*Tools/function-package/fundef-lib.ML*)
(*Tools/function-package/fundef-common.ML*)
(*Tools/function-package/inductive-wrap.ML*)
(*Tools/function-package/context-tree.ML*)
(*Tools/function-package/fundef-core.ML*)
(*Tools/function-package/sum-tree.ML*)
(*Tools/function-package/mutual.ML*)
(*Tools/function-package/pattern-split.ML*)
(*Tools/function-package/fundef-package.ML*)
(*Tools/function-package/auto-term.ML*)
(*Tools/function-package/induction-scheme.ML*)
(*Tools/function-package/measure-functions.ML*)
(*Tools/function-package/lexicographic-order.ML*)
(*Tools/function-package/fundef-datatype.ML*)
begin

Definitions with default value.

definition
THE-default :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
THE-default d P = (*if* $(\exists!x.\ P\ x)$ *then* $(THE\ x.\ P\ x)$ *else d*)

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
⟨*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*⟩

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$
where *is-measure-trivial*: $is\text{-}measure\ f$

⟨*ML*⟩

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*⟩

24.1 Setup for termination proofs

Rules for generating measure functions

lemma [*measure-function*]: *is-measure size*
⟨*proof*⟩

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (λ*p*. *f* (*fst p*))
⟨*proof*⟩
lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (λ*p*. *f* (*snd p*))
⟨*proof*⟩

lemma *termination-basic-simps*[*termination-simp*]:
x < (*y*::*nat*) ⟹ *x* < *y* + *z*
x < *z* ⟹ *x* < *y* + *z*
x ≤ *y* ⟹ *x* ≤ *y* + (*z*::*nat*)
x ≤ *z* ⟹ *x* ≤ *y* + (*z*::*nat*)
x < *y* ⟹ *x* ≤ (*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*⟩

end

25 IntDiv: The Division Operators div and mod; the Divides Relation dvd

theory *IntDiv*
imports *Int Divides FunDef*
begin

constdefs
quorem :: (*int*∗*int*) ∗ (*int*∗*int*) => *bool*
— definition of quotient and remainder
[*code func*]: *quorem* == %((*a*,*b*), (*q*,*r*)).
a = *b*∗*q* + *r* &
(*if* 0 < *b then* 0≤*r* & *r*<*b else b*<*r* & *r* ≤ 0)

adjust :: [*int*, *int*∗*int*] => *int*∗*int*
— for the division algorithm
[*code func*]: *adjust b* == %(*q*,*r*). *if* 0 ≤ *r*−*b then* (2∗*q* + 1, *r*−*b*)
else (2∗*q*, *r*)

algorithm for the case *a*≥0, *b*>0

function
posDivAlg :: *int* ⇒ *int* ⇒ *int* × *int*

where
posDivAlg a b =
(*if* (*a*<*b* | *b*≤*0*) *then* (*0*,*a*)
else adjust b (*posDivAlg a* (*2*∗*b*)))
⟨*proof*⟩
termination ⟨*proof*⟩

algorithm for the case $a<0$, $b>0$

function
negDivAlg :: *int* ⇒ *int* ⇒ *int* × *int*
where
negDivAlg a b =
(*if* (*0*≤*a*+*b* | *b*≤*0*) *then* (−*1*,*a*+*b*)
else adjust b (*negDivAlg a* (*2*∗*b*)))
⟨*proof*⟩
termination ⟨*proof*⟩

algorithm for the general case $b \neq (0::'a)$

constdefs
negateSnd :: *int*∗*int* => *int*∗*int*
[*code func*]: *negateSnd* == %(*q*,*r*). (*q*,−*r*)

definition
divAlg :: *int* × *int* ⇒ *int* × *int*
— The full division algorithm considers all possible signs for a, b including the special case $a=0$, $b<0$ because *negDivAlg* requires $a < (0::'a)$.
where
divAlg = (λ(*a*, *b*). (*if* *0*≤*a then*
if *0*≤*b then posDivAlg a b*
else if *a*=*0 then* (*0*, *0*)
else negateSnd (*negDivAlg* (−*a*) (−*b*))
else
if *0*<*b then negDivAlg a b*
else negateSnd (*posDivAlg* (−*a*) (−*b*))))

instantiation *int* :: *Divides.div*
begin

definition
div-def: *a div b* = *fst* (*divAlg* (*a*, *b*))

definition
mod-def: *a mod b* = *snd* (*divAlg* (*a*, *b*))

instance ⟨*proof*⟩

end

lemma *divAlg-mod-div*:

divAlg $(p,\ q) = (p\ div\ q,\ p\ mod\ q)$
⟨*proof*⟩

Here is the division algorithm in ML:

```
fun posDivAlg (a,b) =
  if a<b then (0,a)
  else let val (q,r) = posDivAlg(a, 2*b)
           in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
       end

fun negDivAlg (a,b) =
  if 0\<le>a+b then (~1,a+b)
  else let val (q,r) = negDivAlg(a, 2*b)
           in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
       end;

fun negateSnd (q,r:int) = (q,~r);

fun divAlg (a,b) = if 0\<le>a then
                      if b>0 then posDivAlg (a,b)
                       else if a=0 then (0,0)
                            else negateSnd (negDivAlg (~a,~b))
                   else
                      if 0<b then negDivAlg (a,b)
                      else        negateSnd (posDivAlg (~a,~b));
```

25.1 Uniqueness and Monotonicity of Quotients and Remainders

lemma *unique-quotient-lemma*:
$[|\ b*q' + r'\ \le b*q + r;\ \ 0 \le r';\ \ r' < b;\ \ r < b\ |]$
$==>\ q' \le (q::int)$
⟨*proof*⟩

lemma *unique-quotient-lemma-neg*:
$[|\ b*q' + r' \le b*q + r;\ \ r \le 0;\ \ b < r;\ \ b < r'\ |]$
$==>\ q \le (q'::int)$
⟨*proof*⟩

lemma *unique-quotient*:
$[|\ quorem\ ((a,b),\ (q,r));\ \ quorem\ ((a,b),\ (q',r'));\ \ b \ne 0\ |]$
$==>\ q = q'$
⟨*proof*⟩

lemma *unique-remainder*:
[| *quorem* ((*a*,*b*), (*q*,*r*)); *quorem* ((*a*,*b*), (*q'*,*r'*)); *b* ≠ *0* |]
==> *r* = *r'*
⟨*proof*⟩

25.2 Correctness of *posDivAlg*, the Algorithm for Non-Negative Dividends

And positive divisors

lemma *adjust-eq* [*simp*]:
adjust b (*q*,*r*) =
(*let diff* = *r*−*b in*
if 0 ≤ *diff then* (2∗*q* + 1, *diff*)
else (2∗*q*, *r*))
⟨*proof*⟩

declare *posDivAlg.simps* [*simp del*]

use with a simproc to avoid repeatedly proving the premise

lemma *posDivAlg-eqn*:
0 < *b* ==>
posDivAlg a b = (*if a*<*b then* (*0*,*a*) *else adjust b* (*posDivAlg a* (*2*∗*b*)))
⟨*proof*⟩

Correctness of *posDivAlg*: it computes quotients correctly

theorem *posDivAlg-correct*:
assumes *0* ≤ *a* **and** *0* < *b*
shows *quorem* ((*a*, *b*), *posDivAlg a b*)
⟨*proof*⟩

25.3 Correctness of *negDivAlg*, the Algorithm for Negative Dividends

And positive divisors

declare *negDivAlg.simps* [*simp del*]

use with a simproc to avoid repeatedly proving the premise

lemma *negDivAlg-eqn*:
0 < *b* ==>
negDivAlg a b =
(*if* 0≤*a*+*b then* (−*1*,*a*+*b*) *else adjust b* (*negDivAlg a* (*2*∗*b*)))
⟨*proof*⟩

lemma *negDivAlg-correct*:
assumes *a* < *0* **and** *b* > *0*

shows *quorem* ((*a*, *b*), *negDivAlg a b*)
⟨*proof*⟩

25.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *quorem-0*: *b* ≠ *0* ==> *quorem* ((*0*,*b*), (*0*,*0*))
⟨*proof*⟩

lemma *posDivAlg-0* [*simp*]: *posDivAlg 0 b* = (*0*, *0*)
⟨*proof*⟩

lemma *negDivAlg-minus1* [*simp*]: *negDivAlg* −*1 b* = (−*1*, *b* − *1*)
⟨*proof*⟩

lemma *negateSnd-eq* [*simp*]: *negateSnd*(*q*,*r*) = (*q*,−*r*)
⟨*proof*⟩

lemma *quorem-neg*: *quorem* ((−*a*,−*b*), *qr*) ==> *quorem* ((*a*,*b*), *negateSnd qr*)
⟨*proof*⟩

lemma *divAlg-correct*: *b* ≠ *0* ==> *quorem* ((*a*,*b*), *divAlg* (*a*, *b*))
⟨*proof*⟩

Arbitrary definitions for division by zero. Useful to simplify certain equations.

lemma *DIVISION-BY-ZERO* [*simp*]: *a div* (*0*::*int*) = *0* & *a mod* (*0*::*int*) = *a*
⟨*proof*⟩

Basic laws about division and remainder

lemma *zmod-zdiv-equality*: (*a*::*int*) = *b* ∗ (*a div b*) + (*a mod b*)
⟨*proof*⟩

lemma *zdiv-zmod-equality*: (*b* ∗ (*a div b*) + (*a mod b*)) + *k* = (*a*::*int*)+*k*
⟨*proof*⟩

lemma *zdiv-zmod-equality2*: ((*a div b*) ∗ *b* + (*a mod b*)) + *k* = (*a*::*int*)+*k*
⟨*proof*⟩

Tool setup

⟨*ML*⟩

lemma *pos-mod-conj* : (*0*::*int*) < *b* ==> *0* ≤ *a mod b* & *a mod b* < *b*
⟨*proof*⟩

lemmas *pos-mod-sign* [*simp*] = *pos-mod-conj* [*THEN conjunct1*, *standard*]
and *pos-mod-bound* [*simp*] = *pos-mod-conj* [*THEN conjunct2*, *standard*]

lemma *neg-mod-conj* : *b* < (*0*::*int*) ==> *a mod b* ≤ *0* & *b* < *a mod b*

⟨*proof*⟩

lemmas *neg-mod-sign* [*simp*] = *neg-mod-conj* [*THEN conjunct1*, *standard*]
and *neg-mod-bound* [*simp*] = *neg-mod-conj* [*THEN conjunct2*, *standard*]

25.5 General Properties of div and mod

lemma *quorem-div-mod*: $b \neq 0 \Longrightarrow quorem\ ((a, b), (a\ div\ b, a\ mod\ b))$
⟨*proof*⟩

lemma *quorem-div*: $[|\ quorem((a,b),(q,r));\ b \neq 0\ |] \Longrightarrow a\ div\ b = q$
⟨*proof*⟩

lemma *quorem-mod*: $[|\ quorem((a,b),(q,r));\ b \neq 0\ |] \Longrightarrow a\ mod\ b = r$
⟨*proof*⟩

lemma *div-pos-pos-trivial*: $[|\ (0::int) \leq a;\ a < b\ |] \Longrightarrow a\ div\ b = 0$
⟨*proof*⟩

lemma *div-neg-neg-trivial*: $[|\ a \leq (0::int);\ b < a\ |] \Longrightarrow a\ div\ b = 0$
⟨*proof*⟩

lemma *div-pos-neg-trivial*: $[|\ (0::int) < a;\ a+b \leq 0\ |] \Longrightarrow a\ div\ b = -1$
⟨*proof*⟩

lemma *mod-pos-pos-trivial*: $[|\ (0::int) \leq a;\ a < b\ |] \Longrightarrow a\ mod\ b = a$
⟨*proof*⟩

lemma *mod-neg-neg-trivial*: $[|\ a \leq (0::int);\ b < a\ |] \Longrightarrow a\ mod\ b = a$
⟨*proof*⟩

lemma *mod-pos-neg-trivial*: $[|\ (0::int) < a;\ a+b \leq 0\ |] \Longrightarrow a\ mod\ b = a+b$
⟨*proof*⟩

There is no *mod-neg-pos-trivial*.

lemma *zdiv-zminus-zminus* [*simp*]: $(-a)\ div\ (-b) = a\ div\ (b::int)$
⟨*proof*⟩

lemma *zmod-zminus-zminus* [*simp*]: $(-a)\ mod\ (-b) = -\ (a\ mod\ (b::int))$
⟨*proof*⟩

25.6 Laws for div and mod with Unary Minus

lemma *zminus1-lemma*:
$quorem((a,b),(q,r))$
$\Longrightarrow quorem\ ((-a,b), (if\ r=0\ then\ -q\ else\ -q\ -\ 1),$

$(if\ r{=}0\ then\ 0\ else\ b{-}r))$

⟨*proof*⟩

lemma *zdiv-zminus1-eq-if*:
$b \neq (0::int)$
$\Longrightarrow (-a)\ div\ b =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ -\ (a\ div\ b) - 1)$
⟨*proof*⟩

lemma *zmod-zminus1-eq-if*:
$(-a::int)\ mod\ b = (if\ a\ mod\ b = 0\ then\ 0\ else\ b - (a\ mod\ b))$
⟨*proof*⟩

lemma *zdiv-zminus2*: $a\ div\ (-b) = (-a::int)\ div\ b$
⟨*proof*⟩

lemma *zmod-zminus2*: $a\ mod\ (-b) = -\ ((-a::int)\ mod\ b)$
⟨*proof*⟩

lemma *zdiv-zminus2-eq-if*:
$b \neq (0::int)$
$\Longrightarrow a\ div\ (-b) =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ -\ (a\ div\ b) - 1)$
⟨*proof*⟩

lemma *zmod-zminus2-eq-if*:
$a\ mod\ (-b::int) = (if\ a\ mod\ b = 0\ then\ 0\ else\ (a\ mod\ b) - b)$
⟨*proof*⟩

25.7 Division of a Number by Itself

lemma *self-quotient-aux1*: $[|\ (0::int) < a;\ a = r + a{*}q;\ r < a\ |] \Longrightarrow 1 \leq q$
⟨*proof*⟩

lemma *self-quotient-aux2*: $[|\ (0::int) < a;\ a = r + a{*}q;\ 0 \leq r\ |] \Longrightarrow q \leq 1$
⟨*proof*⟩

lemma *self-quotient*: $[|\ quorem((a,a),(q,r));\ \ a \neq (0::int)\ |] \Longrightarrow q = 1$
⟨*proof*⟩

lemma *self-remainder*: $[|\ quorem((a,a),(q,r));\ \ a \neq (0::int)\ |] \Longrightarrow r = 0$
⟨*proof*⟩

lemma *zdiv-self* [*simp*]: $a \neq 0 \Longrightarrow a\ div\ a = (1::int)$
⟨*proof*⟩

lemma *zmod-self* [*simp*]: $a\ mod\ a = (0::int)$

⟨*proof*⟩

25.8 Computation of Division and Remainder

lemma *zdiv-zero* [*simp*]: $(0::int)\ div\ b = 0$
⟨*proof*⟩

lemma *div-eq-minus1*: $(0::int) < b \Longrightarrow -1\ div\ b = -1$
⟨*proof*⟩

lemma *zmod-zero* [*simp*]: $(0::int)\ mod\ b = 0$
⟨*proof*⟩

lemma *zdiv-minus1*: $(0::int) < b \Longrightarrow -1\ div\ b = -1$
⟨*proof*⟩

lemma *zmod-minus1*: $(0::int) < b \Longrightarrow -1\ mod\ b = b - 1$
⟨*proof*⟩

a positive, b positive

lemma *div-pos-pos*: $[|\ 0 < a;\ \ 0 \le b\ |] \Longrightarrow a\ div\ b = fst\ (posDivAlg\ a\ b)$
⟨*proof*⟩

lemma *mod-pos-pos*: $[|\ 0 < a;\ \ 0 \le b\ |] \Longrightarrow a\ mod\ b = snd\ (posDivAlg\ a\ b)$
⟨*proof*⟩

a negative, b positive

lemma *div-neg-pos*: $[|\ a < 0;\ \ 0 < b\ |] \Longrightarrow a\ div\ b = fst\ (negDivAlg\ a\ b)$
⟨*proof*⟩

lemma *mod-neg-pos*: $[|\ a < 0;\ \ 0 < b\ |] \Longrightarrow a\ mod\ b = snd\ (negDivAlg\ a\ b)$
⟨*proof*⟩

a positive, b negative

lemma *div-pos-neg*:
$[|\ 0 < a;\ \ b < 0\ |] \Longrightarrow a\ div\ b = fst\ (negateSnd\ (negDivAlg\ (-a)\ (-b)))$
⟨*proof*⟩

lemma *mod-pos-neg*:
$[|\ 0 < a;\ \ b < 0\ |] \Longrightarrow a\ mod\ b = snd\ (negateSnd\ (negDivAlg\ (-a)\ (-b)))$
⟨*proof*⟩

a negative, b negative

lemma *div-neg-neg*:
$[|\ a < 0;\ \ b \le 0\ |] \Longrightarrow a\ div\ b = fst\ (negateSnd\ (posDivAlg\ (-a)\ (-b)))$
⟨*proof*⟩

lemma *mod-neg-neg*:
$[|\ a < 0;\ \ b \le 0\ |] \Longrightarrow a\ mod\ b = snd\ (negateSnd\ (posDivAlg\ (-a)\ (-b)))$

⟨*proof*⟩

Simplify expresions in which div and mod combine numerical constants

lemma *quoremI*:
⟦*a* == *b* ∗ *q* + *r*; *if* 0 < *b* *then* 0 ≤ *r* ∧ *r* < *b* *else* *b* < *r* ∧ *r* ≤ 0⟧
⟹ *quorem* ((*a*, *b*), (*q*, *r*))
⟨*proof*⟩

lemmas *quorem-div-eq* = *quoremI* [*THEN quorem-div*, *THEN eq-reflection*]
lemmas *quorem-mod-eq* = *quoremI* [*THEN quorem-mod*, *THEN eq-reflection*]
lemmas *arithmetic-simps* =
arith-simps
add-special
OrderedGroup.add-0-left
OrderedGroup.add-0-right
mult-zero-left
mult-zero-right
mult-1-left
mult-1-right

⟨*ML*⟩

lemmas *div-pos-pos-number-of* =
div-pos-pos [*of number-of v number-of w*, *standard*]

lemmas *div-neg-pos-number-of* =
div-neg-pos [*of number-of v number-of w*, *standard*]

lemmas *div-pos-neg-number-of* =
div-pos-neg [*of number-of v number-of w*, *standard*]

lemmas *div-neg-neg-number-of* =
div-neg-neg [*of number-of v number-of w*, *standard*]

lemmas *mod-pos-pos-number-of* =
mod-pos-pos [*of number-of v number-of w*, *standard*]

lemmas *mod-neg-pos-number-of* =
mod-neg-pos [*of number-of v number-of w*, *standard*]

lemmas *mod-pos-neg-number-of* =
mod-pos-neg [*of number-of v number-of w*, *standard*]

lemmas *mod-neg-neg-number-of* =
mod-neg-neg [*of number-of v number-of w*, *standard*]

lemmas *posDivAlg-eqn-number-of* [*simp*] =
posDivAlg-eqn [*of number-of v number-of w*, *standard*]

lemmas *negDivAlg-eqn-number-of* [*simp*] =
negDivAlg-eqn [*of number-of v number-of w*, *standard*]

Special-case simplification

lemma *zmod-1* [*simp*]: $a\ mod\ (1::int) = 0$
⟨*proof*⟩

lemma *zdiv-1* [*simp*]: $a\ div\ (1::int) = a$
⟨*proof*⟩

lemma *zmod-minus1-right* [*simp*]: $a\ mod\ (-1::int) = 0$
⟨*proof*⟩

lemma *zdiv-minus1-right* [*simp*]: $a\ div\ (-1::int) = -a$
⟨*proof*⟩

lemmas *div-pos-pos-1-number-of* [*simp*] =
div-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *div-pos-neg-1-number-of* [*simp*] =
div-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *mod-pos-pos-1-number-of* [*simp*] =
mod-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *mod-pos-neg-1-number-of* [*simp*] =
mod-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *posDivAlg-eqn-1-number-of* [*simp*] =
posDivAlg-eqn [*of concl*: *1 number-of w*, *standard*]

lemmas *negDivAlg-eqn-1-number-of* [*simp*] =
negDivAlg-eqn [*of concl*: *1 number-of w*, *standard*]

25.9 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*: $[|\ a \le a';\ \ 0 < (b::int)\ |] ==> a\ div\ b \le a'\ div\ b$
⟨*proof*⟩

lemma *zdiv-mono1-neg*: $[|\ a \le a';\ \ (b::int) < 0\ |] ==> a'\ div\ b \le a\ div\ b$
⟨*proof*⟩

25.10 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
[| $0 \le b'*q' + r'$; $r' < b'$; $0 < b'$ |] ==> $0 \le (q'::int)$
⟨*proof*⟩

lemma *zdiv-mono2-lemma*:
[| $b*q + r = b'*q' + r'$; $0 \le b'*q' + r'$;
$r' < b'$; $0 \le r$; $0 < b'$; $b' \le b$ |]
==> $q \le (q'::int)$
⟨*proof*⟩

lemma *zdiv-mono2*:
[| $(0::int) \le a$; $0 < b'$; $b' \le b$ |] ==> $a \ div \ b \le a \ div \ b'$
⟨*proof*⟩

lemma *q-neg-lemma*:
[| $b'*q' + r' < 0$; $0 \le r'$; $0 < b'$ |] ==> $q' \le (0::int)$
⟨*proof*⟩

lemma *zdiv-mono2-neg-lemma*:
[| $b*q + r = b'*q' + r'$; $b'*q' + r' < 0$;
$r < b$; $0 \le r'$; $0 < b'$; $b' \le b$ |]
==> $q' \le (q::int)$
⟨*proof*⟩

lemma *zdiv-mono2-neg*:
[| $a < (0::int)$; $0 < b'$; $b' \le b$ |] ==> $a \ div \ b' \le a \ div \ b$
⟨*proof*⟩

25.11 More Algebraic Laws for div and mod

proving (a*b) div c = a * (b div c) + a * (b mod c)

lemma *zmult1-lemma*:
[| $quorem((b,c),(q,r))$; $c \ne 0$ |]
==> $quorem\ ((a*b,\ c),\ (a*q + a*r \ div \ c,\ a*r \ mod \ c))$
⟨*proof*⟩

lemma *zdiv-zmult1-eq*: $(a*b) \ div \ c = a*(b \ div \ c) + a*(b \ mod \ c) \ div \ (c::int)$
⟨*proof*⟩

lemma *zmod-zmult1-eq*: $(a*b) \ mod \ c = a*(b \ mod \ c) \ mod \ (c::int)$
⟨*proof*⟩

lemma *zmod-zmult1-eq'*: $(a*b) \ mod \ (c::int) = ((a \ mod \ c) * b) \ mod \ c$
⟨*proof*⟩

lemma *zmod-zmult-distrib*: $(a*b) \ mod \ (c::int) = ((a \ mod \ c) * (b \ mod \ c)) \ mod \ c$
⟨*proof*⟩

lemma *zdiv-zmult-self1* [*simp*]: $b \neq (0::int) \Longrightarrow (a*b)\ div\ b = a$
⟨*proof*⟩

instance *int* :: *semiring-div*
⟨*proof*⟩

lemma *zdiv-zmult-self2* [*simp*]: $b \neq (0::int) \Longrightarrow (b*a)\ div\ b = a$
⟨*proof*⟩

lemma *zmod-zmult-self1* [*simp*]: $(a*b)\ mod\ b = (0::int)$
⟨*proof*⟩

lemma *zmod-zmult-self2* [*simp*]: $(b*a)\ mod\ b = (0::int)$
⟨*proof*⟩

lemma *zmod-eq-0-iff*: $(m\ mod\ d = 0) = (EX\ q::int.\ m = d*q)$
⟨*proof*⟩

lemmas *zmod-eq-0D* [*dest*!] = *zmod-eq-0-iff* [*THEN iffD1*]

proving (a+b) div c = a div c + b div c + ((a mod c + b mod c) div c)

lemma *zadd1-lemma*:
$[|\ quorem((a,c),(aq,ar));\ \ quorem((b,c),(bq,br));\ \ c \neq 0\ |]$
$\Longrightarrow quorem\ ((a+b,\ c),\ (aq\ +\ bq\ +\ (ar+br)\ div\ c,\ (ar+br)\ mod\ c))$
⟨*proof*⟩

lemma *zdiv-zadd1-eq*:
$(a+b)\ div\ (c::int) = a\ div\ c\ +\ b\ div\ c\ +\ ((a\ mod\ c\ +\ b\ mod\ c)\ div\ c)$
⟨*proof*⟩

lemma *zmod-zadd1-eq*: $(a+b)\ mod\ (c::int) = (a\ mod\ c\ +\ b\ mod\ c)\ mod\ c$
⟨*proof*⟩

lemma *mod-div-trivial* [*simp*]: $(a\ mod\ b)\ div\ b = (0::int)$
⟨*proof*⟩

lemma *mod-mod-trivial* [*simp*]: $(a\ mod\ b)\ mod\ b = a\ mod\ (b::int)$
⟨*proof*⟩

lemma *zmod-zadd-left-eq*: $(a+b)\ mod\ (c::int) = ((a\ mod\ c)\ +\ b)\ mod\ c$
⟨*proof*⟩

lemma *zmod-zadd-right-eq*: $(a+b)\ mod\ (c::int) = (a\ +\ (b\ mod\ c))\ mod\ c$
⟨*proof*⟩

lemma *zdiv-zadd-self1*[*simp*]: $a \neq (0::int) \Longrightarrow (a+b)\ div\ a = b\ div\ a\ +\ 1$
⟨*proof*⟩

lemma *zdiv-zadd-self2*[*simp*]: $a \neq (0::int) \Longrightarrow (b+a)\ div\ a = b\ div\ a + 1$
⟨*proof*⟩

lemma *zmod-zadd-self1*[*simp*]: $(a+b)\ mod\ a = b\ mod\ (a::int)$
⟨*proof*⟩

lemma *zmod-zadd-self2*[*simp*]: $(b+a)\ mod\ a = b\ mod\ (a::int)$
⟨*proof*⟩

lemma *zmod-zdiff1-eq*: **fixes** $a::int$
shows $(a - b)\ mod\ c = (a\ mod\ c - b\ mod\ c)\ mod\ c$ (**is** $?l = ?r$)
⟨*proof*⟩

25.12 Proving $a\ div\ (b * c) = a\ div\ b\ div\ c$

first, four lemmas to bound the remainder for the cases b¡0 and b¿0

lemma *zmult2-lemma-aux1*: $[|\ (0::int) < c;\ \ b < r;\ \ r \leq 0\ |] \Longrightarrow b*c < b*(q\ mod\ c) + r$
⟨*proof*⟩

lemma *zmult2-lemma-aux2*:
$[|\ (0::int) < c;\ \ b < r;\ \ r \leq 0\ |] \Longrightarrow b * (q\ mod\ c) + r \leq 0$
⟨*proof*⟩

lemma *zmult2-lemma-aux3*: $[|\ (0::int) < c;\ \ 0 \leq r;\ \ r < b\ |] \Longrightarrow 0 \leq b * (q\ mod\ c) + r$
⟨*proof*⟩

lemma *zmult2-lemma-aux4*: $[|\ (0::int) < c;\ 0 \leq r;\ r < b\ |] \Longrightarrow b * (q\ mod\ c) + r < b * c$
⟨*proof*⟩

lemma *zmult2-lemma*: $[|\ quorem\ ((a,b),\ (q,r));\ \ b \neq 0;\ \ 0 < c\ |]$
$\Longrightarrow quorem\ ((a,\ b*c),\ (q\ div\ c,\ b*(q\ mod\ c) + r))$
⟨*proof*⟩

lemma *zdiv-zmult2-eq*: $(0::int) < c \Longrightarrow a\ div\ (b*c) = (a\ div\ b)\ div\ c$
⟨*proof*⟩

lemma *zmod-zmult2-eq*:
$(0::int) < c \Longrightarrow a\ mod\ (b*c) = b*(a\ div\ b\ mod\ c) + a\ mod\ b$
⟨*proof*⟩

25.13 Cancellation of Common Factors in div

lemma *zdiv-zmult-zmult1-aux1*:
$[|\ (0::int) < b;\ \ c \neq 0\ |] \Longrightarrow (c*a)\ div\ (c*b) = a\ div\ b$

⟨*proof*⟩

lemma *zdiv-zmult-zmult1-aux2*:
[| $b < (0::int)$; $c \neq 0$ |] ==> $(c*a)\ div\ (c*b) = a\ div\ b$
⟨*proof*⟩

lemma *zdiv-zmult-zmult1*: $c \neq (0::int)$ ==> $(c*a)\ div\ (c*b) = a\ div\ b$
⟨*proof*⟩

lemma *zdiv-zmult-zmult1-if*[*simp*]:
$(k*m)\ div\ (k*n) = (if\ k = (0::int)\ then\ 0\ else\ m\ div\ n)$
⟨*proof*⟩

25.14 Distribution of Factors over mod

lemma *zmod-zmult-zmult1-aux1*:
[| $(0::int) < b$; $c \neq 0$ |] ==> $(c*a)\ mod\ (c*b) = c * (a\ mod\ b)$
⟨*proof*⟩

lemma *zmod-zmult-zmult1-aux2*:
[| $b < (0::int)$; $c \neq 0$ |] ==> $(c*a)\ mod\ (c*b) = c * (a\ mod\ b)$
⟨*proof*⟩

lemma *zmod-zmult-zmult1*: $(c*a)\ mod\ (c*b) = (c::int) * (a\ mod\ b)$
⟨*proof*⟩

lemma *zmod-zmult-zmult2*: $(a*c)\ mod\ (b*c) = (a\ mod\ b) * (c::int)$
⟨*proof*⟩

lemma *zmod-zmod-cancel*:
assumes $n\ dvd\ m$ **shows** $(k::int)\ mod\ m\ mod\ n = k\ mod\ n$
⟨*proof*⟩

25.15 Splitting Rules for div and mod

The proofs of the two lemmas below are essentially identical

lemma *split-pos-lemma*:
$0<k$ ==>
$P(n\ div\ k :: int)(n\ mod\ k) = (\forall i\ j.\ 0 \leq j\ \&\ j<k\ \&\ n = k*i + j\ --> P\ i\ j)$
⟨*proof*⟩

lemma *split-neg-lemma*:
$k<0$ ==>
$P(n\ div\ k :: int)(n\ mod\ k) = (\forall i\ j.\ k<j\ \&\ j \leq 0\ \&\ n = k*i + j\ --> P\ i\ j)$
⟨*proof*⟩

lemma *split-zdiv*:
$P(n\ div\ k :: int) =$
$((k = 0\ --> P\ 0)\ \&$

$(0<k \longrightarrow (\forall i\ j.\ 0\leq j\ \&\ j<k\ \&\ n = k*i + j \longrightarrow P\ i))\ \&$
$(k<0 \longrightarrow (\forall i\ j.\ k<j\ \&\ j\leq 0\ \&\ n = k*i + j \longrightarrow P\ i)))$
⟨*proof*⟩

lemma *split-zmod*:
$P(n\ mod\ k :: int) =$
$((k = 0 \longrightarrow P\ n)\ \&$
$(0<k \longrightarrow (\forall i\ j.\ 0\leq j\ \&\ j<k\ \&\ n = k*i + j \longrightarrow P\ j))\ \&$
$(k<0 \longrightarrow (\forall i\ j.\ k<j\ \&\ j\leq 0\ \&\ n = k*i + j \longrightarrow P\ j)))$
⟨*proof*⟩

declare *split-zdiv* [*of - - number-of k*, *simplified*, *standard*, *arith-split*]
declare *split-zmod* [*of - - number-of k*, *simplified*, *standard*, *arith-split*]

25.16 Speeding up the Division Algorithm with Shifting

computing div by shifting

lemma *pos-zdiv-mult-2*: $(0::int) \leq a \Longrightarrow (1 + 2*b)\ div\ (2*a) = b\ div\ a$
⟨*proof*⟩

lemma *neg-zdiv-mult-2*: $a \leq (0::int) \Longrightarrow (1 + 2*b)\ div\ (2*a) = (b+1)\ div\ a$
⟨*proof*⟩

lemma *not-0-le-lemma*: $\sim\ 0 \leq x \Longrightarrow x \leq (0::int)$
⟨*proof*⟩

lemma *zdiv-number-of-Bit0* [*simp*]:
$number\text{-}of\ (Int.Bit0\ v)\ div\ number\text{-}of\ (Int.Bit0\ w) =$
$number\text{-}of\ v\ div\ (number\text{-}of\ w :: int)$
⟨*proof*⟩

lemma *zdiv-number-of-Bit1* [*simp*]:
$number\text{-}of\ (Int.Bit1\ v)\ div\ number\text{-}of\ (Int.Bit0\ w) =$
$(if\ (0::int) \leq number\text{-}of\ w$
$then\ number\text{-}of\ v\ div\ (number\text{-}of\ w)$
$else\ (number\text{-}of\ v + (1::int))\ div\ (number\text{-}of\ w))$
⟨*proof*⟩

25.17 Computing mod by Shifting (proofs resemble those for div)

lemma *pos-zmod-mult-2*:
$(0::int) \leq a \Longrightarrow (1 + 2*b)\ mod\ (2*a) = 1 + 2 * (b\ mod\ a)$
⟨*proof*⟩

lemma *neg-zmod-mult-2*:
$a \leq (0::int) \Longrightarrow (1 + 2*b)\ mod\ (2*a) = 2 * ((b+1)\ mod\ a) - 1$

⟨*proof*⟩

lemma *zmod-number-of-Bit0* [*simp*]:
    number-of (*Int.Bit0 v*) *mod number-of* (*Int.Bit0 w*) =
    (2::*int*) ∗ (*number-of v mod number-of w*)
⟨*proof*⟩

lemma *zmod-number-of-Bit1* [*simp*]:
    number-of (*Int.Bit1 v*) *mod number-of* (*Int.Bit0 w*) =
    (*if* (0::*int*) ≤ *number-of w*
          then 2 ∗ (*number-of v mod number-of w*) + 1
          else 2 ∗ ((*number-of v* + (1::*int*)) *mod number-of w*) − 1)
⟨*proof*⟩

25.18 Quotients of Signs

lemma *div-neg-pos-less0*: [| *a* < (0::*int*); 0 < *b* |] ==> *a div b* < 0
⟨*proof*⟩

lemma *div-nonneg-neg-le0*: [| (0::*int*) ≤ *a*; *b* < 0 |] ==> *a div b* ≤ 0
⟨*proof*⟩

lemma *pos-imp-zdiv-nonneg-iff*: (0::*int*) < *b* ==> (0 ≤ *a div b*) = (0 ≤ *a*)
⟨*proof*⟩

lemma *neg-imp-zdiv-nonneg-iff*:
    b < (0::*int*) ==> (0 ≤ *a div b*) = (*a* ≤ (0::*int*))
⟨*proof*⟩

lemma *pos-imp-zdiv-neg-iff*: (0::*int*) < *b* ==> (*a div b* < 0) = (*a* < 0)
⟨*proof*⟩

lemma *neg-imp-zdiv-neg-iff*: *b* < (0::*int*) ==> (*a div b* < 0) = (0 < *a*)
⟨*proof*⟩

25.19 The Divides Relation

lemma *zdvd-iff-zmod-eq-0*: (*m dvd n*) = (*n mod m* = (0::*int*))
⟨*proof*⟩

lemmas *zdvd-iff-zmod-eq-0-number-of* [*simp*] =
zdvd-iff-zmod-eq-0 [*of number-of x number-of y*, *standard*]

lemma *zdvd-0-right* [*iff*]: (*m*::*int*) *dvd* 0
⟨*proof*⟩

lemma *zdvd-0-left* [*iff*,*noatp*]: (0 *dvd* (*m*::*int*)) = (*m* = 0)
⟨*proof*⟩

lemma *zdvd-1-left* [*iff*]: *1 dvd* (*m*::*int*)
⟨*proof*⟩

lemma *zdvd-refl* [*simp*]: *m dvd* (*m*::*int*)
⟨*proof*⟩

lemma *zdvd-trans*: *m dvd n* ==> *n dvd k* ==> *m dvd* (*k*::*int*)
⟨*proof*⟩

lemma *zdvd-zminus-iff*: (*m dvd* −*n*) = (*m dvd* (*n*::*int*))
⟨*proof*⟩

lemma *zdvd-zminus2-iff*: (−*m dvd n*) = (*m dvd* (*n*::*int*))
⟨*proof*⟩
lemma *zdvd-abs1*: (|*i*::*int*| *dvd j*) = (*i dvd j*)
⟨*proof*⟩
lemma *zdvd-abs2*: ((*i*::*int*) *dvd* |*j*|) = (*i dvd j*)
⟨*proof*⟩

lemma *zdvd-anti-sym*:
0 < *m* ==> *0* < *n* ==> *m dvd n* ==> *n dvd m* ==> *m* = (*n*::*int*)
⟨*proof*⟩

lemma *zdvd-zadd*: *k dvd m* ==> *k dvd n* ==> *k dvd* (*m* + *n* :: *int*)
⟨*proof*⟩

lemma *zdvd-dvd-eq*: **assumes** *anz*:*a* ≠ *0* **and** *ab*: (*a*::*int*) *dvd b* **and** *ba*:*b dvd a*
shows |*a*| = |*b*|
⟨*proof*⟩

lemma *zdvd-zdiff*: *k dvd m* ==> *k dvd n* ==> *k dvd* (*m* − *n* :: *int*)
⟨*proof*⟩

lemma *zdvd-zdiffD*: *k dvd m* − *n* ==> *k dvd n* ==> *k dvd* (*m*::*int*)
⟨*proof*⟩

lemma *zdvd-zmult*: *k dvd* (*n*::*int*) ==> *k dvd m* ∗ *n*
⟨*proof*⟩

lemma *zdvd-zmult2*: *k dvd* (*m*::*int*) ==> *k dvd m* ∗ *n*
⟨*proof*⟩

lemma *zdvd-triv-right* [*iff*]: (*k*::*int*) *dvd m* ∗ *k*
⟨*proof*⟩

lemma *zdvd-triv-left* [*iff*]: (*k*::*int*) *dvd k* ∗ *m*
⟨*proof*⟩

lemma *zdvd-zmultD2*: $j * k$ *dvd* $n ==> j$ *dvd* $(n::int)$
⟨*proof*⟩

lemma *zdvd-zmultD*: $j * k$ *dvd* $n ==> k$ *dvd* $(n::int)$
⟨*proof*⟩

lemma *zdvd-zmult-mono*: i *dvd* $m ==> j$ *dvd* $(n::int) ==> i * j$ *dvd* $m * n$
⟨*proof*⟩

lemma *zdvd-reduce*: $(k$ *dvd* $n + k * m) = (k$ *dvd* $(n::int))$
⟨*proof*⟩

lemma *zdvd-zmod*: f *dvd* $m ==> f$ *dvd* $(n::int) ==> f$ *dvd* m *mod* n
⟨*proof*⟩

lemma *zdvd-zmod-imp-zdvd*: k *dvd* m *mod* $n ==> k$ *dvd* $n ==> k$ *dvd* $(m::int)$
⟨*proof*⟩

lemma *zdvd-not-zless*: $0 < m ==> m < n ==> \neg\ n$ *dvd* $(m::int)$
⟨*proof*⟩
lemma *zmult-div-cancel*: $(n::int) * (m$ *div* $n) = m - (m$ *mod* $n)$
⟨*proof*⟩

lemma *zdvd-mult-div-cancel*:$(n::int)$ *dvd* $m \Longrightarrow n * (m$ *div* $n) = m$
⟨*proof*⟩

lemma *zdvd-mult-cancel*: **assumes** $d{:}k * m$ *dvd* $k * n$ **and** $kz{:}k \neq (0::int)$
shows m *dvd* n
⟨*proof*⟩

lemma *zdvd-zmult-cancel-disj*[*simp*]:
$(k*m)$ *dvd* $(k*n) = (k=0 \mid m$ *dvd* $(n::int))$
⟨*proof*⟩

theorem *ex-nat*: $(\exists\, x::nat.\ P\ x) = (\exists\, x::int.\ 0 <= x \wedge P\ (nat\ x))$
⟨*proof*⟩

theorem *zdvd-int*: $(x$ *dvd* $y) = (int\ x$ *dvd* $int\ y)$
⟨*proof*⟩

lemma *zdvd1-eq*[*simp*]: $(x::int)$ *dvd* $1 = (\ |x| = 1)$
⟨*proof*⟩
lemma *zdvd-mult-cancel1*:
assumes $mp{:}m \neq (0::int)$ **shows** $(m * n$ *dvd* $m) = (|n| = 1)$
⟨*proof*⟩

lemma *int-dvd-iff*: $(int\ m$ *dvd* $z) = (m$ *dvd* $nat\ (abs\ z))$
⟨*proof*⟩

lemma *dvd-int-iff*: $(z\ dvd\ int\ m) = (nat\ (abs\ z)\ dvd\ m)$
⟨*proof*⟩

lemma *nat-dvd-iff*: $(nat\ z\ dvd\ m) = (if\ 0 \leq z\ then\ (z\ dvd\ int\ m)\ else\ m = 0)$
⟨*proof*⟩

lemma *zminus-dvd-iff* [*iff*]: $(-z\ dvd\ w) = (z\ dvd\ (w::int))$
⟨*proof*⟩

lemma *dvd-zminus-iff* [*iff*]: $(z\ dvd\ -w) = (z\ dvd\ (w::int))$
⟨*proof*⟩

lemma *zdvd-imp-le*: $[|\ z\ dvd\ n;\ 0 < n\ |] ==> z \leq (n::int)$
⟨*proof*⟩

lemma *zpower-zmod*: $((x::int)\ mod\ m)\ \hat{}\ y\ mod\ m = x\ \hat{}\ y\ mod\ m$
⟨*proof*⟩

lemma *zdiv-int*: $int\ (a\ div\ b) = (int\ a)\ div\ (int\ b)$
⟨*proof*⟩

lemma *zmod-int*: $int\ (a\ mod\ b) = (int\ a)\ mod\ (int\ b)$
⟨*proof*⟩

Suggested by Matthias Daum

lemma *int-power-div-base*:
$\llbracket 0 < m;\ 0 < k \rrbracket \Longrightarrow k\ \hat{}\ m\ div\ k = (k::int)\ \hat{}\ (m - Suc\ 0)$
⟨*proof*⟩

by Brian Huffman

lemma *zminus-zmod*: $-\ ((x::int)\ mod\ m)\ mod\ m = -\ x\ mod\ m$
⟨*proof*⟩

lemma *zdiff-zmod-left*: $(x\ mod\ m - y)\ mod\ m = (x - y)\ mod\ (m::int)$
⟨*proof*⟩

lemma *zdiff-zmod-right*: $(x - y\ mod\ m)\ mod\ m = (x - y)\ mod\ (m::int)$
⟨*proof*⟩

lemmas *zmod-simps* =
IntDiv.zmod-zadd-left-eq [*symmetric*]
IntDiv.zmod-zadd-right-eq [*symmetric*]
IntDiv.zmod-zmult1-eq [*symmetric*]
IntDiv.zmod-zmult1-eq' [*symmetric*]
IntDiv.zpower-zmod
zminus-zmod zdiff-zmod-left zdiff-zmod-right

code generator setup

context *ring-1*
begin

lemma *of-int-num* [*code func*]:
of-int k = (*if k* = *0 then 0 else if k* < *0 then*
− of-int (*− k*) *else let*
(*l*, *m*) = *divAlg* (*k*, *2*);
l′ = *of-int l*
in if m = *0 then l′* + *l′ else l′* + *l′* + *1*)
⟨*proof*⟩

end

code-modulename *SML*
IntDiv Integer

code-modulename *OCaml*
IntDiv Integer

code-modulename *Haskell*
IntDiv Integer

end

26 NatBin: Binary arithmetic for the natural numbers

theory *NatBin*
imports *IntDiv*
begin

Arithmetic for naturals is reduced to that for the non-negative integers.

instantiation *nat* :: *number*
begin

definition
nat-number-of-def [*code inline*]: *number-of v* = *nat* (*number-of v*)

instance ⟨*proof*⟩

end

lemma [*code post*]:
nat (*number-of v*) = *number-of v*
⟨*proof*⟩

abbreviation (*xsymbols*)

square :: *'a*::*power* => *'a* ((-^{2}) [*1000*] *999*) **where**
x^2 == *x^2*

notation (*latex* **output**)
square ((-^{2}) [*1000*] *999*)

notation (*HTML* **output**)
square ((-^{2}) [*1000*] *999*)

26.1 Function *nat*: Coercion from Type *int* to *nat*

declare *nat-0* [*simp*] *nat-1* [*simp*]

lemma *nat-number-of* [*simp*]: *nat* (*number-of w*) = *number-of w*
⟨*proof*⟩

lemma *nat-numeral-0-eq-0* [*simp*]: *Numeral0* = (*0*::*nat*)
⟨*proof*⟩

lemma *nat-numeral-1-eq-1* [*simp*]: *Numeral1* = (*1*::*nat*)
⟨*proof*⟩

lemma *numeral-1-eq-Suc-0*: *Numeral1* = *Suc 0*
⟨*proof*⟩

lemma *numeral-2-eq-2*: *2* = *Suc* (*Suc 0*)
⟨*proof*⟩

Distributive laws for type *nat*. The others are in theory *IntArith*, but these require div and mod to be defined for type "int". They also need some of the lemmas proved above.

lemma *nat-div-distrib*: (*0*::*int*) <= *z* ==> *nat* (*z div z'*) = *nat z div nat z'*
⟨*proof*⟩

lemma *nat-mod-distrib*:
[| (*0*::*int*) <= *z*; *0* <= *z'* |] ==> *nat* (*z mod z'*) = *nat z mod nat z'*
⟨*proof*⟩

Suggested by Matthias Daum

lemma *int-div-less-self*: ⟦*0* < *x*; *1* < *k*⟧ ⟹ *x div k* < (*x*::*int*)
⟨*proof*⟩

26.2 Function *int*: Coercion from Type *nat* to *int*

lemma *int-nat-number-of* [*simp*]:
int (*number-of v*) =
(*if neg* (*number-of v* :: *int*) *then 0*
else (*number-of v* :: *int*))

⟨*proof*⟩

26.2.1 Successor

lemma *Suc-nat-eq-nat-zadd1*: $(0::int) <= z ==> Suc\ (nat\ z) = nat\ (1 + z)$
⟨*proof*⟩

lemma *Suc-nat-number-of-add*:
$Suc\ (number\text{-}of\ v + n) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 1+n\ else\ number\text{-}of\ (Int.succ\ v) + n)$
⟨*proof*⟩

lemma *Suc-nat-number-of* [*simp*]:
$Suc\ (number\text{-}of\ v) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 1\ else\ number\text{-}of\ (Int.succ\ v))$
⟨*proof*⟩

26.2.2 Addition

lemma *add-nat-number-of* [*simp*]:
$(number\text{-}of\ v :: nat) + number\text{-}of\ v' =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ number\text{-}of\ v'$
$else\ if\ neg\ (number\text{-}of\ v' :: int)\ then\ number\text{-}of\ v$
$else\ number\text{-}of\ (v + v'))$
⟨*proof*⟩

26.2.3 Subtraction

lemma *diff-nat-eq-if*:
$nat\ z - nat\ z' =$
$(if\ neg\ z'\ then\ nat\ z$
$else\ let\ d = z-z'\ in$
$if\ neg\ d\ then\ 0\ else\ nat\ d)$
⟨*proof*⟩

lemma *diff-nat-number-of* [*simp*]:
$(number\text{-}of\ v :: nat) - number\text{-}of\ v' =$
$(if\ neg\ (number\text{-}of\ v' :: int)\ then\ number\text{-}of\ v$
$else\ let\ d = number\text{-}of\ (v + uminus\ v')\ in$
$if\ neg\ d\ then\ 0\ else\ nat\ d)$
⟨*proof*⟩

26.2.4 Multiplication

lemma *mult-nat-number-of* [*simp*]:
$(number\text{-}of\ v :: nat) * number\text{-}of\ v' =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 0\ else\ number\text{-}of\ (v * v'))$
⟨*proof*⟩

26.2.5 Quotient

lemma *div-nat-number-of* [*simp*]:
(*number-of v* :: *nat*) *div number-of v′* =
(*if neg* (*number-of v* :: *int*) *then 0*
else nat (*number-of v div number-of v′*))
⟨*proof*⟩

lemma *one-div-nat-number-of* [*simp*]:
(*Suc 0*) *div number-of v′* = (*nat* (*1 div number-of v′*))
⟨*proof*⟩

26.2.6 Remainder

lemma *mod-nat-number-of* [*simp*]:
(*number-of v* :: *nat*) *mod number-of v′* =
(*if neg* (*number-of v* :: *int*) *then 0*
else if neg (*number-of v′* :: *int*) *then number-of v*
else nat (*number-of v mod number-of v′*))
⟨*proof*⟩

lemma *one-mod-nat-number-of* [*simp*]:
(*Suc 0*) *mod number-of v′* =
(*if neg* (*number-of v′* :: *int*) *then Suc 0*
else nat (*1 mod number-of v′*))
⟨*proof*⟩

26.2.7 Divisibility

lemmas *dvd-eq-mod-eq-0-number-of* =
dvd-eq-mod-eq-0 [*of number-of x number-of y*, *standard*]

declare *dvd-eq-mod-eq-0-number-of* [*simp*]

⟨*ML*⟩

26.3 Comparisons

26.3.1 Equals (=)

lemma *eq-nat-nat-iff*:
[| (*0*::*int*) <= *z*; *0* <= *z′* |] ==> (*nat z* = *nat z′*) = (*z*=*z′*)
⟨*proof*⟩

lemma *eq-nat-number-of* [*simp*]:
((*number-of v* :: *nat*) = *number-of v′*) =
(*if neg* (*number-of v* :: *int*) *then* (*iszero* (*number-of v′* :: *int*) | *neg* (*number-of v′* :: *int*))
else if neg (*number-of v′* :: *int*) *then iszero* (*number-of v* :: *int*)
else iszero (*number-of* (*v* + *uminus v′*) :: *int*))

⟨*proof*⟩

26.3.2 Less-than (¡)

lemma *less-nat-number-of* [*simp*]:
((*number-of v* :: *nat*) < *number-of v'*) =
(*if neg* (*number-of v* :: *int*) *then neg* (*number-of* (*uminus v'*) :: *int*)
else neg (*number-of* (*v* + *uminus v'*) :: *int*))
⟨*proof*⟩

lemmas *numerals* = *nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2*

26.4 Powers with Numeric Exponents

We cannot refer to the number *2*::'*a* in *Ring-and-Field.thy*. We cannot prove general results about the numeral −*1*::'*a*, so we have to use − (*1*::'*a*) instead.

lemma *power2-eq-square*: (*a*::'*a*::*recpower*)^2 = *a* * *a*
⟨*proof*⟩

lemma *zero-power2* [*simp*]: (*0*::'*a*::{*semiring-1*,*recpower*})^2 = *0*
⟨*proof*⟩

lemma *one-power2* [*simp*]: (*1*::'*a*::{*semiring-1*,*recpower*})^2 = *1*
⟨*proof*⟩

lemma *power3-eq-cube*: (*x*::'*a*::*recpower*) ^ *3* = *x* * *x* * *x*
⟨*proof*⟩

Squares of literal numerals will be evaluated.

lemmas *power2-eq-square-number-of* =
power2-eq-square [*of number-of w*, *standard*]
declare *power2-eq-square-number-of* [*simp*]

lemma *zero-le-power2*[*simp*]: *0* ≤ (*a*^{2}::'*a*::{*ordered-idom*,*recpower*})
⟨*proof*⟩

lemma *zero-less-power2*[*simp*]:
(*0* < *a*^{2}) = (*a* ≠ (*0*::'*a*::{*ordered-idom*,*recpower*}))
⟨*proof*⟩

lemma *power2-less-0*[*simp*]:
fixes *a* :: '*a*::{*ordered-idom*,*recpower*}
shows ~ (*a*^{2} < *0*)
⟨*proof*⟩

lemma *zero-eq-power2*[*simp*]:
$(a^2 = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\}))$
⟨*proof*⟩

lemma *abs-power2*[*simp*]:
$abs(a^2) = (a^2::'a::\{ordered\text{-}idom,recpower\})$
⟨*proof*⟩

lemma *power2-abs*[*simp*]:
$(abs\ a)^2 = (a^2::'a::\{ordered\text{-}idom,recpower\})$
⟨*proof*⟩

lemma *power2-minus*[*simp*]:
$(-\ a)^2 = (a^2::'a::\{comm\text{-}ring\text{-}1,recpower\})$
⟨*proof*⟩

lemma *power2-le-imp-le*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 \leq y^2;\ 0 \leq y]\!] \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *power2-less-imp-less*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 < y^2;\ 0 \leq y]\!] \Longrightarrow x < y$
⟨*proof*⟩

lemma *power2-eq-imp-eq*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 = y^2;\ 0 \leq x;\ 0 \leq y]\!] \Longrightarrow x = y$
⟨*proof*⟩

lemma *power-minus1-even*[*simp*]: $(-\ 1)\ \hat{}\ (2*n) = (1::'a::\{comm\text{-}ring\text{-}1,recpower\})$
⟨*proof*⟩

lemma *power-even-eq*: $(a::'a::recpower)\ \hat{}\ (2*n) = (a\ \hat{}\ n)\ \hat{}\ 2$
⟨*proof*⟩

lemma *power-odd-eq*: $(a::int)\ \hat{}\ Suc(2*n) = a * (a\ \hat{}\ n)\ \hat{}\ 2$
⟨*proof*⟩

lemma *power-minus-even* [*simp*]:
$(-a)\ \hat{}\ (2*n) = (a::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ (2*n)$
⟨*proof*⟩

lemma *zero-le-even-power'*[*simp*]:
$0 \leq (a::'a::\{ordered\text{-}idom,recpower\})\ \hat{}\ (2*n)$
⟨*proof*⟩

lemma *odd-power-less-zero*:
(*a*::'*a*::{*ordered-idom*,*recpower*}) < *0* ==> *a* ^ *Suc*(*2*∗*n*) < *0*
⟨*proof*⟩

lemma *odd-0-le-power-imp-0-le*:
0 ≤ *a* ^ *Suc*(*2*∗*n*) ==> *0* ≤ (*a*::'*a*::{*ordered-idom*,*recpower*})
⟨*proof*⟩

Simprules for comparisons where common factors can be cancelled.

lemmas *zero-compare-simps* =
add-strict-increasing add-strict-increasing2 add-increasing
zero-le-mult-iff zero-le-divide-iff
zero-less-mult-iff zero-less-divide-iff
mult-le-0-iff divide-le-0-iff
mult-less-0-iff divide-less-0-iff
zero-le-power2 power2-less-0

26.4.1 Nat

lemma *Suc-pred'*: *0* < *n* ==> *n* = *Suc*(*n* − *1*)
⟨*proof*⟩

lemmas *expand-Suc* = *Suc-pred'* [*of number-of v*, *standard*]

26.4.2 Arith

lemma *Suc-eq-add-numeral-1*: *Suc n* = *n* + *1*
⟨*proof*⟩

lemma *Suc-eq-add-numeral-1-left*: *Suc n* = *1* + *n*
⟨*proof*⟩

lemma *add-eq-if*: (*m*::*nat*) + *n* = (*if m=0 then n else Suc* ((*m* − *1*) + *n*))
⟨*proof*⟩

lemma *mult-eq-if*: (*m*::*nat*) ∗ *n* = (*if m=0 then 0 else n* + ((*m* − *1*) ∗ *n*))
⟨*proof*⟩

lemma *power-eq-if*: (*p* ^ *m* :: *nat*) = (*if m=0 then 1 else p* ∗ (*p* ^ (*m* − *1*)))
⟨*proof*⟩

26.5 Comparisons involving (0::nat)

Simplification already does *n* < (*0*::'*a*), *n* ≤ (*0*::'*a*) and (*0*::'*a*) ≤ *n*.

lemma *eq-number-of-0* [*simp*]:
(*number-of v* = (*0*::*nat*)) =

(*if neg* (*number-of v* :: *int*) *then True else iszero* (*number-of v* :: *int*))
⟨*proof*⟩

lemma *eq-0-number-of* [*simp*]:
((*0*::*nat*) = *number-of v*) =
(*if neg* (*number-of v* :: *int*) *then True else iszero* (*number-of v* :: *int*))
⟨*proof*⟩

lemma *less-0-number-of* [*simp*]:
((*0*::*nat*) < *number-of v*) = *neg* (*number-of* (*uminus v*) :: *int*)
⟨*proof*⟩

lemma *neg-imp-number-of-eq-0*: *neg* (*number-of v* :: *int*) ==> *number-of v* = (*0*::*nat*)
⟨*proof*⟩

26.6 Comparisons involving *Suc*

lemma *eq-number-of-Suc* [*simp*]:
(*number-of v* = *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
⟨*proof*⟩

lemma *Suc-eq-number-of* [*simp*]:
(*Suc n* = *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
⟨*proof*⟩

lemma *less-number-of-Suc* [*simp*]:
(*number-of v* < *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv < *n*)
⟨*proof*⟩

lemma *less-Suc-number-of* [*simp*]:
(*Suc n* < *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else n < *nat pv*)
⟨*proof*⟩

lemma *le-number-of-Suc* [*simp*]:
(*number-of v* <= *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv <= *n*)
⟨*proof*⟩

lemma *le-Suc-number-of* [*simp*]:
(*Suc n* <= *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else n <= *nat pv*)
⟨*proof*⟩

lemma *eq-number-of-Pls-Min*: (*Numeral0* ::*int*) ~= *number-of Int.Min*
⟨*proof*⟩

26.7 Max and Min Combined with *Suc*

lemma *max-number-of-Suc* [*simp*]:
max (*Suc n*) (*number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then Suc n else Suc(*max n* (*nat pv*)))
⟨*proof*⟩

lemma *max-Suc-number-of* [*simp*]:
max (*number-of v*) (*Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then Suc n else Suc(*max* (*nat pv*) *n*))
⟨*proof*⟩

lemma *min-number-of-Suc* [*simp*]:
min (*Suc n*) (*number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then 0 else Suc(*min n* (*nat pv*)))
⟨*proof*⟩

lemma *min-Suc-number-of* [*simp*]:
min (*number-of v*) (*Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then 0 else Suc(*min* (*nat pv*) *n*))
⟨*proof*⟩

26.8 Literal arithmetic involving powers

lemma *nat-power-eq*: (*0*::*int*) <= *z* ==> *nat* (*z*^*n*) = *nat z* ^ *n*
⟨*proof*⟩

lemma *power-nat-number-of*:
(*number-of v* :: *nat*) ^ *n* =
(*if neg* (*number-of v* :: *int*) *then 0*^*n else nat* ((*number-of v* :: *int*) ^ *n*))
⟨*proof*⟩

lemmas *power-nat-number-of-number-of* = *power-nat-number-of* [*of* - *number-of w*, *standard*]
declare *power-nat-number-of-number-of* [*simp*]

For arbitrary rings

lemma *power-number-of-even*:
fixes $z :: 'a::\{number\text{-}ring,recpower\}$
shows $z \hat{} number\text{-}of\ (Int.Bit0\ w) = (let\ w = z \hat{} (number\text{-}of\ w)\ in\ w * w)$
⟨*proof*⟩

lemma *power-number-of-odd*:
fixes $z :: 'a::\{number\text{-}ring,recpower\}$
shows $z \hat{} number\text{-}of\ (Int.Bit1\ w) = (if\ (0::int) <= number\text{-}of\ w$
then $(let\ w = z \hat{} (number\text{-}of\ w)\ in\ z * w * w)\ else\ 1)$
⟨*proof*⟩

lemmas *zpower-number-of-even* = *power-number-of-even* [**where** $'a=int$]
lemmas *zpower-number-of-odd* = *power-number-of-odd* [**where** $'a=int$]

lemmas *power-number-of-even-number-of* [*simp*] =
power-number-of-even [*of number-of v*, *standard*]

lemmas *power-number-of-odd-number-of* [*simp*] =
power-number-of-odd [*of number-of v*, *standard*]

⟨*ML*⟩

declare *split-div*[*of - - number-of k*, *standard*, *arith-split*]
declare *split-mod*[*of - - number-of k*, *standard*, *arith-split*]

lemma *nat-number-of-Pls*: $Numeral0 = (0::nat)$
⟨*proof*⟩

lemma *nat-number-of-Min*: $number\text{-}of\ Int.Min = (0::nat)$
⟨*proof*⟩

lemma *nat-number-of-Bit0*:
$number\text{-}of\ (Int.Bit0\ w) = (let\ n::nat = number\text{-}of\ w\ in\ n + n)$
⟨*proof*⟩

lemma *nat-number-of-Bit1*:
$number\text{-}of\ (Int.Bit1\ w) =$
$(if\ neg\ (number\text{-}of\ w :: int)\ then\ 0$
$else\ let\ n = number\text{-}of\ w\ in\ Suc\ (n + n))$
⟨*proof*⟩

lemmas *nat-number* =
nat-number-of-Pls nat-number-of-Min
nat-number-of-Bit0 nat-number-of-Bit1

lemma *Let-Suc* [*simp*]: *Let* (*Suc n*) *f* == *f* (*Suc n*)
⟨*proof*⟩

lemma *power-m1-even*: (−1) ^ (2∗*n*) = (1::'*a*::{*number-ring*,*recpower*})
⟨*proof*⟩

lemma *power-m1-odd*: (−1) ^ *Suc*(2∗*n*) = (−1::'*a*::{*number-ring*,*recpower*})
⟨*proof*⟩

26.9 Literal arithmetic and *of-nat*

lemma *of-nat-double*:
0 ≤ *x* ==> *of-nat* (*nat* (2 ∗ *x*)) = *of-nat* (*nat x*) + *of-nat* (*nat x*)
⟨*proof*⟩

lemma *nat-numeral-m1-eq-0*: −1 = (0::*nat*)
⟨*proof*⟩

lemma *of-nat-number-of-lemma*:
of-nat (*number-of v* :: *nat*) =
(*if* 0 ≤ (*number-of v* :: *int*)
then (*number-of v* :: '*a* :: *number-ring*)
else 0)
⟨*proof*⟩

lemma *of-nat-number-of-eq* [*simp*]:
of-nat (*number-of v* :: *nat*) =
(*if neg* (*number-of v* :: *int*) *then* 0
else (*number-of v* :: '*a* :: *number-ring*))
⟨*proof*⟩

26.10 Lemmas for the Combination and Cancellation Simprocs

lemma *nat-number-of-add-left*:
number-of v + (*number-of v'* + (*k*::*nat*)) =
(*if neg* (*number-of v* :: *int*) *then number-of v'* + *k*
else if neg (*number-of v'* :: *int*) *then number-of v* + *k*
else number-of (*v* + *v'*) + *k*)
⟨*proof*⟩

lemma *nat-number-of-mult-left*:
number-of v ∗ (*number-of v'* ∗ (*k*::*nat*)) =
(*if neg* (*number-of v* :: *int*) *then* 0
else number-of (*v* ∗ *v'*) ∗ *k*)
⟨*proof*⟩

26.10.1 For *combine-numerals*

lemma *left-add-mult-distrib*: *i*∗*u* + (*j*∗*u* + *k*) = (*i*+*j*)∗*u* + (*k*::*nat*)

⟨*proof*⟩

26.10.2 For *cancel-numerals*

lemma *nat-diff-add-eq1*:
$j <= (i::nat) ==> ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)$
⟨*proof*⟩

lemma *nat-diff-add-eq2*:
$i <= (j::nat) ==> ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))$
⟨*proof*⟩

lemma *nat-eq-add-iff1*:
$j <= (i::nat) ==> (i*u + m = j*u + n) = ((i-j)*u + m = n)$
⟨*proof*⟩

lemma *nat-eq-add-iff2*:
$i <= (j::nat) ==> (i*u + m = j*u + n) = (m = (j-i)*u + n)$
⟨*proof*⟩

lemma *nat-less-add-iff1*:
$j <= (i::nat) ==> (i*u + m < j*u + n) = ((i-j)*u + m < n)$
⟨*proof*⟩

lemma *nat-less-add-iff2*:
$i <= (j::nat) ==> (i*u + m < j*u + n) = (m < (j-i)*u + n)$
⟨*proof*⟩

lemma *nat-le-add-iff1*:
$j <= (i::nat) ==> (i*u + m <= j*u + n) = ((i-j)*u + m <= n)$
⟨*proof*⟩

lemma *nat-le-add-iff2*:
$i <= (j::nat) ==> (i*u + m <= j*u + n) = (m <= (j-i)*u + n)$
⟨*proof*⟩

26.10.3 For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: $(0::nat) < k ==> (k*m <= k*n) = (m<=n)$
⟨*proof*⟩

lemma *nat-mult-less-cancel1*: $(0::nat) < k ==> (k*m < k*n) = (m<n)$
⟨*proof*⟩

lemma *nat-mult-eq-cancel1*: $(0::nat) < k ==> (k*m = k*n) = (m=n)$
⟨*proof*⟩

lemma *nat-mult-div-cancel1*: $(0::nat) < k ==> (k*m)\ div\ (k*n) = (m\ div\ n)$
⟨*proof*⟩

lemma *nat-mult-dvd-cancel-disj*[*simp*]:
(*k*∗*m*) *dvd* (*k*∗*n*) = (*k*=*0* | *m dvd* (*n*::*nat*))
⟨*proof*⟩

lemma *nat-mult-dvd-cancel1*: *0* < *k* ⟹ (*k*∗*m*) *dvd* (*k*∗*n*::*nat*) = (*m dvd n*)
⟨*proof*⟩

26.10.4 For *cancel-factor*

lemma *nat-mult-le-cancel-disj*: (*k*∗*m* <= *k*∗*n*) = ((*0*::*nat*) < *k* −−> *m*<=*n*)
⟨*proof*⟩

lemma *nat-mult-less-cancel-disj*: (*k*∗*m* < *k*∗*n*) = ((*0*::*nat*) < *k* & *m*<*n*)
⟨*proof*⟩

lemma *nat-mult-eq-cancel-disj*: (*k*∗*m* = *k*∗*n*) = (*k* = (*0*::*nat*) | *m*=*n*)
⟨*proof*⟩

lemma *nat-mult-div-cancel-disj*[*simp*]:
(*k*∗*m*) *div* (*k*∗*n*) = (*if k* = (*0*::*nat*) *then 0 else m div n*)
⟨*proof*⟩

end

27 Groebner-Basis: Semiring normalization and Groebner Bases

theory *Groebner-Basis*
imports *NatBin*
uses
Tools/Groebner-Basis/misc.ML
Tools/Groebner-Basis/normalizer-data.ML
(*Tools/Groebner-Basis/normalizer.ML*)
(*Tools/Groebner-Basis/groebner.ML*)
begin

27.1 Semiring normalization

⟨*ML*⟩

locale *gb-semiring* =
fixes *add mul pwr r0 r1*
assumes *add-a*:(*add x* (*add y z*) = *add* (*add x y*) *z*)
and *add-c*: *add x y* = *add y x* **and** *add-0*:*add r0 x* = *x*
and *mul-a*:*mul x* (*mul y z*) = *mul* (*mul x y*) *z* **and** *mul-c*:*mul x y* = *mul y x*
and *mul-1*:*mul r1 x* = *x* **and** *mul-0*:*mul r0 x* = *r0*
and *mul-d*:*mul x* (*add y z*) = *add* (*mul x y*) (*mul x z*)

and *pwr-0*:*pwr x 0 = r1* **and** *pwr-Suc*:*pwr x (Suc n) = mul x (pwr x n)*
begin

lemma *mul-pwr*:*mul (pwr x p) (pwr x q) = pwr x (p + q)*
⟨*proof*⟩

lemma *pwr-mul*: *pwr (mul x y) q = mul (pwr x q) (pwr y q)*
⟨*proof*⟩

lemma *pwr-pwr*: *pwr (pwr x p) q = pwr x (p * q)*
⟨*proof*⟩

27.1.1 Declaring the abstract theory

lemma *semiring-ops*:
includes *meta-term-syntax*
shows *TERM (add x y)* **and** *TERM (mul x y)* **and** *TERM (pwr x n)*
and *TERM r0* **and** *TERM r1*
⟨*proof*⟩

lemma *semiring-rules*:
add (mul a m) (mul b m) = mul (add a b) m
add (mul a m) m = mul (add a r1) m
add m (mul a m) = mul (add a r1) m
add m m = mul (add r1 r1) m
add r0 a = a
add a r0 = a
mul a b = mul b a
mul (add a b) c = add (mul a c) (mul b c)
mul r0 a = r0
mul a r0 = r0
mul r1 a = a
mul a r1 = a
mul (mul lx ly) (mul rx ry) = mul (mul lx rx) (mul ly ry)
mul (mul lx ly) (mul rx ry) = mul lx (mul ly (mul rx ry))
mul (mul lx ly) (mul rx ry) = mul rx (mul (mul lx ly) ry)
mul (mul lx ly) rx = mul (mul lx rx) ly
mul (mul lx ly) rx = mul lx (mul ly rx)
mul lx (mul rx ry) = mul (mul lx rx) ry
mul lx (mul rx ry) = mul rx (mul lx ry)
add (add a b) (add c d) = add (add a c) (add b d)
add (add a b) c = add a (add b c)
add a (add c d) = add c (add a d)
add (add a b) c = add (add a c) b
add a c = add c a
add a (add c d) = add (add a c) d
mul (pwr x p) (pwr x q) = pwr x (p + q)
mul x (pwr x q) = pwr x (Suc q)
mul (pwr x q) x = pwr x (Suc q)

mul x x = pwr x 2
pwr (mul x y) q = mul (pwr x q) (pwr y q)
*pwr (pwr x p) q = pwr x (p * q)*
pwr x 0 = r1
pwr x 1 = x
mul x (add y z) = add (mul x y) (mul x z)
pwr x (Suc q) = mul x (pwr x q)
*pwr x (2*n) = mul (pwr x n) (pwr x n)*
*pwr x (Suc (2*n)) = mul x (mul (pwr x n) (pwr x n))*
⟨*proof*⟩

lemmas *gb-semiring-axioms' =*
gb-semiring-axioms [normalizer
semiring ops: semiring-ops
semiring rules: semiring-rules]

end

interpretation *class-semiring: gb-semiring*
*[op + op * op ^ 0::'a::{comm-semiring-1, recpower} 1]*
⟨*proof*⟩

lemmas *nat-arith =*
add-nat-number-of diff-nat-number-of mult-nat-number-of eq-nat-number-of less-nat-number-of

lemma *not-iszero-Numeral1: ¬ iszero (Numeral1::'a::number-ring)*
⟨*proof*⟩
lemmas *comp-arith = Let-def arith-simps nat-arith rel-simps if-False*
if-True add-0 add-Suc add-number-of-left mult-number-of-left
numeral-1-eq-1[symmetric] Suc-eq-add-numeral-1
numeral-0-eq-0[symmetric] numerals[symmetric] not-iszero-1
iszero-number-of-Bit1 iszero-number-of-Bit0 nonzero-number-of-Min
iszero-number-of-Pls iszero-0 not-iszero-Numeral1

lemmas *semiring-norm = comp-arith*

⟨*ML*⟩

locale *gb-ring = gb-semiring +*
fixes *sub :: 'a ⇒ 'a ⇒ 'a*
and *neg :: 'a ⇒ 'a*
assumes *neg-mul: neg x = mul (neg r1) x*
and *sub-add: sub x y = add x (neg y)*
begin

lemma *ring-ops:*
includes *meta-term-syntax*

shows *TERM* (*sub x y*) **and** *TERM* (*neg x*) ⟨*proof*⟩

lemmas *ring-rules* = *neg-mul sub-add*

lemmas *gb-ring-axioms'* =
gb-ring-axioms [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*]

end

interpretation *class-ring*: *gb-ring* [*op* + *op* ∗ *op* ^
0::'*a*::{*comm-semiring-1*,*recpower*,*number-ring*} *1 op* − *uminus*]
⟨*proof*⟩

⟨*ML*⟩

locale *gb-field* = *gb-ring* +
fixes *divide* :: 'a ⇒ 'a ⇒ 'a
and *inverse*:: 'a ⇒ 'a
assumes *divide*: *divide x y* = *mul x* (*inverse y*)
and *inverse*: *inverse x* = *divide r1 x*
begin

lemmas *gb-field-axioms'* =
gb-field-axioms [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*]

end

27.2 Groebner Bases

locale *semiringb* = *gb-semiring* +
assumes *add-cancel*: *add* (*x*::'*a*) *y* = *add x z* ⟷ *y* = *z*
and *add-mul-solve*: *add* (*mul w y*) (*mul x z*) =
add (*mul w z*) (*mul x y*) ⟷ *w* = *x* ∨ *y* = *z*
begin

lemma *noteq-reduce*: *a* ≠ *b* ∧ *c* ≠ *d* ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
⟨*proof*⟩

lemma *add-scale-eq-noteq*: $[\![r \neq r0 \ ; \ (a = b) \wedge \sim(c = d)]\!]$
$\Longrightarrow$ *add a* (*mul r c*) $\neq$ *add b* (*mul r d*)
⟨*proof*⟩

lemma *add-r0-iff*: $x =$ *add x a* $\longleftrightarrow$ $a = r0$
⟨*proof*⟩

declare *gb-semiring-axioms'* [*normalizer del*]

lemmas *semiringb-axioms'* = *semiringb-axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
idom rules: *noteq-reduce add-scale-eq-noteq*]

end

locale *ringb* = *semiringb* + *gb-ring* +
assumes *subr0-iff*: *sub x y* = *r0* $\longleftrightarrow$ $x = y$
begin

declare *gb-ring-axioms'* [*normalizer del*]

lemmas *ringb-axioms'* = *ringb-axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*
idom rules: *noteq-reduce add-scale-eq-noteq*
ideal rules: *subr0-iff add-r0-iff*]

end

lemma *no-zero-divirors-neq0*:
assumes *az*: (*a*::'*a*::*no-zero-divisors*) $\neq$ *0*
and *ab*: *a*∗*b* = *0* **shows** *b* = *0*
⟨*proof*⟩

interpretation *class-ringb*: *ringb*
[*op* + *op* ∗ *op* ^ *0*::'*a*::{*idom*,*recpower*,*number-ring*} *1 op* − *uminus*]
⟨*proof*⟩

⟨*ML*⟩

interpretation *natgb*: *semiringb*
[*op* + *op* ∗ *op* ^ *0*::*nat 1*]
⟨*proof*⟩

⟨*ML*⟩

locale *fieldgb* = *ringb* + *gb-field*
begin

declare *gb-field-axioms'* [*normalizer del*]

lemmas *fieldgb-axioms'* = *fieldgb-axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*
idom rules: *noteq-reduce add-scale-eq-noteq*
ideal rules: *subr0-iff add-r0-iff*]

end

lemmas *bool-simps* = *simp-thms*(*1−34*)
lemma *dnf*:
(*P* & (*Q* | *R*)) = ((*P*&*Q*) | (*P*&*R*)) ((*Q* | *R*) & *P*) = ((*Q*&*P*) | (*R*&*P*))
(*P* ∧ *Q*) = (*Q* ∧ *P*) (*P* ∨ *Q*) = (*Q* ∨ *P*)
⟨*proof*⟩

lemmas *weak-dnf-simps* = *dnf bool-simps*

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

lemma *PFalse*:
P ≡ *False* ⟹ ¬ *P*
¬ *P* ⟹ (*P* ≡ *False*)
⟨*proof*⟩

⟨*ML*⟩

end

28 Arith-Tools: Setup of arithmetic tools

theory *Arith-Tools*
imports *Groebner-Basis*
uses
~~/src/Provers/Arith/cancel-numeral-factor.ML
~~/src/Provers/Arith/extract-common-term.ML
int-factor-simprocs.ML

nat-simprocs.ML
begin

28.1 Simprocs for the Naturals

⟨*ML*⟩

28.1.1 For simplifying *Suc m − K* and *K − Suc m*

Where K above is a literal

lemma *Suc-diff-eq-diff-pred*: *Numeral0 < n ==> Suc m − n = m − (n − Numeral1)*
⟨*proof*⟩

Now just instantiating *n* to *number-of v* does the right simplification, but with some redundant inequality tests.

lemma *neg-number-of-pred-iff-0*:
neg (number-of (Int.pred v)::int) = (number-of v = (0::nat))
⟨*proof*⟩

No longer required as a simprule because of the *inverse-fold* simproc

lemma *Suc-diff-number-of*:
neg (number-of (uminus v)::int) ==>
Suc m − (number-of v) = m − (number-of (Int.pred v))
⟨*proof*⟩

lemma *diff-Suc-eq-diff-pred*: *m − Suc n = (m − 1) − n*
⟨*proof*⟩

28.1.2 For *nat-case* and *nat-rec*

lemma *nat-case-number-of* [*simp*]:
nat-case a f (number-of v) =
(let pv = number-of (Int.pred v) in
if neg pv then a else f (nat pv))
⟨*proof*⟩

lemma *nat-case-add-eq-if* [*simp*]:
nat-case a f ((number-of v) + n) =
(let pv = number-of (Int.pred v) in
if neg pv then nat-case a f n else f (nat pv + n))
⟨*proof*⟩

lemma *nat-rec-number-of* [*simp*]:
nat-rec a f (number-of v) =
(let pv = number-of (Int.pred v) in
if neg pv then a else f (nat pv) (nat-rec a f (nat pv)))
⟨*proof*⟩

lemma *nat-rec-add-eq-if* [*simp*]:
    nat-rec a f (*number-of v* + *n*) =
      (*let pv* = *number-of* (*Int.pred v*) *in*
       if neg pv then nat-rec a f n
             else f (*nat pv* + *n*) (*nat-rec a f* (*nat pv* + *n*)))
⟨*proof*⟩

28.1.3 Various Other Lemmas

Evens and Odds, for Mutilated Chess Board

Lemmas for specialist use, NOT as default simprules

lemma *nat-mult-2*: *2* ∗ *z* = (*z*+*z*::*nat*)
⟨*proof*⟩

lemma *nat-mult-2-right*: *z* ∗ *2* = (*z*+*z*::*nat*)
⟨*proof*⟩

Case analysis on $n < (2::'a)$

lemma *less-2-cases*: (*n*::*nat*) < *2* ==> *n* = *0* | *n* = *Suc 0*
⟨*proof*⟩

lemma *div2-Suc-Suc* [*simp*]: *Suc*(*Suc m*) *div 2* = *Suc* (*m div 2*)
⟨*proof*⟩

lemma *add-self-div-2* [*simp*]: (*m* + *m*) *div 2* = (*m*::*nat*)
⟨*proof*⟩

lemma *mod2-Suc-Suc* [*simp*]: *Suc*(*Suc*(*m*)) *mod 2* = *m mod 2*
⟨*proof*⟩

lemma *mod2-gr-0* [*simp*]: !!*m*::*nat*. (*0* < *m mod 2*) = (*m mod 2* = *1*)
⟨*proof*⟩

Removal of Small Numerals: 0, 1 and (in additive positions) 2

lemma *add-2-eq-Suc* [*simp*]: *2* + *n* = *Suc* (*Suc n*)
⟨*proof*⟩

lemma *add-2-eq-Suc'* [*simp*]: *n* + *2* = *Suc* (*Suc n*)
⟨*proof*⟩

Can be used to eliminate long strings of Sucs, but not by default

lemma *Suc3-eq-add-3*: *Suc* (*Suc* (*Suc n*)) = *3* + *n*
⟨*proof*⟩

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

lemma *div-Suc-eq-div-add3* [*simp*]: *m div* (*Suc* (*Suc* (*Suc n*))) = *m div* (*3*+*n*)
⟨*proof*⟩

lemma *mod-Suc-eq-mod-add3* [*simp*]: *m mod* (*Suc* (*Suc* (*Suc n*))) = *m mod* (*3*+*n*)
⟨*proof*⟩

lemma *Suc-div-eq-add3-div*: (*Suc* (*Suc* (*Suc m*))) *div n* = (*3*+*m*) *div n*
⟨*proof*⟩

lemma *Suc-mod-eq-add3-mod*: (*Suc* (*Suc* (*Suc m*))) *mod n* = (*3*+*m*) *mod n*
⟨*proof*⟩

lemmas *Suc-div-eq-add3-div-number-of* =
Suc-div-eq-add3-div [*of - number-of v*, *standard*]
declare *Suc-div-eq-add3-div-number-of* [*simp*]

lemmas *Suc-mod-eq-add3-mod-number-of* =
Suc-mod-eq-add3-mod [*of - number-of v*, *standard*]
declare *Suc-mod-eq-add3-mod-number-of* [*simp*]

28.1.4 Special Simplification for Constants

These belong here, late in the development of HOL, to prevent their interfering with proofs of abstract properties of instances of the function *number-of*

These distributive laws move literals inside sums and differences.

lemmas *left-distrib-number-of* = *left-distrib* [*of - - number-of v*, *standard*]
declare *left-distrib-number-of* [*simp*]

lemmas *right-distrib-number-of* = *right-distrib* [*of number-of v*, *standard*]
declare *right-distrib-number-of* [*simp*]

lemmas *left-diff-distrib-number-of* =
left-diff-distrib [*of - - number-of v*, *standard*]
declare *left-diff-distrib-number-of* [*simp*]

lemmas *right-diff-distrib-number-of* =
right-diff-distrib [*of number-of v*, *standard*]
declare *right-diff-distrib-number-of* [*simp*]

These are actually for fields, like real: but where else to put them?

lemmas *zero-less-divide-iff-number-of* =
zero-less-divide-iff [*of number-of w*, *standard*]
declare *zero-less-divide-iff-number-of* [*simp*,*noatp*]

lemmas *divide-less-0-iff-number-of* =
divide-less-0-iff [*of number-of w*, *standard*]
declare *divide-less-0-iff-number-of* [*simp*,*noatp*]

lemmas *zero-le-divide-iff-number-of* =
zero-le-divide-iff [*of number-of w*, *standard*]
declare *zero-le-divide-iff-number-of* [*simp*,*noatp*]

lemmas *divide-le-0-iff-number-of* =
divide-le-0-iff [*of number-of w*, *standard*]
declare *divide-le-0-iff-number-of* [*simp*,*noatp*]

Replaces *inverse #nn* by *1/#nn*. It looks strange, but then other simprocs simplify the quotient.

lemmas *inverse-eq-divide-number-of* =
inverse-eq-divide [*of number-of w*, *standard*]
declare *inverse-eq-divide-number-of* [*simp*]

These laws simplify inequalities, moving unary minus from a term into the literal.

lemmas *less-minus-iff-number-of* =
less-minus-iff [*of number-of v*, *standard*]
declare *less-minus-iff-number-of* [*simp*,*noatp*]

lemmas *le-minus-iff-number-of* =
le-minus-iff [*of number-of v*, *standard*]
declare *le-minus-iff-number-of* [*simp*,*noatp*]

lemmas *equation-minus-iff-number-of* =
equation-minus-iff [*of number-of v*, *standard*]
declare *equation-minus-iff-number-of* [*simp*,*noatp*]

lemmas *minus-less-iff-number-of* =
minus-less-iff [*of - number-of v*, *standard*]
declare *minus-less-iff-number-of* [*simp*,*noatp*]

lemmas *minus-le-iff-number-of* =
minus-le-iff [*of - number-of v*, *standard*]
declare *minus-le-iff-number-of* [*simp*,*noatp*]

lemmas *minus-equation-iff-number-of* =
minus-equation-iff [*of - number-of v*, *standard*]
declare *minus-equation-iff-number-of* [*simp*,*noatp*]

To Simplify Inequalities Where One Side is the Constant 1

lemma *less-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::{*ordered-idom*,*number-ring*}
shows (*1* < − *b*) = (*b* < −*1*)
⟨*proof*⟩

lemma *le-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::{*ordered-idom*,*number-ring*}
shows (*1* ≤ − *b*) = (*b* ≤ −*1*)
⟨*proof*⟩

lemma *equation-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::*number-ring*
shows (*1* = − *b*) = (*b* = −*1*)
⟨*proof*⟩

lemma *minus-less-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::{*ordered-idom*,*number-ring*}
shows (− *a* < *1*) = (−*1* < *a*)
⟨*proof*⟩

lemma *minus-le-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::{*ordered-idom*,*number-ring*}
shows (− *a* ≤ *1*) = (−*1* ≤ *a*)
⟨*proof*⟩

lemma *minus-equation-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::*number-ring*
shows (− *a* = *1*) = (*a* = −*1*)
⟨*proof*⟩

Cancellation of constant factors in comparisons (< and ≤)

lemmas *mult-less-cancel-left-number-of* =
mult-less-cancel-left [*of number-of v*, *standard*]
declare *mult-less-cancel-left-number-of* [*simp*,*noatp*]

lemmas *mult-less-cancel-right-number-of* =
mult-less-cancel-right [*of - number-of v*, *standard*]
declare *mult-less-cancel-right-number-of* [*simp*,*noatp*]

lemmas *mult-le-cancel-left-number-of* =
mult-le-cancel-left [*of number-of v*, *standard*]
declare *mult-le-cancel-left-number-of* [*simp*,*noatp*]

lemmas *mult-le-cancel-right-number-of* =
mult-le-cancel-right [*of - number-of v*, *standard*]
declare *mult-le-cancel-right-number-of* [*simp*,*noatp*]

Multiplying out constant divisors in comparisons (<, ≤ and =)

lemmas *le-divide-eq-number-of1* [*simp*] = *le-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-le-eq-number-of1* [*simp*] = *divide-le-eq* [*of - number-of w*, *standard*]
lemmas *less-divide-eq-number-of1* [*simp*] = *less-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-less-eq-number-of1* [*simp*] = *divide-less-eq* [*of - number-of w*, *stan-*

dard]
lemmas *eq-divide-eq-number-of1* [*simp*] = *eq-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-eq-eq-number-of1* [*simp*] = *divide-eq-eq* [*of - number-of w*, *standard*]

28.1.5 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas *le-divide-eq-number-of* = *le-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-le-eq-number-of* = *divide-le-eq* [*of - - number-of w*, *standard*]
lemmas *less-divide-eq-number-of* = *less-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-less-eq-number-of* = *divide-less-eq* [*of - - number-of w*, *standard*]
lemmas *eq-divide-eq-number-of* = *eq-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-eq-eq-number-of* = *divide-eq-eq* [*of - - number-of w*, *standard*]

Not good as automatic simprules because they cause case splits.

lemmas *divide-const-simps* =
le-divide-eq-number-of divide-le-eq-number-of less-divide-eq-number-of
divide-less-eq-number-of eq-divide-eq-number-of divide-eq-eq-number-of
le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1

Division By -1

lemma *divide-minus1* [*simp*]:
$x/-1 = -(x::'a::\{field, division\text{-}by\text{-}zero, number\text{-}ring\})$
⟨*proof*⟩

lemma *minus1-divide* [*simp*]:
$-1 \;/\; (x::'a::\{field, division\text{-}by\text{-}zero, number\text{-}ring\}) = -\;(1/x)$
⟨*proof*⟩

lemma *half-gt-zero-iff*:
$(0 < r/2) = (0 < (r::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero, number\text{-}ring\}))$
⟨*proof*⟩

lemmas *half-gt-zero* = *half-gt-zero-iff* [*THEN iffD2*, *standard*]
declare *half-gt-zero* [*simp*]

lemma *nat-dvd-not-less*:
[| $0 < m$; $m < n$ |] ==> $\neg$ n *dvd* $(m::nat)$
⟨*proof*⟩

⟨*ML*⟩

28.2 Groebner Bases for fields

interpretation *class-fieldgb*:

fieldgb[*op + op * op ^ 0*::'*a*::{*field*,*recpower*,*number-ring*} *1 op − uminus op / inverse*] ⟨*proof*⟩

lemma *divide-Numeral1*: (*x*::'*a*::{*field*,*number-ring*}) / *Numeral1* = *x* ⟨*proof*⟩
lemma *divide-Numeral0*: (*x*::'*a*::{*field*,*number-ring*, *division-by-zero*}) / *Numeral0* = *0*
⟨*proof*⟩
lemma *mult-frac-frac*: ((*x*::'*a*::{*field*,*division-by-zero*}) / *y*) * (*z* / *w*) = (*x***z*) / (*y***w*)
⟨*proof*⟩
lemma *mult-frac-num*: ((*x*::'*a*::{*field*, *division-by-zero*}) / *y*) * *z* = (*x***z*) / *y*
⟨*proof*⟩
lemma *mult-num-frac*: ((*x*::'*a*::{*field*, *division-by-zero*}) / *y*) * *z* = (*x***z*) / *y*
⟨*proof*⟩

lemma *Numeral1-eq1-nat*: (*1*::*nat*) = *Numeral1* ⟨*proof*⟩

lemma *add-frac-num*: *y*≠ *0* ⟹ (*x*::'*a*::{*field*, *division-by-zero*}) / *y* + *z* = (*x* + *z***y*) / *y*
⟨*proof*⟩
lemma *add-num-frac*: *y*≠ *0* ⟹ *z* + (*x*::'*a*::{*field*, *division-by-zero*}) / *y* = (*x* + *z***y*) / *y*
⟨*proof*⟩

⟨*ML*⟩
end

29 SetInterval: Set intervals

theory *SetInterval*
imports *Int*
begin

context *ord*
begin
definition
lessThan :: '*a* => '*a set* ((*1*{..<-})) **where**
{..<*u*} == {*x*. *x* < *u*}

definition
atMost :: '*a* => '*a set* ((*1*{..-})) **where**
{..*u*} == {*x*. *x* ≤ *u*}

definition
greaterThan :: '*a* => '*a set* ((*1*{-<..})) **where**
{*l*<..} == {*x*. *l*<*x*}

definition
atLeast :: *'a => 'a set* ((*1*{-..})) **where**
{*l*..} == {*x*. *l*≤*x*}

definition
greaterThanLessThan :: *'a => 'a => 'a set* ((*1*{-<..<-})) **where**
{*l*<..<*u*} == {*l*<..} *Int* {..<*u*}

definition
atLeastLessThan :: *'a => 'a => 'a set* ((*1*{-..<-})) **where**
{*l*..<*u*} == {*l*..} *Int* {..<*u*}

definition
greaterThanAtMost :: *'a => 'a => 'a set* ((*1*{-<..-})) **where**
{*l*<..*u*} == {*l*<..} *Int* {..*u*}

definition
atLeastAtMost :: *'a => 'a => 'a set* ((*1*{-..-})) **where**
{*l*..*u*} == {*l*..} *Int* {..*u*}

end

A note of warning when using {..<*n*} on type *nat*: it is equivalent to {*0*..<*n*} but some lemmas involving {*m*..<*n*} may not exist in {..<*n*}-form as well.

syntax
@*UNION-le* :: *nat => nat => 'b set => 'b set* ((*3UN* -<=-./ -) *10*)
@*UNION-less* :: *nat => nat => 'b set => 'b set* ((*3UN* -<-./ -) *10*)
@*INTER-le* :: *nat => nat => 'b set => 'b set* ((*3INT* -<=-./ -) *10*)
@*INTER-less* :: *nat => nat => 'b set => 'b set* ((*3INT* -<-./ -) *10*)

syntax (*input*)
@*UNION-le* :: *nat => nat => 'b set => 'b set* ((*3*⋃ -≤-./ -) *10*)
@*UNION-less* :: *nat => nat => 'b set => 'b set* ((*3*⋃ -<-./ -) *10*)
@*INTER-le* :: *nat => nat => 'b set => 'b set* ((*3*⋂ -≤-./ -) *10*)
@*INTER-less* :: *nat => nat => 'b set => 'b set* ((*3*⋂ -<-./ -) *10*)

syntax (*xsymbols*)
@*UNION-le* :: *nat ⇒ nat => 'b set => 'b set* ((*3*⋃(*00*_ ≤ _)/ -) *10*)
@*UNION-less* :: *nat ⇒ nat => 'b set => 'b set* ((*3*⋃(*00*_ < _)/ -) *10*)
@*INTER-le* :: *nat ⇒ nat => 'b set => 'b set* ((*3*⋂(*00*_ ≤ _)/ -) *10*)
@*INTER-less* :: *nat ⇒ nat => 'b set => 'b set* ((*3*⋂(*00*_ < _)/ -) *10*)

translations
UN i<=n. A == *UN i:{..n}. A*
UN i<n. A == *UN i:{..<n}. A*
INT i<=n. A == *INT i:{..n}. A*
INT i<n. A == *INT i:{..<n}. A*

29.1 Various equivalences

lemma (**in** *ord*) *lessThan-iff* [*iff*]: (*i*: *lessThan k*) = (*i*<*k*)
⟨*proof*⟩

lemma *Compl-lessThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*lessThan k* = *atLeast k*
⟨*proof*⟩

lemma *single-Diff-lessThan* [*simp*]: !!*k*:: *'a*::*order*. {*k*} − *lessThan k* = {*k*}
⟨*proof*⟩

lemma (**in** *ord*) *greaterThan-iff* [*iff*]: (*i*: *greaterThan k*) = (*k*<*i*)
⟨*proof*⟩

lemma *Compl-greaterThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*greaterThan k* = *atMost k*
⟨*proof*⟩

lemma *Compl-atMost* [*simp*]: !!*k*:: *'a*::*linorder*. −*atMost k* = *greaterThan k*
⟨*proof*⟩

lemma (**in** *ord*) *atLeast-iff* [*iff*]: (*i*: *atLeast k*) = (*k*<=*i*)
⟨*proof*⟩

lemma *Compl-atLeast* [*simp*]:
!!*k*:: *'a*::*linorder*. −*atLeast k* = *lessThan k*
⟨*proof*⟩

lemma (**in** *ord*) *atMost-iff* [*iff*]: (*i*: *atMost k*) = (*i*<=*k*)
⟨*proof*⟩

lemma *atMost-Int-atLeast*: !!*n*:: *'a*::*order*. *atMost n Int atLeast n* = {*n*}
⟨*proof*⟩

29.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-subset-iff* [*iff*]:
(*atLeast x* ⊆ *atLeast y*) = (*y* ≤ (*x*::*'a*::*order*))
⟨*proof*⟩

lemma *atLeast-eq-iff* [*iff*]:
(*atLeast x* = *atLeast y*) = (*x* = (*y*::*'a*::*linorder*))
⟨*proof*⟩

lemma *greaterThan-subset-iff* [*iff*]:
(*greaterThan x* ⊆ *greaterThan y*) = (*y* ≤ (*x*::*'a*::*linorder*))
⟨*proof*⟩

lemma *greaterThan-eq-iff* [*iff*]:

$(greaterThan\ x = greaterThan\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *atMost-subset-iff* [*iff*]: $(atMost\ x \subseteq atMost\ y) = (x \leq (y::'a::order))$
⟨*proof*⟩

lemma *atMost-eq-iff* [*iff*]: $(atMost\ x = atMost\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *lessThan-subset-iff* [*iff*]:
$(lessThan\ x \subseteq lessThan\ y) = (x \leq (y::'a::linorder))$
⟨*proof*⟩

lemma *lessThan-eq-iff* [*iff*]:
$(lessThan\ x = lessThan\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

29.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [*simp*,*noatp*]:
$(i : \{l<..<u\}) = (l < i\ \&\ i < u)$
⟨*proof*⟩

lemma *atLeastLessThan-iff* [*simp*,*noatp*]:
$(i : \{l..<u\}) = (l <= i\ \&\ i < u)$
⟨*proof*⟩

lemma *greaterThanAtMost-iff* [*simp*,*noatp*]:
$(i : \{l<..u\}) = (l < i\ \&\ i <= u)$
⟨*proof*⟩

lemma *atLeastAtMost-iff* [*simp*,*noatp*]:
$(i : \{l..u\}) = (l <= i\ \&\ i <= u)$
⟨*proof*⟩

The above four lemmas could be declared as iffs. If we do so, a call to blast in Hyperreal/Star.ML, lemma *STAR-Int* seems to take forever (more than one hour).

end

29.3.1 Emptyness and singletons

context *order*
begin

lemma *atLeastAtMost-empty* [*simp*]: $n < m ==> \{m..n\} = \{\}$

⟨*proof*⟩

lemma *atLeastLessThan-empty*[*simp*]: $n \leq m \Longrightarrow \{m..<n\} = \{\}$
⟨*proof*⟩

lemma *greaterThanAtMost-empty*[*simp*]:$l \leq k \Longrightarrow \{k<..l\} = \{\}$
⟨*proof*⟩

lemma *greaterThanLessThan-empty*[*simp*]:$l \leq k \Longrightarrow \{k<..l\} = \{\}$
⟨*proof*⟩

lemma *atLeastAtMost-singleton* [*simp*]: $\{a..a\} = \{a\}$
⟨*proof*⟩

end

29.4 Intervals of natural numbers

29.4.1 The Constant *lessThan*

lemma *lessThan-0* [*simp*]: *lessThan* (*0*::*nat*) = {}
⟨*proof*⟩

lemma *lessThan-Suc*: *lessThan* (*Suc k*) = *insert k* (*lessThan k*)
⟨*proof*⟩

lemma *lessThan-Suc-atMost*: *lessThan* (*Suc k*) = *atMost k*
⟨*proof*⟩

lemma *UN-lessThan-UNIV*: (*UN m*::*nat*. *lessThan m*) = *UNIV*
⟨*proof*⟩

29.4.2 The Constant *greaterThan*

lemma *greaterThan-0* [*simp*]: *greaterThan 0* = *range Suc*
⟨*proof*⟩

lemma *greaterThan-Suc*: *greaterThan* (*Suc k*) = *greaterThan k* − {*Suc k*}
⟨*proof*⟩

lemma *INT-greaterThan-UNIV*: (*INT m*::*nat*. *greaterThan m*) = {}
⟨*proof*⟩

29.4.3 The Constant *atLeast*

lemma *atLeast-0* [*simp*]: *atLeast* (*0*::*nat*) = *UNIV*
⟨*proof*⟩

lemma *atLeast-Suc*: *atLeast* (*Suc k*) = *atLeast k* − {*k*}
⟨*proof*⟩

lemma *atLeast-Suc-greaterThan*: *atLeast* (*Suc k*) = *greaterThan k*
⟨*proof*⟩

lemma *UN-atLeast-UNIV*: (*UN m*::*nat*. *atLeast m*) = *UNIV*
⟨*proof*⟩

29.4.4 The Constant *atMost*

lemma *atMost-0* [*simp*]: *atMost* (*0*::*nat*) = {*0*}
⟨*proof*⟩

lemma *atMost-Suc*: *atMost* (*Suc k*) = *insert* (*Suc k*) (*atMost k*)
⟨*proof*⟩

lemma *UN-atMost-UNIV*: (*UN m*::*nat*. *atMost m*) = *UNIV*
⟨*proof*⟩

29.4.5 The Constant *atLeastLessThan*

The orientation of the following rule is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

lemma *atLeast0LessThan*: {*0*::*nat*..<*n*} = {..<*n*}
⟨*proof*⟩

declare *atLeast0LessThan*[*symmetric*, *code unfold*]

lemma *atLeastLessThan0*: {*m*..<*0*::*nat*} = {}
⟨*proof*⟩

29.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
{*m*..<*Suc n*} = (*if m* ≤ *n then insert n* {*m*..<*n*} *else* {})
⟨*proof*⟩

lemma *atLeastLessThan-singleton* [*simp*]: {*m*..<*Suc m*} = {*m*}
⟨*proof*⟩

lemma *atLeastLessThanSuc-atLeastAtMost*: {*l*..<*Suc u*} = {*l*..*u*}
⟨*proof*⟩

lemma *atLeastSucAtMost-greaterThanAtMost*: {*Suc l*..*u*} = {*l*<..*u*}
⟨*proof*⟩

lemma *atLeastSucLessThan-greaterThanLessThan*: $\{Suc\ l..<u\} = \{l<..<u\}$
⟨*proof*⟩

lemma *atLeastAtMostSuc-conv*: $m \leq Suc\ n \Longrightarrow \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
⟨*proof*⟩

29.4.7 Image

lemma *image-add-atLeastAtMost*:
$(\%n{::}nat.\ n+k)\ `\ \{i..j\} = \{i+k..j+k\}$ (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *image-add-atLeastLessThan*:
$(\%n{::}nat.\ n+k)\ `\ \{i..<j\} = \{i+k..<j+k\}$ (**is** *?A* = *?B*)
⟨*proof*⟩

corollary *image-Suc-atLeastAtMost*[*simp*]:
$Suc\ `\ \{i..j\} = \{Suc\ i..Suc\ j\}$
⟨*proof*⟩

corollary *image-Suc-atLeastLessThan*[*simp*]:
$Suc\ `\ \{i..<j\} = \{Suc\ i..<Suc\ j\}$
⟨*proof*⟩

lemma *image-add-int-atLeastLessThan*:
$(\%x.\ x + (l{::}int))\ `\ \{0..<u-l\} = \{l..<u\}$
⟨*proof*⟩

29.4.8 Finiteness

lemma *finite-lessThan* [*iff*]: **fixes** $k :: nat$ **shows** *finite* $\{..<k\}$
⟨*proof*⟩

lemma *finite-atMost* [*iff*]: **fixes** $k :: nat$ **shows** *finite* $\{..k\}$
⟨*proof*⟩

lemma *finite-greaterThanLessThan* [*iff*]:
fixes $l :: nat$ **shows** *finite* $\{l<..<u\}$
⟨*proof*⟩

lemma *finite-atLeastLessThan* [*iff*]:
fixes $l :: nat$ **shows** *finite* $\{l..<u\}$
⟨*proof*⟩

lemma *finite-greaterThanAtMost* [*iff*]:
fixes $l :: nat$ **shows** *finite* $\{l<..u\}$
⟨*proof*⟩

lemma *finite-atLeastAtMost* [*iff*]:
fixes *l* :: *nat* **shows** *finite* {*l*..*u*}
⟨*proof*⟩

lemma *bounded-nat-set-is-finite*:
(*ALL i:N*. *i* < (*n*::*nat*)) ==> *finite N*
— A bounded set of natural numbers is finite.
⟨*proof*⟩

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
A <= {*k*..<*k*+*card A*} ⟹ *A* = {*k*..<*k*+*card A*} (**is** *PROP ?P*)
⟨*proof*⟩

29.4.9 Cardinality

lemma *card-lessThan* [*simp*]: *card* {..<*u*} = *u*
⟨*proof*⟩

lemma *card-atMost* [*simp*]: *card* {..*u*} = *Suc u*
⟨*proof*⟩

lemma *card-atLeastLessThan* [*simp*]: *card* {*l*..<*u*} = *u* − *l*
⟨*proof*⟩

lemma *card-atLeastAtMost* [*simp*]: *card* {*l*..*u*} = *Suc u* − *l*
⟨*proof*⟩

lemma *card-greaterThanAtMost* [*simp*]: *card* {*l*<..*u*} = *u* − *l*
⟨*proof*⟩

lemma *card-greaterThanLessThan* [*simp*]: *card* {*l*<..<*u*} = *u* − *Suc l*
⟨*proof*⟩

lemma *ex-bij-betw-nat-finite*:
finite M ⟹ ∃ *h*. *bij-betw h* {*0*..<*card M*} *M*
⟨*proof*⟩

lemma *ex-bij-betw-finite-nat*:
finite M ⟹ ∃ *h*. *bij-betw h M* {*0*..<*card M*}
⟨*proof*⟩

29.5 Intervals of integers

lemma *atLeastLessThanPlusOne-atLeastAtMost-int*: {*l*..<*u*+*1*} = {*l*..(*u*::*int*)}
⟨*proof*⟩

lemma *atLeastPlusOneAtMost-greaterThanAtMost-int*: $\{l+1..u\} = \{l<..(u::int)\}$
⟨*proof*⟩

lemma *atLeastPlusOneLessThan-greaterThanLessThan-int*:
$\{l+1..<u\} = \{l<..<u::int\}$
⟨*proof*⟩

29.5.1 Finiteness

lemma *image-atLeastZeroLessThan-int*: $0 \leq u \Longrightarrow$
$\{(0::int)..<u\} = int\ `\ \{..<nat\ u\}$
⟨*proof*⟩

lemma *finite-atLeastZeroLessThan-int*: *finite* $\{(0::int)..<u\}$
⟨*proof*⟩

lemma *finite-atLeastLessThan-int* [*iff*]: *finite* $\{l..<u::int\}$
⟨*proof*⟩

lemma *finite-atLeastAtMost-int* [*iff*]: *finite* $\{l..(u::int)\}$
⟨*proof*⟩

lemma *finite-greaterThanAtMost-int* [*iff*]: *finite* $\{l<..(u::int)\}$
⟨*proof*⟩

lemma *finite-greaterThanLessThan-int* [*iff*]: *finite* $\{l<..<u::int\}$
⟨*proof*⟩

29.5.2 Cardinality

lemma *card-atLeastZeroLessThan-int*: *card* $\{(0::int)..<u\} = nat\ u$
⟨*proof*⟩

lemma *card-atLeastLessThan-int* [*simp*]: *card* $\{l..<u\} = nat\ (u - l)$
⟨*proof*⟩

lemma *card-atLeastAtMost-int* [*simp*]: *card* $\{l..u\} = nat\ (u - l + 1)$
⟨*proof*⟩

lemma *card-greaterThanAtMost-int* [*simp*]: *card* $\{l<..u\} = nat\ (u - l)$
⟨*proof*⟩

lemma *card-greaterThanLessThan-int* [*simp*]: *card* $\{l<..<u\} = nat\ (u - (l + 1))$
⟨*proof*⟩

29.6 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy

29.6.1 Disjoint Unions

Singletons and open intervals

lemma *ivl-disj-un-singleton*:
$\{l::'a::linorder\}\ Un\ \{l<..\} = \{l..\}$
$\{..<u\}\ Un\ \{u::'a::linorder\} = \{..u\}$
$(l::'a::linorder) < u ==> \{l\}\ Un\ \{l<..<u\} = \{l..<u\}$
$(l::'a::linorder) < u ==> \{l<..<u\}\ Un\ \{u\} = \{l<..u\}$
$(l::'a::linorder) <= u ==> \{l\}\ Un\ \{l<..u\} = \{l..u\}$
$(l::'a::linorder) <= u ==> \{l..<u\}\ Un\ \{u\} = \{l..u\}$
⟨*proof*⟩

One- and two-sided intervals

lemma *ivl-disj-un-one*:
$(l::'a::linorder) < u ==> \{..l\}\ Un\ \{l<..<u\} = \{..<u\}$
$(l::'a::linorder) <= u ==> \{..<l\}\ Un\ \{l..<u\} = \{..<u\}$
$(l::'a::linorder) <= u ==> \{..l\}\ Un\ \{l<..u\} = \{..u\}$
$(l::'a::linorder) <= u ==> \{..<l\}\ Un\ \{l..u\} = \{..u\}$
$(l::'a::linorder) <= u ==> \{l<..u\}\ Un\ \{u<..\} = \{l<..\}$
$(l::'a::linorder) < u ==> \{l<..<u\}\ Un\ \{u..\} = \{l<..\}$
$(l::'a::linorder) <= u ==> \{l..u\}\ Un\ \{u<..\} = \{l..\}$
$(l::'a::linorder) <= u ==> \{l..<u\}\ Un\ \{u..\} = \{l..\}$
⟨*proof*⟩

Two- and two-sided intervals

lemma *ivl-disj-un-two*:
$[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..<u\} = \{l<..<u\}$
$[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l<..m\}\ Un\ \{m<..<u\} = \{l<..<u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..<u\} = \{l..<u\}$
$[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l..m\}\ Un\ \{m<..<u\} = \{l..<u\}$
$[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..u\} = \{l<..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l<..m\}\ Un\ \{m<..u\} = \{l<..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..u\} = \{l..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..m\}\ Un\ \{m<..u\} = \{l..u\}$
⟨*proof*⟩

lemmas *ivl-disj-un = ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two*

29.6.2 Disjoint Intersections

Singletons and open intervals

lemma *ivl-disj-int-singleton*:
$\{l::'a::order\}\ Int\ \{l<..\} = \{\}$
$\{..<u\}\ Int\ \{u\} = \{\}$
$\{l\}\ Int\ \{l<..<u\} = \{\}$
$\{l<..<u\}\ Int\ \{u\} = \{\}$
$\{l\}\ Int\ \{l<..u\} = \{\}$
$\{l..<u\}\ Int\ \{u\} = \{\}$

⟨*proof*⟩

One- and two-sided intervals

lemma *ivl-disj-int-one*:
{..*l*::'*a*::*order*} *Int* {*l*<..<*u*} = {}
{..<*l*} *Int* {*l*..<*u*} = {}
{..*l*} *Int* {*l*<..*u*} = {}
{..<*l*} *Int* {*l*..*u*} = {}
{*l*<..*u*} *Int* {*u*<..} = {}
{*l*<..<*u*} *Int* {*u*..} = {}
{*l*..*u*} *Int* {*u*<..} = {}
{*l*..<*u*} *Int* {*u*..} = {}
⟨*proof*⟩

Two- and two-sided intervals

lemma *ivl-disj-int-two*:
{*l*::'*a*::*order*<..<*m*} *Int* {*m*..<*u*} = {}
{*l*<..*m*} *Int* {*m*<..<*u*} = {}
{*l*..<*m*} *Int* {*m*..<*u*} = {}
{*l*..*m*} *Int* {*m*<..<*u*} = {}
{*l*<..<*m*} *Int* {*m*..*u*} = {}
{*l*<..*m*} *Int* {*m*<..*u*} = {}
{*l*..<*m*} *Int* {*m*..*u*} = {}
{*l*..*m*} *Int* {*m*<..*u*} = {}
⟨*proof*⟩

lemmas *ivl-disj-int* = *ivl-disj-int-singleton ivl-disj-int-one ivl-disj-int-two*

29.6.3 Some Differences

lemma *ivl-diff* [*simp*]:
i ≤ *n* ⟹ {*i*..<*m*} − {*i*..<*n*} = {*n*..<(*m*::'*a*::*linorder*)}
⟨*proof*⟩

29.6.4 Some Subset Conditions

lemma *ivl-subset* [*simp*,*noatp*]:
({*i*..<*j*} ⊆ {*m*..<*n*}) = (*j* ≤ *i* | *m* ≤ *i* & *j* ≤ (*n*::'*a*::*linorder*))
⟨*proof*⟩

29.7 Summation indexed over intervals

syntax
-from-to-setsum :: *idt* ⇒ '*a* ⇒ '*a* ⇒ '*b* ⇒ '*b* ((*SUM* - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ '*a* ⇒ '*a* ⇒ '*b* ⇒ '*b* ((*SUM* - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ '*a* ⇒ '*b* ⇒ '*b* ((*SUM* -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ '*a* ⇒ '*b* ⇒ '*b* ((*SUM* -<=-./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)

-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -≤-./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -≤-./ -) [*0*,*0*,*10*] *10*)
syntax (*latex-sum* **output**)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑$^{-}_{-}$ = _ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑$^{<-}_{-}$ = _ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑$_{-}$ < _ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑$_{-}$ ≤ _ -) [*0*,*0*,*10*] *10*)

translations
∑ *x*=*a*..*b*. *t* == *setsum* (%*x*. *t*) {*a*..*b*}
∑ *x*=*a*..<*b*. *t* == *setsum* (%*x*. *t*) {*a*..<*b*}
∑ *i*≤*n*. *t* == *setsum* (λ*i*. *t*) {..*n*}
∑ *i*<*n*. *t* == *setsum* (λ*i*. *t*) {..<*n*}

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	LaTeX
∑ *x*∈{*a*..*b*}. *e*	∑ *x* = *a*..*b*. *e*	$\sum_{x\,=\,a}^{b}\ e$
∑ *x*∈{*a*..<*b*}. *e*	∑ *x* = *a*..<*b*. *e*	$\sum_{x\,=\,a}^{<b}\ e$
∑ *x*∈{..*b*}. *e*	∑ *x*≤*b*. *e*	$\sum_{x\ \leq\ b}\ e$
∑ *x*∈{..<*b*}. *e*	∑ *x*<*b*. *e*	$\sum_{x\ <\ b}\ e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode* = *latex-sum* in antiquotations). It is not the default LaTeX output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use ∑ *x* = *0*..<*n*. *e* rather than ∑ *x*<*n*. *e*: *setsum* may not provide all lemmas available for {*m*..<*n*} also in the special form for {..<*n*}.

This congruence rule should be used for sums over intervals as the standard theorem *setsum-cong* does not work well with the simplifier who adds the

unsimplified premise $x \in B$ to the context.

lemma *setsum-ivl-cong*:
$[\![a = c;\ b = d;\ !!x.\ [\![c \le x;\ x < d]\!] \Longrightarrow f\ x = g\ x]\!] \Longrightarrow$
$setsum\ f\ \{a..<b\} = setsum\ g\ \{c..<d\}$
⟨*proof*⟩

lemma *setsum-atMost-Suc*[*simp*]: $(\sum i \le Suc\ n.\ f\ i) = (\sum i \le n.\ f\ i) + f(Suc\ n)$
⟨*proof*⟩

lemma *setsum-lessThan-Suc*[*simp*]: $(\sum i < Suc\ n.\ f\ i) = (\sum i < n.\ f\ i) + f\ n$
⟨*proof*⟩

lemma *setsum-cl-ivl-Suc*[*simp*]:
$setsum\ f\ \{m..Suc\ n\} = (if\ Suc\ n < m\ then\ 0\ else\ setsum\ f\ \{m..n\} + f(Suc\ n))$
⟨*proof*⟩

lemma *setsum-op-ivl-Suc*[*simp*]:
$setsum\ f\ \{m..<Suc\ n\} = (if\ n < m\ then\ 0\ else\ setsum\ f\ \{m..<n\} + f(n))$
⟨*proof*⟩

lemma *setsum-add-nat-ivl*: $[\![m \le n;\ n \le p]\!] \Longrightarrow$
$setsum\ f\ \{m..<n\} + setsum\ f\ \{n..<p\} = setsum\ f\ \{m..<p::nat\}$
⟨*proof*⟩

lemma *setsum-diff-nat-ivl*:
fixes $f :: nat \Rightarrow 'a::ab\text{-}group\text{-}add$
shows $[\![m \le n;\ n \le p]\!] \Longrightarrow$
$setsum\ f\ \{m..<p\} - setsum\ f\ \{m..<n\} = setsum\ f\ \{n..<p\}$
⟨*proof*⟩

29.8 Shifting bounds

lemma *setsum-shift-bounds-nat-ivl*:
$setsum\ f\ \{m+k..<n+k\} = setsum\ (\%i.\ f(i + k))\{m..<n::nat\}$
⟨*proof*⟩

lemma *setsum-shift-bounds-cl-nat-ivl*:
$setsum\ f\ \{m+k..n+k\} = setsum\ (\%i.\ f(i + k))\{m..n::nat\}$
⟨*proof*⟩

corollary *setsum-shift-bounds-cl-Suc-ivl*:
$setsum\ f\ \{Suc\ m..Suc\ n\} = setsum\ (\%i.\ f(Suc\ i))\{m..n\}$
⟨*proof*⟩

corollary *setsum-shift-bounds-Suc-ivl*:
$setsum\ f\ \{Suc\ m..<Suc\ n\} = setsum\ (\%i.\ f(Suc\ i))\{m..<n\}$
⟨*proof*⟩

lemma *setsum-head*:
fixes n :: *nat*
assumes *mn*: $m <= n$
shows $(\sum x{\in}\{m..n\}.\ P\ x) = P\ m + (\sum x{\in}\{m<..n\}.\ P\ x)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *setsum-head-upt*:
fixes *m*::*nat*
assumes *m*: $0 < m$
shows $(\sum x{<}m.\ P\ x) = P\ 0 + (\sum x{\in}\{1..<m\}.\ P\ x)$
⟨*proof*⟩

29.9 The formula for geometric sums

lemma *geometric-sum*:
$x \ ^{\sim}= 1 \Longrightarrow (\sum i{=}0..<n.\ x\ \hat{}\ i) =$
$(x\ \hat{}\ n - 1) / (x - 1{::}'a{::}\{field, recpower\})$
⟨*proof*⟩

29.10 The formula for arithmetic sums

lemma *gauss-sum*:
$((1{::}'a{::}comm\text{-}semiring\text{-}1) + 1){*}(\sum i{\in}\{1..n\}.\ of\text{-}nat\ i) =$
$of\text{-}nat\ n{*}((of\text{-}nat\ n){+}1)$
⟨*proof*⟩

theorem *arith-series-general*:
$((1{::}'a{::}comm\text{-}semiring\text{-}1) + 1) * (\sum i{\in}\{..<n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1){*}d))$
⟨*proof*⟩

lemma *arith-series-nat*:
$Suc\ (Suc\ 0) * (\sum i{\in}\{..<n\}.\ a{+}i{*}d) = n * (a + (a{+}(n - 1){*}d))$
⟨*proof*⟩

lemma *arith-series-int*:
$(2{::}int) * (\sum i{\in}\{..<n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1){*}d))$
⟨*proof*⟩

lemma *sum-diff-distrib*:
fixes *P*::*nat*⇒*nat*
shows
$\forall x.\ Q\ x \leq P\ x \Longrightarrow$
$(\sum x{<}n.\ P\ x) - (\sum x{<}n.\ Q\ x) = (\sum x{<}n.\ P\ x - Q\ x)$
⟨*proof*⟩

end

30 Presburger: Decision Procedure for Presburger Arithmetic

theory *Presburger*
imports *Arith-Tools SetInterval*
uses
Tools/Qelim/cooper-data.ML
Tools/Qelim/generated-cooper.ML
Tools/Qelim/qelim.ML
(*Tools/Qelim/cooper.ML*)
(*Tools/Qelim/presburger.ML*)
begin

⟨*ML*⟩

30.1 The $-\infty$ and $+\infty$ Properties

lemma *minf*:
$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x < t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \leq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x > t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \geq t) = False$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})<z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})<z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x<z.\ F = F$
⟨*proof*⟩

lemma *pinf*:
$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x < t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \leq t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x > t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \geq t) = True$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})>z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})>z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x>z.\ F = F$
⟨*proof*⟩

lemma *inf-period*:
$[\![\forall x\ k.\ P\ x = P\ (x - k*D); \forall x\ k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \wedge Q\ x) = (P\ (x - k*D) \wedge Q\ (x - k*D))$
$[\![\forall x\ k.\ P\ x = P\ (x - k*D); \forall x\ k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \vee Q\ x) = (P\ (x - k*D) \vee Q\ (x - k*D))$
$(d::'a::\{comm\text{-}ring,Divides.div\})\ dvd\ D \Longrightarrow \forall x\ k.\ (d\ dvd\ x + t) = (d\ dvd\ (x - k*D) + t)$
$(d::'a::\{comm\text{-}ring,Divides.div\})\ dvd\ D \Longrightarrow \forall x\ k.\ (\neg d\ dvd\ x + t) = (\neg d\ dvd\ (x - k*D) + t)$
$\forall x\ k.\ F = F$
⟨*proof*⟩

30.2 The A and B sets

lemma *bset*:
$[\![\forall x.(\forall j \in \{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)\ ;$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x - D) \wedge Q\ (x - D))$
$[\![\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)\ ;$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \vee Q\ x) \longrightarrow (P(x - D) \vee Q\ (x - D))$
$[\![D{>}0;\ t - 1{\in}\ B]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
$D{>}0 \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$
$D{>}0 \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t))$
$[\![D{>}0\ ;\ t - 1 \in B]\!] \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x - D) + t))$
$d\ dvd\ D \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x - D) + t))$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow F \longrightarrow F$
⟨*proof*⟩

lemma *aset*:
$[\![\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow P\ x \longrightarrow P(x + D)\ ;$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x + D) \wedge Q\ (x + D))$
$[\![\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow P\ x \longrightarrow P(x + D)\ ;$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$

$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (P\ x \vee Q\ x) \longrightarrow (P(x + D) \vee Q\ (x + D))$
$[\![D{>}0;\ t + 1{\in}\ A]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (x = t) \longrightarrow (x + D = t))$
$[\![D{>}0\ ;\ t \in A]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$
$[\![D{>}0;\ t{\in}\ A]\!] \Longrightarrow (\forall (x{::}int).\ (\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (x < t) \longrightarrow (x + D < t))$
$[\![D{>}0;\ t + 1 \in A]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x + D) + t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x + D) + t))$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow F \longrightarrow F$
⟨*proof*⟩

30.3 Cooper's Theorem $-\infty$ and $+\infty$ Version

30.3.1 First some trivial facts about periodic sets or predicates

lemma *periodic-finite-ex*:
assumes *dpos*: $(0{::}int) < d$ **and** *modd*: $ALL\ x\ k.\ P\ x = P(x - k{*}d)$
shows $(EX\ x.\ P\ x) = (EX\ j : \{1..d\}.\ P\ j)$
(**is** *?LHS = ?RHS*)
⟨*proof*⟩

30.3.2 The $-\infty$ Version

lemma *decr-lemma*: $0 < (d{::}int) \Longrightarrow x - (abs(x{-}z){+}1) * d < z$
⟨*proof*⟩

lemma *incr-lemma*: $0 < (d{::}int) \Longrightarrow z < x + (abs(x{-}z){+}1) * d$
⟨*proof*⟩

theorem *int-induct*[*case-names base step1 step2*]:
assumes
base: $P(k{::}int)$ **and** *step1*: $\bigwedge i.\ [\![k \leq i;\ P\ i]\!] \Longrightarrow P(i{+}1)$ **and**
step2: $\bigwedge i.\ [\![k \geq i;\ P\ i]\!] \Longrightarrow P(i - 1)$
shows $P\ i$
⟨*proof*⟩

lemma *decr-mult-lemma*:
assumes *dpos*: $(0{::}int) < d$ **and** *minus*: $\forall x.\ P\ x \longrightarrow P(x - d)$ **and** *knneg*: $0 <= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x - k{*}d)$

⟨*proof*⟩

lemma *minusinfinity*:
assumes *dpos*: $0 < d$ **and**
P1eqP1: $ALL\ x\ k.\ P1\ x = P1(x - k*d)$ **and** *ePeqP1*: $EX\ z::int.\ ALL\ x.\ x < z \longrightarrow (P\ x = P1\ x)$
shows $(EX\ x.\ P1\ x) \longrightarrow (EX\ x.\ P\ x)$
⟨*proof*⟩

lemma *cpmi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists\, z.\ \forall\ x< z.\ P\ x = P'\ x$
and *nb*:$\forall\, x.(\forall\ j\in \{1..D\}.\ \forall\, (b::int) \in B.\ x \neq b+j) \text{ --> } P\ (x) \text{ --> } P\ (x - D)$
and *pd*: $\forall\ x\ k.\ P'\ x = P'\ (x-k*D)$
shows $(\exists\, x.\ P\ x) = ((\exists\ j\in \{1..D\}\ .\ P'\ j) \mid (\exists\ j \in \{1..D\}.\exists\ b\in B.\ P\ (b+j)))$
(**is** *?L* = (*?R1* ∨ *?R2*))
⟨*proof*⟩

30.3.3 The $+\infty$ Version

lemma *plusinfinity*:
assumes *dpos*: $(0::int) < d$ **and**
P1eqP1: $\forall\, x\ k.\ P'\ x = P'(x - k*d)$ **and** *ePeqP1*: $\exists\ z.\ \forall\ x>z.\ P\ x = P'\ x$
shows $(\exists\ x.\ P'\ x) \longrightarrow (\exists\ x.\ P\ x)$
⟨*proof*⟩

lemma *incr-mult-lemma*:
assumes *dpos*: $(0::int) < d$ **and** *plus*: $ALL\ x::int.\ P\ x \longrightarrow P(x + d)$ **and** *knneg*: $0 <= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x + k*d)$
⟨*proof*⟩

lemma *cppi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists\, z.\ \forall\ x> z.\ P\ x = P'\ x$
and *nb*:$\forall\, x.(\forall\ j\in \{1..D\}.\ \forall\, (b::int) \in A.\ x \neq b - j) \text{ --> } P\ (x) \text{ --> } P\ (x + D)$
and *pd*: $\forall\ x\ k.\ P'\ x= P'\ (x-k*D)$
shows $(\exists\, x.\ P\ x) = ((\exists\ j\in \{1..D\}\ .\ P'\ j) \mid (\exists\ j \in \{1..D\}.\exists\ b\in A.\ P\ (b - j)))$
(**is** *?L* = (*?R1* ∨ *?R2*))
⟨*proof*⟩

lemma *simp-from-to*: $\{i..j::int\} = (if\ j < i\ then\ \{\}\ else\ insert\ i\ \{i+1..j\})$
⟨*proof*⟩

theorem *unity-coeff-ex*: $(\exists\, (x::'a::\{semiring\text{-}0,Divides.div\}).\ P\ (l * x)) \equiv (\exists\, x.\ l\ dvd\ (x + 0) \wedge P\ x)$
⟨*proof*⟩

lemma *zdvd-mono*: **assumes** *not0*: $(k::int) \neq 0$

shows $((m::int)\ dvd\ t) \equiv (k*m\ dvd\ k*t)$
⟨*proof*⟩

lemma *uminus-dvd-conv*: $(d\ dvd\ (t::int)) \equiv (-d\ dvd\ t)\ (d\ dvd\ (t::int)) \equiv (d\ dvd\ -t)$
⟨*proof*⟩

Theorems for transforming predicates on nat to predicates on *int*

lemma *all-nat*: $(\forall x::nat.\ P\ x) = (\forall x::int.\ 0 <= x \longrightarrow P\ (nat\ x))$
⟨*proof*⟩

lemma *ex-nat*: $(\exists x::nat.\ P\ x) = (\exists x::int.\ 0 <= x \wedge P\ (nat\ x))$
⟨*proof*⟩

lemma *zdiff-int-split*: $P\ (int\ (x - y)) =$
$((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
⟨*proof*⟩

lemma *number-of1*: $(0::int) <= number\text{-}of\ n \Longrightarrow (0::int) <= number\text{-}of\ (Int.Bit0\ n) \wedge (0::int) <= number\text{-}of\ (Int.Bit1\ n)$
⟨*proof*⟩
lemma *number-of2*: $(0::int) <= Numeral0$ ⟨*proof*⟩
lemma *Suc-plus1*: $Suc\ n = n + 1$ ⟨*proof*⟩

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*: $(0 <= x \Longrightarrow P = P') \Longrightarrow (0 <= (x::int) \longrightarrow P) = (0 <= x \longrightarrow P')$ ⟨*proof*⟩

theorem *conj-le-cong*: $(0 <= x \Longrightarrow P = P') \Longrightarrow (0 <= (x::int) \wedge P) = (0 <= x \wedge P')$
⟨*proof*⟩
lemma *int-eq-number-of-eq*:
$(((number\text{-}of\ v)::int) = (number\text{-}of\ w)) = iszero\ ((number\text{-}of\ (v + (uminus\ w)))::int)$
⟨*proof*⟩

lemma *mod-eq0-dvd-iff*[*presburger*]: $(m::nat)\ mod\ n = 0 \longleftrightarrow n\ dvd\ m$
⟨*proof*⟩

lemma *zmod-eq0-zdvd-iff*[*presburger*]: $(m::int)\ mod\ n = 0 \longleftrightarrow n\ dvd\ m$
⟨*proof*⟩
declare *mod-1*[*presburger*]
declare *mod-0*[*presburger*]
declare *zmod-1*[*presburger*]
declare *zmod-zero*[*presburger*]
declare *zmod-self*[*presburger*]
declare *mod-self*[*presburger*]
declare *DIVISION-BY-ZERO-MOD*[*presburger*]

declare *nat-mod-div-trivial*[*presburger*]
declare *div-mod-equality2*[*presburger*]
declare *div-mod-equality*[*presburger*]
declare *mod-div-equality2*[*presburger*]
declare *mod-div-equality*[*presburger*]
declare *mod-mult-self1*[*presburger*]
declare *mod-mult-self2*[*presburger*]
declare *zdiv-zmod-equality2*[*presburger*]
declare *zdiv-zmod-equality*[*presburger*]
declare *mod2-Suc-Suc*[*presburger*]
lemma [*presburger*]: (*a*::*int*) *div 0 = 0* **and** [*presburger*]: *a mod 0 = a*
⟨*proof*⟩

⟨*ML*⟩

lemma [*presburger*]: *m mod 2* = (*1*::*nat*) ⟷ ¬ *2 dvd m* ⟨*proof*⟩
lemma [*presburger*]: *m mod 2 = Suc 0* ⟷ ¬ *2 dvd m* ⟨*proof*⟩
lemma [*presburger*]: *m mod* (*Suc* (*Suc 0*)) = (*1*::*nat*) ⟷ ¬ *2 dvd m* ⟨*proof*⟩
lemma [*presburger*]: *m mod* (*Suc* (*Suc 0*)) = *Suc 0* ⟷ ¬ *2 dvd m* ⟨*proof*⟩
lemma [*presburger*]: *m mod 2* = (*1*::*int*) ⟷ ¬ *2 dvd m* ⟨*proof*⟩

lemma *zdvd-period*:
  fixes *a d* :: *int*
  assumes *advdd*: *a dvd d*
  shows *a dvd* (*x* + *t*) ⟷ *a dvd* ((*x* + *c* ∗ *d*) + *t*)
⟨*proof*⟩

end

31 Refute: Refute

theory *Refute*
imports *Inductive*
uses
  Tools/prop-logic.ML
  Tools/sat-solver.ML
  Tools/refute.ML
  Tools/refute-isar.ML
begin

⟨*ML*⟩

```
(* ------------------------------------------------------------------------- *)
(* REFUTE                                                                    *)
(*                                                                           *)
(* We use a SAT solver to search for a (finite) model that refutes a given   *)
```

```
(* HOL formula.                                                              *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* NOTE                                                                     *)
(*                                                                          *)
(* I strongly recommend that you install a stand-alone SAT solver if you    *)
(* want to use 'refute'.  For details see 'HOL/Tools/sat_solver.ML'.  If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME' *)
(* in 'etc/settings'.                                                       *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* USAGE                                                                    *)
(*                                                                          *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples.  The supported   *)
(* parameters are explained below.                                          *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* CURRENT LIMITATIONS                                                      *)
(*                                                                          *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and          *)
(* inductively defined sets.  Constants are unfolded automatically, and sort *)
(* axioms are added as well.  Other, user-asserted axioms however are       *)
(* ignored.  Inductive datatypes and recursive functions are supported, but *)
(* may lead to spurious countermodels.                                      *)
(*                                                                          *)
(* The (space) complexity of the algorithm is non-elementary.               *)
(*                                                                          *)
(* Schematic type variables are not supported.                              *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* PARAMETERS                                                               *)
(*                                                                          *)
(* The following global parameters are currently supported (and required):  *)
(*                                                                          *)
(* Name          Type    Description                                        *)
(*                                                                          *)
(* "minsize"     int     Only search for models with size at least          *)
(*                       'minsize'.                                         *)
(* "maxsize"     int     If >0, only search for models with size at most    *)
(*                       'maxsize'.                                         *)
(* "maxvars"     int     If >0, use at most 'maxvars' boolean variables     *)
(*                       when transforming the term into a propositional    *)
(*                       formula.                                           *)
```

```
(* "maxtime"     int     If >0, terminate after at most 'maxtime' seconds.   *)
(*                       This value is ignored under some ML compilers.      *)
(* "satsolver"   string  Name of the SAT solver to be used.                  *)
(*                                                                           *)
(* See 'HOL/SAT.thy' for default values.                                     *)
(*                                                                           *)
(* The size of particular types can be specified in the form type=size       *)
(* (where 'type' is a string, and 'size' is an int).  Examples:              *)
(* "'a"=1                                                                    *)
(* "List.list"=2                                                             *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* FILES                                                                     *)
(*                                                                           *)
(* HOL/Tools/prop_logic.ML    Propositional logic                            *)
(* HOL/Tools/sat_solver.ML    SAT solvers                                    *)
(* HOL/Tools/refute.ML        Translation HOL -> propositional logic and     *)
(*                            Boolean assignment -> HOL model                *)
(* HOL/Tools/refute_isar.ML   Adds 'refute'/'refute_params' to Isabelle's    *)
(*                            syntax                                         *)
(* HOL/Refute.thy             This file: loads the ML files, basic setup,    *)
(*                            documentation                                  *)
(* HOL/SAT.thy                Sets default parameters                        *)
(* HOL/ex/RefuteExamples.thy  Examples                                       *)
(* ------------------------------------------------------------------------- *)
```

end

32 SAT: Reconstructing external resolution proofs for propositional logic

theory *SAT*
imports *Refute*
uses
Tools/cnf-funcs.ML
Tools/sat-funcs.ML
begin

Late package setup: default values for refute, see also theory *Refute*.

refute-params
[*itself=1*,
minsize=1,
maxsize=8,
maxvars=10000,
maxtime=60,

satsolver=auto]

⟨*ML*⟩

end

33 Recdef: TFL: recursive function definitions

theory *Recdef*
imports *FunDef*
uses
(*Tools/TFL/casesplit.ML*)
(*Tools/TFL/utils.ML*)
(*Tools/TFL/usyntax.ML*)
(*Tools/TFL/dcterm.ML*)
(*Tools/TFL/thms.ML*)
(*Tools/TFL/rules.ML*)
(*Tools/TFL/thry.ML*)
(*Tools/TFL/tfl.ML*)
(*Tools/TFL/post.ML*)
(*Tools/recdef-package.ML*)
begin

* This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-wfrec*: [| f==*wfrec* r H; $wf(r)$ |] ==> $f(a) = H$ (*cut* f r a) a
⟨*proof*⟩

lemma *tfl-wf-induct*: *ALL* R. *wf* R −−>
(*ALL* P. (*ALL* x. (*ALL* y. (y,x):R −−> P y) −−> P x) −−> (*ALL* x. P x))
⟨*proof*⟩

lemma *tfl-cut-apply*: *ALL* f R. (x,a):R −−> (*cut* f R a)(x) = f(x)
⟨*proof*⟩

lemma *tfl-wfrec*:
ALL M R f. (f=*wfrec* R M) −−> *wf* R −−> (*ALL* x. f x = M (*cut* f R x) x)
⟨*proof*⟩

lemma *tfl-eq-True*: (x = *True*) −−> x
⟨*proof*⟩

lemma *tfl-rev-eq-mp*: (x = y) −−> y −−> x
⟨*proof*⟩

lemma *tfl-simp-thm*: (x −−> y) −−> (x = x') −−> (x' −−> y)
⟨*proof*⟩

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
⟨*proof*⟩

lemma *tfl-imp-trans*: $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$
⟨*proof*⟩

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
⟨*proof*⟩

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$
⟨*proof*⟩

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \longrightarrow Q \Longrightarrow Q$
⟨*proof*⟩

⟨*ML*⟩

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong let-cong image-cong INT-cong UN-cong bex-cong ball-cong imp-cong

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-pred-nat
wf-same-fst
wf-empty

end

34 Datatype: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

theory *Datatype*
imports *Finite-Set Wellfounded*
begin

lemma *size-bool* [*code func*]:
size (*b*::*bool*) = *0* ⟨*proof*⟩

declare *prod.size* [*noatp*]

typedef (*Node*)
('*a*,'*b*) *node* = {*p*. *EX f x k*. *p* = (*f*::*nat*=>'*b*+*nat*, *x*::'*a*+*nat*) & *f k* = *Inr 0*}
— it is a subtype of (*nat*=>'*b*+*nat*) * ('*a*+*nat*)
⟨*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*⟩

34.1 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N ≠ 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*⟩

34.2 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*]

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*⟩

hides popular names

hide (**open**) *type node item*
hide (**open**) *const Push Node Atom Leaf Numb Lim Split Case*

35 Datatypes

35.1 Representing sums

rep-datatype *sum*
  distinct *Inl-not-Inr Inr-not-Inl*
  inject *Inl-eq Inr-eq*
  induction *sum-induct*

lemma *sum-case-KK*[*simp*]: *sum-case* (%*x*. *a*) (%*x*. *a*) = (%*x*. *a*)
  ⟨*proof*⟩

lemma *surjective-sum*: *sum-case* (%*x*::'*a*. *f* (*Inl x*)) (%*y*::'*b*. *f* (*Inr y*)) *s* = *f*(*s*)
  ⟨*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*⟩

lemma *sum-case-inject*:
  sum-case f1 f2 = *sum-case g1 g2* ==> (*f1* = *g1* ==> *f2* = *g2* ==> *P*) ==> *P*
⟨*proof*⟩

constdefs
Suml :: ('*a* => '*c*) => '*a* + '*b* => '*c*
Suml == (%*f*. *sum-case f arbitrary*)

Sumr :: ('*b* => '*c*) => '*a* + '*b* => '*c*
Sumr == *sum-case arbitrary*

lemma *Suml-inject*: *Suml f* = *Suml g* ==> *f* = *g*
⟨*proof*⟩

lemma *Sumr-inject*: *Sumr f* = *Sumr g* ==> *f* = *g*
⟨*proof*⟩

hide (**open**) *const Suml Sumr*

35.2 The option datatype

datatype '*a option* = *None* | *Some* '*a*

lemma *not-None-eq* [*iff*]: (*x* ~= *None*) = (*EX y*. *x* = *Some y*)
⟨*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 *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 *insert-None-conv-UNIV*: *insert None* (*range Some*) = *UNIV*
⟨*proof*⟩

instance *option* :: (*finite*) *finite*
⟨*proof*⟩

35.2.1 Operations

consts
the :: '*a option* => '*a*
primrec
the (*Some x*) = *x*

consts

o2s :: *'a option => 'a set*
primrec
o2s None = {}
o2s (*Some x*) = {*x*}

lemma *ospec* [*dest*]: (*ALL x:o2s A. P x*) ==> *A* = *Some x* ==> *P x*
⟨*proof*⟩

⟨*ML*⟩

lemma *elem-o2s* [*iff*]: (*x* : *o2s xo*) = (*xo* = *Some x*)
⟨*proof*⟩

lemma *o2s-empty-eq* [*simp*]: (*o2s xo* = {}) = (*xo* = *None*)
⟨*proof*⟩

definition
option-map :: (*'a* ⇒ *'b*) ⇒ *'a option* ⇒ *'b option*
where
[*code func del*]: *option-map* = (%*f y. case y of None => None* | *Some x => Some* (*f x*))

lemma *option-map-None* [*simp*, *code*]: *option-map f None* = *None*
⟨*proof*⟩

lemma *option-map-Some* [*simp*, *code*]: *option-map f* (*Some x*) = *Some* (*f x*)
⟨*proof*⟩

lemma *option-map-is-None* [*iff*]:
(*option-map f opt* = *None*) = (*opt* = *None*)
⟨*proof*⟩

lemma *option-map-eq-Some* [*iff*]:
(*option-map f xo* = *Some y*) = (*EX z. xo* = *Some z* & *f z* = *y*)
⟨*proof*⟩

lemma *option-map-comp*:
option-map f (*option-map g opt*) = *option-map* (*f o g*) *opt*
⟨*proof*⟩

lemma *option-map-o-sum-case* [*simp*]:
option-map f o sum-case g h = *sum-case* (*option-map f o g*) (*option-map f o h*)
⟨*proof*⟩

35.2.2 Code generator setup

definition
is-none :: *'a option* ⇒ *bool* **where**
is-none-none [*code post*, *symmetric*, *code inline*]: *is-none x* ⟷ *x* = *None*

lemma *is-none-code* [*code*]:
  shows *is-none None* ⟷ *True*
    and *is-none* (*Some x*) ⟷ *False*
  ⟨*proof*⟩

hide (**open**) *const is-none*

code-type *option*
  (*SML* - *option*)
  (*OCaml* - *option*)
  (*Haskell Maybe* -)

code-const *None* **and** *Some*
  (*SML NONE* **and** *SOME*)
  (*OCaml None* **and** *Some* -)
  (*Haskell Nothing* **and** *Just*)

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

code-const *op* = :: *'a*::*eq option* ⇒ *'a option* ⇒ *bool*
  (*Haskell* **infixl** *4* ==)

code-reserved *SML*
  option NONE SOME

code-reserved *OCaml*
  option None Some

code-modulename *SML*
  Datatype Nat

code-modulename *OCaml*
  Datatype Nat

code-modulename *Haskell*
  Datatype Nat

end

36 Extraction: Program extraction for HOL

theory *Extraction*
imports *Datatype*
uses *Tools/rewrite-hol-proof.ML*
begin

36.1 Setup

⟨*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-forall-def induct-implies-def induct-equal-def induct-conj-def
induct-atomize induct-rulify induct-rulify-fallback
True-implies-equals TrueE

datatype *sumbool* = *Left* | *Right*

36.2 Type of extracted program

extract-type
$typeof\ (Trueprop\ P) \equiv typeof\ P$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('Q))$

$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE(Null))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('P \Rightarrow 'Q))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$typeof\ (\forall x.\ P\ x) \equiv Type\ (TYPE(Null))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\forall x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$

$typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

36.3 Realizability

realizability

$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv (\forall x{::}'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P{::}bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

36.4 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*:
$P\ y\ x \Longrightarrow P\ (snd\ (x,\ y))\ (fst\ (x,\ y))\ \langle proof \rangle$

theorem *exE-realizer*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$
$\langle proof \rangle$

theorem *exE-realizer'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q\ \langle proof \rangle$

realizers
impI $(P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ P\ Q\ pq\ (h\colon -).\ allI \cdot - \bullet (\Lambda\ x.\ impI \cdot - \cdot - \cdot - \bullet (h \cdot x))$

impI (P): *Null*
$\Lambda\ P\ Q\ (h\colon -).\ allI \cdot - \bullet (\Lambda\ x.\ impI \cdot - \cdot - \cdot - \bullet (h \cdot x))$

impI (Q): $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ impI \cdot - \cdot -$

impI: *Null impI*

mp $(P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ P\ Q\ pq\ (h\colon -)\ p.\ mp \cdot - \cdot - \cdot - \bullet (spec \cdot - \cdot p \bullet h)$

mp (P): *Null*
$\Lambda\ P\ Q\ (h\colon -)\ p.\ mp \cdot - \cdot - \cdot - \bullet (spec \cdot - \cdot p \bullet h)$

mp (Q): $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ mp \cdot - \cdot -$

mp: *Null mp*

allI (P): $\lambda p.\ p\ \Lambda\ P\ p.\ allI \cdot -$

allI: *Null allI*

spec (P): $\lambda x\ p.\ p\ x\ \Lambda\ P\ x\ p.\ spec \cdot - \cdot x$

spec: *Null spec*

exI (P): $\lambda x\ p.\ (x,\ p)\ \Lambda\ P\ x\ p.\ exI\text{-}realizer \cdot P \cdot p \cdot x$

exI: $\lambda x.\ x\ \Lambda\ P\ x\ (h\colon -).\ h$

exE $(P,\ Q)$: $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
$\Lambda\ P\ Q\ p\ (h\colon -)\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \bullet h$

exE (P): *Null*
$\Lambda\ P\ Q\ p.\ exE\text{-}realizer' \cdot - \cdot - \cdot - \cdot -$

exE (*Q*): $\lambda x\ pq.\ pq\ x$
  $\Lambda\ P\ Q\ x\ (h1\text{: -})\ pq\ (h2\text{: -}).\ h2 \cdot x \bullet h1$

exE: *Null*
  $\Lambda\ P\ Q\ x\ (h1\text{: -})\ (h2\text{: -}).\ h2 \cdot x \bullet h1$

conjI (*P*, *Q*): *Pair*
  $\Lambda\ P\ Q\ p\ (h\text{: -})\ q.\ conjI\text{-}realizer \cdot P \cdot p \cdot Q \cdot q \bullet h$

conjI (*P*): $\lambda p.\ p$
  $\Lambda\ P\ Q\ p.\ conjI \cdot - \cdot -$

conjI (*Q*): $\lambda q.\ q$
  $\Lambda\ P\ Q\ (h\text{: -})\ q.\ conjI \cdot - \cdot - \bullet h$

conjI: *Null conjI*

conjunct1 (*P*, *Q*): *fst*
  $\Lambda\ P\ Q\ pq.\ conjunct1 \cdot - \cdot -$

conjunct1 (*P*): $\lambda p.\ p$
  $\Lambda\ P\ Q\ p.\ conjunct1 \cdot - \cdot -$

conjunct1 (*Q*): *Null*
  $\Lambda\ P\ Q\ q.\ conjunct1 \cdot - \cdot -$

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
  $\Lambda\ P\ Q\ pq.\ conjunct2 \cdot - \cdot -$

conjunct2 (*P*): *Null*
  $\Lambda\ P\ Q\ p.\ conjunct2 \cdot - \cdot -$

conjunct2 (*Q*): $\lambda p.\ p$
  $\Lambda\ P\ Q\ p.\ conjunct2 \cdot - \cdot -$

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
  $\Lambda\ P\ Q\ p.\ iffD2 \cdot - \cdot - \bullet (sum.cases\text{-}1 \cdot P \cdot - \cdot p)$

disjI1 (*P*): *Some*
  $\Lambda\ P\ Q\ p.\ iffD2 \cdot - \cdot - \bullet (option.cases\text{-}2 \cdot - \cdot P \cdot p)$

disjI1 (*Q*): *None*
  $\Lambda\ P\ Q.\ iffD2 \cdot - \cdot - \bullet (option.cases\text{-}1 \cdot - \cdot -)$

disjI1: *Left*
  Λ *P Q*. *iffD2* · - · - · (*sumbool.cases-1* · - · -)

disjI2 (*P*, *Q*): *Inr*
  Λ *Q P q*. *iffD2* · - · - · (*sum.cases-2* · - · *Q* · *q*)

disjI2 (*P*): *None*
  Λ *Q P*. *iffD2* · - · - · (*option.cases-1* · - · -)

disjI2 (*Q*): *Some*
  Λ *Q P q*. *iffD2* · - · - · (*option.cases-2* · - · *Q* · *q*)

disjI2: *Right*
  Λ *Q P*. *iffD2* · - · - · (*sumbool.cases-2* · - · -)

disjE (*P*, *Q*, *R*): λ*pq pr qr*.
  (*case pq of Inl p* $\Rightarrow$ *pr p* | *Inr q* $\Rightarrow$ *qr q*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*Q*, *R*): λ*pq pr qr*.
  (*case pq of None* $\Rightarrow$ *pr* | *Some q* $\Rightarrow$ *qr q*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer2 · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*P*, *R*): λ*pq pr qr*.
  (*case pq of None* $\Rightarrow$ *qr* | *Some p* $\Rightarrow$ *pr p*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr* (*h3*: -).
    disjE-realizer2 · - · - · *pq* · *R* · *qr* · *pr* · *h1* · *h3* · *h2*

disjE (*R*): λ*pq pr qr*.
  (*case pq of Left* $\Rightarrow$ *pr* | *Right* $\Rightarrow$ *qr*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer3 · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*P*, *Q*): *Null*
  Λ *P Q R pq*. *disjE-realizer* · - · - · *pq* · (λ*x*. *R*) · - · -

disjE (*Q*): *Null*
  Λ *P Q R pq*. *disjE-realizer2* · - · - · *pq* · (λ*x*. *R*) · - · -

disjE (*P*): *Null*
  Λ *P Q R pq* (*h1*: -) (*h2*: -) (*h3*: -).
    disjE-realizer2 · - · - · *pq* · (λ*x*. *R*) · - · - · *h1* · *h3* · *h2*

disjE: *Null*
  Λ *P Q R pq*. *disjE-realizer3* · - · - · *pq* · (λ*x*. *R*) · - · -

FalseE (*P*): *arbitrary*

$\Lambda\ P.\ \mathit{FalseE} \cdot \text{-}$

FalseE: *Null FalseE*

notI (*P*): *Null*
$\Lambda\ P\ (h\colon \text{-}).\ \mathit{allI} \cdot \text{-} \cdot (\Lambda\ x.\ \mathit{notI} \cdot \text{-} \cdot (h \cdot x))$

notI: *Null notI*

notE (*P*, *R*): $\lambda p.$ *arbitrary*
$\Lambda\ P\ R\ (h\colon \text{-})\ p.\ \mathit{notE} \cdot \text{-} \cdot \text{-} \cdot (\mathit{spec} \cdot \text{-} \cdot p \cdot h)$

notE (*P*): *Null*
$\Lambda\ P\ R\ (h\colon \text{-})\ p.\ \mathit{notE} \cdot \text{-} \cdot \text{-} \cdot (\mathit{spec} \cdot \text{-} \cdot p \cdot h)$

notE (*R*): *arbitrary*
$\Lambda\ P\ R.\ \mathit{notE} \cdot \text{-} \cdot \text{-}$

notE: *Null notE*

subst (*P*): $\lambda s\ t\ ps.\ ps$
$\Lambda\ s\ t\ P\ (h\colon \text{-})\ ps.\ \mathit{subst} \cdot s \cdot t \cdot P\ ps \cdot h$

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
$\Lambda\ Q\ P\ pq\ (h\colon \text{-})\ p.$
$\quad \mathit{mp} \cdot \text{-} \cdot \text{-} \cdot (\mathit{spec} \cdot \text{-} \cdot p \cdot (\mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot h))$

iffD1 (*P*): $\lambda p.\ p$
$\Lambda\ Q\ P\ p\ (h\colon \text{-}).\ \mathit{mp} \cdot \text{-} \cdot \text{-} \cdot (\mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot h)$

iffD1 (*Q*): *Null*
$\Lambda\ Q\ P\ q1\ (h\colon \text{-})\ q2.$
$\quad \mathit{mp} \cdot \text{-} \cdot \text{-} \cdot (\mathit{spec} \cdot \text{-} \cdot q2 \cdot (\mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot h))$

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
$\Lambda\ P\ Q\ pq\ (h\colon \text{-})\ q.$
$\quad \mathit{mp} \cdot \text{-} \cdot \text{-} \cdot (\mathit{spec} \cdot \text{-} \cdot q \cdot (\mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot h))$

iffD2 (*P*): $\lambda p.\ p$
$\Lambda\ P\ Q\ p\ (h\colon \text{-}).\ \mathit{mp} \cdot \text{-} \cdot \text{-} \cdot (\mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot h)$

iffD2 (*Q*): *Null*
$\Lambda\ P\ Q\ q1\ (h\colon \text{-})\ q2.$
$\quad \mathit{mp} \cdot \text{-} \cdot \text{-} \cdot (\mathit{spec} \cdot \text{-} \cdot q2 \cdot (\mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot h))$

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
 Λ *P Q pq* (*h1* : -) *qp* (*h2* : -). *conjI-realizer* ·
 (λ*pq*. ∀ *x*. *P x* ⟶ *Q* (*pq x*)) · *pq* ·
 (λ*qp*. ∀ *x*. *Q x* ⟶ *P* (*qp x*)) · *qp* ·
 (*allI* · - · (Λ *x*. *impI* · - · - · - · (*h1* · *x*))) ·
 (*allI* · - · (Λ *x*. *impI* · - · - · - · (*h2* · *x*)))

iffI (*P*): λ*p*. *p*
 Λ *P Q* (*h1* : -) *p* (*h2* : -). *conjI* · - · - · -
 (*allI* · - · (Λ *x*. *impI* · - · - · - · (*h1* · *x*))) ·
 (*impI* · - · - · - · *h2*)

iffI (*Q*): λ*q*. *q*
 Λ *P Q q* (*h1* : -) (*h2* : -). *conjI* · - · - · -
 (*impI* · - · - · - · *h1*) ·
 (*allI* · - · (Λ *x*. *impI* · - · - · - · (*h2* · *x*)))

iffI: *Null iffI*

end

37 Relation-Power: Powers of Relations and Functions

theory *Relation-Power*
imports *Power Transitive-Closure*
begin

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

overloading
 relpow ≡ *power* :: (′*a* × ′*a*) *set* ⇒ *nat* ⇒ (′*a* × ′*a*) *set* (**unchecked**)
begin

R ^ *n* = *R O* ... *O R*, the n-fold composition of *R*

primrec *relpow* **where**
 (*R* :: (′*a* × ′*a*) *set*) ^ *0* = *Id*
 | (*R* :: (′*a* × ′*a*) *set*) ^ *Suc n* = *R O* (*R* ^ *n*)

end

overloading

funpow ≡ *power* :: (′*a* ⇒ ′*a*) ⇒ *nat* ⇒ ′*a* ⇒ ′*a* (**unchecked**)
begin

f ^ *n* = *f o ... o f*, the n-fold composition of *f*

primrec *funpow* **where**
(*f* :: ′*a* ⇒ ′*a*) ^ 0 = *id*
| (*f* :: ′*a* ⇒ ′*a*) ^ *Suc n* = *f o* (*f* ^ *n*)

end

WARNING: due to the limits of Isabelle's type classes, exponentiation on functions and relations has too general a domain, namely (′*a* × ′*b*) *set* and ′*a* ⇒ ′*b*. Explicit type constraints may therefore be necessary. For example, *range* (*f* ^ *n*) = *A* and *Range* (*R* ^ *n*) = *B* need constraints.

Circumvent this problem for code generation:

primrec
fun-pow :: *nat* ⇒ (′*a* ⇒ ′*a*) ⇒ ′*a* ⇒ ′*a*
where
fun-pow 0 f = *id*
| *fun-pow* (*Suc n*) *f* = *f o fun-pow n f*

lemma *funpow-fun-pow* [*code inline*]: *f* ^ *n* = *fun-pow n f*
⟨*proof*⟩

lemma *funpow-add*: *f* ^ (*m*+*n*) = *f*^*m o f*^*n*
⟨*proof*⟩

lemma *funpow-swap1*: *f*((*f*^*n*) *x*) = (*f*^*n*)(*f x*)
⟨*proof*⟩

lemma *rel-pow-1* [*simp*]:
fixes *R* :: (′*a*∗′*a*)*set*
shows *R*^*1* = *R*
⟨*proof*⟩

lemma *rel-pow-0-I*: (*x*,*x*) : *R*^*0*
⟨*proof*⟩

lemma *rel-pow-Suc-I*: [| (*x*,*y*) : *R*^*n*; (*y*,*z*):*R* |] ==> (*x*,*z*):*R*^(*Suc n*)
⟨*proof*⟩

lemma *rel-pow-Suc-I2*:
(*x*, *y*) : *R* ⟹ (*y*, *z*) : *R*^*n* ⟹ (*x*,*z*) : *R*^(*Suc n*)
⟨*proof*⟩

lemma *rel-pow-0-E*: [| (*x*,*y*) : *R*^*0*; *x*=*y* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *rel-pow-Suc-E*:
$[\!|\ (x,z) : R\hat{}(Suc\ n);\ \ !!y.\ [\!|\ (x,y) : R\hat{}n;\ (y,z) : R\ |\!] ==> P\ |\!] ==> P$
⟨*proof*⟩

lemma *rel-pow-E*:
$[\!|\ (x,z) : R\hat{}n;\ \ [\!|\ n=0;\ x = z\ |\!] ==> P;$
$\quad !!y\ m.\ [\!|\ n = Suc\ m;\ (x,y) : R\hat{}m;\ (y,z) : R\ |\!] ==> P$
$\ |\!] ==> P$
⟨*proof*⟩

lemma *rel-pow-Suc-D2*:
$(x,\ z) : R\hat{}(Suc\ n) \Longrightarrow (\exists\, y.\ (x,y) : R\ \&\ (y,z) : R\hat{}n)$
⟨*proof*⟩

lemma *rel-pow-Suc-D2'*:
$\forall\, x\ y\ z.\ (x,y) : R\hat{}n\ \&\ (y,z) : R\ --> (\exists\, w.\ (x,w) : R\ \&\ (w,z) : R\hat{}n)$
⟨*proof*⟩

lemma *rel-pow-E2*:
$[\!|\ (x,z) : R\hat{}n;\ \ [\!|\ n=0;\ x = z\ |\!] ==> P;$
$\quad !!y\ m.\ [\!|\ n = Suc\ m;\ (x,y) : R;\ (y,z) : R\hat{}m\ |\!] ==> P$
$\ |\!] ==> P$
⟨*proof*⟩

lemma *rtrancl-imp-UN-rel-pow*: $!!p.\ p{:}R\hat{}* ==> p : (UN\ n.\ R\hat{}n)$
⟨*proof*⟩

lemma *rel-pow-imp-rtrancl*: $!!p.\ p{:}R\hat{}n ==> p{:}R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-is-UN-rel-pow*: $R\hat{}* = (UN\ n.\ R\hat{}n)$
⟨*proof*⟩

lemma *trancl-power*:
$x \in r\hat{}+ = (\exists\, n > 0.\ x \in r\hat{}n)$
⟨*proof*⟩

lemma *single-valued-rel-pow*:
$!!r{::}('a * 'a)set.\ single\text{-}valued\ r ==> single\text{-}valued\ (r\hat{}n)$
⟨*proof*⟩

⟨*ML*⟩

end

38 Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *Nat Wellfounded*
uses (*Tools/meson.ML*) (*Tools/specification-package.ML*)
begin

38.1 Hilbert's epsilon

axiomatization
  Eps :: (*'a* => *bool*) => *'a*
where
  someI: *P x* ==> *P* (*Eps P*)

syntax (*epsilon*)
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3ϵ -./ -*) [*0*, *10*] *10*)
syntax (*HOL*)
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3@ -./ -*) [*0*, *10*] *10*)
syntax
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3SOME -./ -*) [*0*, *10*] *10*)
translations
  SOME x. P == *CONST Eps* (%*x. P*)

⟨*ML*⟩

constdefs
  inv :: (*'a* => *'b*) => (*'b* => *'a*)
  inv(*f* :: *'a* => *'b*) == %*y. SOME x. f x* = *y*

  Inv :: *'a set* => (*'a* => *'b*) => (*'b* => *'a*)
  Inv A f == %*x. SOME y. y* ∈ *A* & *f y* = *x*

38.2 Hilbert's Epsilon-operator

Easier to apply than *someI* if the witness comes from an existential formula

lemma *someI-ex* [*elim?*]: ∃ *x. P x* ==> *P* (*SOME x. P x*)
⟨*proof*⟩

Easier to apply than *someI* because the conclusion has only one occurrence of *P*.

lemma *someI2*: [| *P a*; !!*x. P x* ==> *Q x* |] ==> *Q* (*SOME x. P x*)
⟨*proof*⟩

Easier to apply than *someI2* if the witness comes from an existential formula

lemma *someI2-ex*: [| ∃ *a. P a*; !!*x. P x* ==> *Q x* |] ==> *Q* (*SOME x. P x*)
⟨*proof*⟩

lemma *some-equality* [*intro*]:
[| *P a*; !!*x*. *P x* ==> *x*=*a* |] ==> (*SOME x*. *P x*) = *a*
⟨*proof*⟩

lemma *some1-equality*: [| *EX*!*x*. *P x*; *P a* |] ==> (*SOME x*. *P x*) = *a*
⟨*proof*⟩

lemma *some-eq-ex*: *P* (*SOME x*. *P x*) = (∃ *x*. *P x*)
⟨*proof*⟩

lemma *some-eq-trivial* [*simp*]: (*SOME y*. *y*=*x*) = *x*
⟨*proof*⟩

lemma *some-sym-eq-trivial* [*simp*]: (*SOME y*. *x*=*y*) = *x*
⟨*proof*⟩

38.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

lemma *choice*: ∀ *x*. ∃ *y*. *Q x y* ==> ∃ *f*. ∀ *x*. *Q x* (*f x*)
⟨*proof*⟩

lemma *bchoice*: ∀ *x*∈*S*. ∃ *y*. *Q x y* ==> ∃ *f*. ∀ *x*∈*S*. *Q x* (*f x*)
⟨*proof*⟩

38.4 Function Inverse

lemma *inv-id* [*simp*]: *inv id* = *id*
⟨*proof*⟩

A one-to-one function has an inverse.

lemma *inv-f-f* [*simp*]: *inj f* ==> *inv f* (*f x*) = *x*
⟨*proof*⟩

lemma *inv-f-eq*: [| *inj f*; *f x* = *y* |] ==> *inv f y* = *x*
⟨*proof*⟩

lemma *inj-imp-inv-eq*: [| *inj f*; ∀ *x*. *f*(*g x*) = *x* |] ==> *inv f* = *g*
⟨*proof*⟩

But is it useful?

lemma *inj-transfer*:
assumes *injf*: *inj f* **and** *minor*: !!*y*. *y* ∈ *range*(*f*) ==> *P*(*inv f y*)
shows *P x*
⟨*proof*⟩

lemma *inj-iff*: $(inj\ f) = (inv\ f\ o\ f = id)$
⟨*proof*⟩

lemma *inv-o-cancel*[*simp*]: $inj\ f \Longrightarrow inv\ f\ o\ f = id$
⟨*proof*⟩

lemma *o-inv-o-cancel*[*simp*]: $inj\ f \Longrightarrow g\ o\ inv\ f\ o\ f = g$
⟨*proof*⟩

lemma *inv-image-cancel*[*simp*]:
$inj\ f \Longrightarrow inv\ f\ `\ f\ `\ S = S$
⟨*proof*⟩

lemma *inj-imp-surj-inv*: $inj\ f \Longrightarrow surj\ (inv\ f)$
⟨*proof*⟩

lemma *f-inv-f*: $y \in range(f) \Longrightarrow f(inv\ f\ y) = y$
⟨*proof*⟩

lemma *surj-f-inv-f*: $surj\ f \Longrightarrow f(inv\ f\ y) = y$
⟨*proof*⟩

lemma *inv-injective*:
assumes *eq*: $inv\ f\ x = inv\ f\ y$
and *x*: x: $range\ f$
and *y*: y: $range\ f$
shows $x{=}y$
⟨*proof*⟩

lemma *inj-on-inv*: $A <= range(f) \Longrightarrow inj\text{-}on\ (inv\ f)\ A$
⟨*proof*⟩

lemma *surj-imp-inj-inv*: $surj\ f \Longrightarrow inj\ (inv\ f)$
⟨*proof*⟩

lemma *surj-iff*: $(surj\ f) = (f\ o\ inv\ f = id)$
⟨*proof*⟩

lemma *surj-imp-inv-eq*: $[|\ surj\ f; \forall x.\ g(f\ x) = x\ |] \Longrightarrow inv\ f = g$
⟨*proof*⟩

lemma *bij-imp-bij-inv*: $bij\ f \Longrightarrow bij\ (inv\ f)$
⟨*proof*⟩

lemma *inv-equality*: $[|\ !!x.\ g\ (f\ x) = x;\ \ !!y.\ f\ (g\ y) = y\ |] \Longrightarrow inv\ f = g$
⟨*proof*⟩

lemma *inv-inv-eq*: $bij\ f \Longrightarrow inv\ (inv\ f) = f$
⟨*proof*⟩

lemma *o-inv-distrib*: [| *bij f*; *bij g* |] ==> *inv* (*f o g*) = *inv g o inv f*
⟨*proof*⟩

lemma *image-surj-f-inv-f*: *surj f* ==> *f* ‘ (*inv f* ‘ *A*) = *A*
⟨*proof*⟩

lemma *image-inv-f-f*: *inj f* ==> (*inv f*) ‘ (*f* ‘ *A*) = *A*
⟨*proof*⟩

lemma *inv-image-comp*: *inj f* ==> *inv f* ‘ (*f*‘*X*) = *X*
⟨*proof*⟩

lemma *bij-image-Collect-eq*: *bij f* ==> *f* ‘ *Collect P* = {*y*. *P* (*inv f y*)}
⟨*proof*⟩

lemma *bij-vimage-eq-inv-image*: *bij f* ==> *f* −‘ *A* = *inv f* ‘ *A*
⟨*proof*⟩

38.5 Inverse of a PI-function (restricted domain)

lemma *Inv-f-f*: [| *inj-on f A*; *x* ∈ *A* |] ==> *Inv A f* (*f x*) = *x*
⟨*proof*⟩

lemma *f-Inv-f*: *y* ∈ *f*‘*A* ==> *f* (*Inv A f y*) = *y*
⟨*proof*⟩

lemma *Inv-injective*:
assumes *eq*: *Inv A f x* = *Inv A f y*
and *x*: *x*: *f*‘*A*
and *y*: *y*: *f*‘*A*
shows *x*=*y*
⟨*proof*⟩

lemma *inj-on-Inv*: *B* <= *f*‘*A* ==> *inj-on* (*Inv A f*) *B*
⟨*proof*⟩

lemma *Inv-mem*: [| *f* ‘ *A* = *B*; *x* ∈ *B* |] ==> *Inv A f x* ∈ *A*
⟨*proof*⟩

lemma *Inv-f-eq*: [| *inj-on f A*; *f x* = *y*; *x* ∈ *A* |] ==> *Inv A f y* = *x*
⟨*proof*⟩

lemma *Inv-comp*:
[| *inj-on f* (*g* ‘ *A*); *inj-on g A*; *x* ∈ *f* ‘ *g* ‘ *A* |] ==>
Inv A (*f o g*) *x* = (*Inv A g o Inv* (*g* ‘ *A*) *f*) *x*

⟨*proof*⟩

lemma *bij-betw-Inv*: *bij-betw f A B* ⟹ *bij-betw* (*Inv A f*) *B A*
⟨*proof*⟩

38.6 Other Consequences of Hilbert's Epsilon

Hilbert's Epsilon and the *split* Operator

Looping simprule

lemma *split-paired-Eps*: (*SOME x. P x*) = (*SOME* (*a*,*b*). *P*(*a*,*b*))
⟨*proof*⟩

lemma *Eps-split*: *Eps* (*split P*) = (*SOME xy. P* (*fst xy*) (*snd xy*))
⟨*proof*⟩

lemma *Eps-split-eq* [*simp*]: (@(*x'*,*y'*). *x* = *x'* & *y* = *y'*) = (*x*,*y*)
⟨*proof*⟩

A relation is wellfounded iff it has no infinite descending chain

lemma *wf-iff-no-infinite-down-chain*:
wf r = (~(∃*f*. ∀ *i*. (*f*(*Suc i*),*f i*) ∈ *r*))
⟨*proof*⟩

A dynamically-scoped fact for TFL

lemma *tfl-some*: ∀ *P x*. *P x* −−> *P* (*Eps P*)
⟨*proof*⟩

38.7 Least value operator

constdefs
LeastM :: ['*a* => '*b*::*ord*, '*a* => *bool*] => '*a*
LeastM m P == *SOME x*. *P x* & (∀ *y*. *P y* −−> *m x* <= *m y*)

syntax
-*LeastM* :: [*pttrn*, '*a* => '*b*::*ord*, *bool*] => '*a* (*LEAST* - *WRT* -. - [*0*, *4*, *10*] *10*)
translations
LEAST x WRT m. *P* == *LeastM m* (%*x*. *P*)

lemma *LeastMI2*:
P x ==> (!!*y*. *P y* ==> *m x* <= *m y*)
==> (!!*x*. *P x* ==> ∀ *y*. *P y* −−> *m x* ≤ *m y* ==> *Q x*)
==> *Q* (*LeastM m P*)
⟨*proof*⟩

lemma *LeastM-equality*:
P k ==> (!!*x*. *P x* ==> *m k* <= *m x*)
==> *m* (*LEAST x WRT m*. *P x*) = (*m k*::'*a*::*order*)

⟨*proof*⟩

lemma *wf-linord-ex-has-least*:
wf r ==> ∀ x y. ((x,y):r^+) = ((y,x)~:r^*) ==> P k
==> ∃ x. P x & (!y. P y --> (m x,m y):r^*)
⟨*proof*⟩

lemma *ex-has-least-nat*:
P k ==> ∃ x. P x & (∀ y. P y --> m x <= (m y::nat))
⟨*proof*⟩

lemma *LeastM-nat-lemma*:
P k ==> P (LeastM m P) & (∀ y. P y --> m (LeastM m P) <= (m y::nat))
⟨*proof*⟩

lemmas *LeastM-natI = LeastM-nat-lemma [THEN conjunct1, standard]*

lemma *LeastM-nat-le*: *P x ==> m (LeastM m P) <= (m x::nat)*
⟨*proof*⟩

38.8 Greatest value operator

constdefs
GreatestM :: ['a => 'b::ord, 'a => bool] => 'a
GreatestM m P == SOME x. P x & (∀ y. P y --> m y <= m x)

Greatest :: ('a::ord => bool) => 'a (**binder** *GREATEST 10*)
Greatest == GreatestM (%x. x)

syntax
-GreatestM :: [pttrn, 'a=>'b::ord, bool] => 'a
(GREATEST - WRT -. - [0, 4, 10] 10)

translations
GREATEST x WRT m. P == GreatestM m (%x. P)

lemma *GreatestMI2*:
P x ==> (!!y. P y ==> m y <= m x)
==> (!!x. P x ==> ∀ y. P y --> m y ≤ m x ==> Q x)
==> Q (GreatestM m P)
⟨*proof*⟩

lemma *GreatestM-equality*:
P k ==> (!!x. P x ==> m x <= m k)
==> m (GREATEST x WRT m. P x) = (m k::'a::order)
⟨*proof*⟩

lemma *Greatest-equality*:
P (k::'a::order) ==> (!!x. P x ==> x <= k) ==> (GREATEST x. P x) = k

⟨*proof*⟩

lemma *ex-has-greatest-nat-lemma*:
$P\ k \Longrightarrow \forall x.\ P\ x \longrightarrow (\exists y.\ P\ y\ \&\ \sim ((m\ y{::}nat) <= m\ x))$
$\Longrightarrow \exists y.\ P\ y\ \&\ \sim (m\ y < m\ k + n)$
⟨*proof*⟩

lemma *ex-has-greatest-nat*:
$P\ k \Longrightarrow \forall y.\ P\ y \longrightarrow m\ y < b$
$\Longrightarrow \exists x.\ P\ x\ \&\ (\forall y.\ P\ y \longrightarrow (m\ y{::}nat) <= m\ x)$
⟨*proof*⟩

lemma *GreatestM-nat-lemma*:
$P\ k \Longrightarrow \forall y.\ P\ y \longrightarrow m\ y < b$
$\Longrightarrow P\ (GreatestM\ m\ P)\ \&\ (\forall y.\ P\ y \longrightarrow (m\ y{::}nat) <= m\ (GreatestM\ m\ P))$
⟨*proof*⟩

lemmas *GreatestM-natI* = *GreatestM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *GreatestM-nat-le*:
$P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow m\ y < b$
$\Longrightarrow (m\ x{::}nat) <= m\ (GreatestM\ m\ P)$
⟨*proof*⟩

Specialization to *GREATEST*.

lemma *GreatestI*: $P\ (k{::}nat) \Longrightarrow \forall y.\ P\ y \longrightarrow y < b \Longrightarrow P\ (GREATEST\ x.\ P\ x)$
⟨*proof*⟩

lemma *Greatest-le*:
$P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow y < b \Longrightarrow (x{::}nat) <= (GREATEST\ x.\ P\ x)$
⟨*proof*⟩

38.9 The Meson proof procedure

38.9.1 Negation Normal Form

de Morgan laws

lemma *meson-not-conjD*: $\sim(P\&Q) \Longrightarrow \sim P \mid \sim Q$
and *meson-not-disjD*: $\sim(P|Q) \Longrightarrow \sim P\ \&\ \sim Q$
and *meson-not-notD*: $\sim\sim P \Longrightarrow P$
and *meson-not-allD*: $!!P.\ \sim(\forall x.\ P(x)) \Longrightarrow \exists x.\ \sim P(x)$
and *meson-not-exD*: $!!P.\ \sim(\exists x.\ P(x)) \Longrightarrow \forall x.\ \sim P(x)$
⟨*proof*⟩

Removal of $\longrightarrow$ and $<\!\!\longrightarrow$ (positive and negative occurrences)

lemma *meson-imp-to-disjD*: $P \longrightarrow Q \Longrightarrow \sim P \mid Q$

and *meson-not-impD*: $\sim(P{-\!-}{>}Q) \Longrightarrow P \,\&\, \sim Q$
and *meson-iff-to-disjD*: $P{=}Q \Longrightarrow (\sim P \mid Q) \,\&\, (\sim Q \mid P)$
and *meson-not-iffD*: $\sim(P{=}Q) \Longrightarrow (P \mid Q) \,\&\, (\sim P \mid \sim Q)$
— Much more efficient than $P \wedge \neg\, Q \vee Q \wedge \neg\, P$ for computing CNF
and *meson-not-refl-disj-D*: $x \sim{=}\, x \mid P \Longrightarrow P$
⟨*proof*⟩

38.9.2 Pulling out the existential quantifiers

Conjunction

lemma *meson-conj-exD1*: $!!P\ Q.\ (\exists x.\ P(x)) \,\&\, Q \Longrightarrow \exists x.\ P(x) \,\&\, Q$
and *meson-conj-exD2*: $!!P\ Q.\ P \,\&\, (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P \,\&\, Q(x)$
⟨*proof*⟩

Disjunction

lemma *meson-disj-exD*: $!!P\ Q.\ (\exists x.\ P(x)) \mid (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P(x) \mid Q(x)$
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and *meson-disj-exD1*: $!!P\ Q.\ (\exists x.\ P(x)) \mid Q \Longrightarrow \exists x.\ P(x) \mid Q$
and *meson-disj-exD2*: $!!P\ Q.\ P \mid (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P \mid Q(x)$
⟨*proof*⟩

38.9.3 Generating clauses for the Meson Proof Procedure

Disjunctions

lemma *meson-disj-assoc*: $(P|Q)|R \Longrightarrow P|(Q|R)$
and *meson-disj-comm*: $P|Q \Longrightarrow Q|P$
and *meson-disj-FalseD1*: $False|P \Longrightarrow P$
and *meson-disj-FalseD2*: $P|False \Longrightarrow P$
⟨*proof*⟩

38.10 Lemmas for Meson, the Model Elimination Procedure

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

lemma *make-neg-rule*: $\sim P|Q \Longrightarrow ((\sim P{\Longrightarrow}P) \Longrightarrow Q)$
⟨*proof*⟩

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: $\sim P|Q \Longrightarrow (P \Longrightarrow Q)$
⟨*proof*⟩

P should be a literal

lemma *make-pos-rule*: $P|Q \Longrightarrow ((P{\Longrightarrow}\sim P) \Longrightarrow Q)$

⟨*proof*⟩

Versions of *make-neg-rule* and *make-pos-rule* that don't insert new assumptions, for ordinary resolution.

lemmas *make-neg-rule'* = *make-refined-neg-rule*

lemma *make-pos-rule'*: [|*P*|*Q*; ~*P*|] ==> *Q*
⟨*proof*⟩

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: ~*P* ==> ((~*P*==>*P*) ==> *False*)
⟨*proof*⟩

lemma *make-pos-goal*: *P* ==> ((*P*==>~*P*) ==> *False*)
⟨*proof*⟩

38.10.1 Lemmas for Forward Proof

There is a similarity to congruence rules

lemma *conj-forward*: [| *P'*&*Q'*; *P'* ==> *P*; *Q'* ==> *Q* |] ==> *P*&*Q*
⟨*proof*⟩

lemma *disj-forward*: [| *P'*|*Q'*; *P'* ==> *P*; *Q'* ==> *Q* |] ==> *P*|*Q*
⟨*proof*⟩

lemma *disj-forward2*:
[| *P'*|*Q'*; *P'* ==> *P*; [| *Q'*; *P*==>*False* |] ==> *Q* |] ==> *P*|*Q*
⟨*proof*⟩

lemma *all-forward*: [| ∀*x*. *P'*(*x*); !!*x*. *P'*(*x*) ==> *P*(*x*) |] ==> ∀*x*. *P*(*x*)
⟨*proof*⟩

lemma *ex-forward*: [| ∃*x*. *P'*(*x*); !!*x*. *P'*(*x*) ==> *P*(*x*) |] ==> ∃*x*. *P*(*x*)
⟨*proof*⟩

Many of these bindings are used by the ATP linkup, and not just by legacy proof scripts.

⟨*ML*⟩

38.11 Meson package

⟨*ML*⟩

38.12 Specification package – Hilbertized version

lemma *exE-some*: [| *Ex P* ; *c* == *Eps P* |] ==> *P c*
⟨*proof*⟩

⟨*ML*⟩

end

39 ATP-Linkup: The Isabelle-ATP Linkup

theory *ATP-Linkup*
imports *Record Presburger SAT Recdef Extraction Relation-Power Hilbert-Choice*

uses
Tools/polyhash.ML
Tools/res-clause.ML
(*Tools/res-hol-clause.ML*)
(*Tools/res-axioms.ML*)
(*Tools/res-reconstruct.ML*)
(*Tools/watcher.ML*)
(*Tools/res-atp.ML*)
(*Tools/res-atp-provers.ML*)
(*Tools/res-atp-methods.ML*)
~~/*src/Tools/Metis/metis.ML*
(*Tools/metis-tools.ML*)
begin

definition *COMBI* :: *'a => 'a*
where *COMBI P == P*

definition *COMBK* :: *'a => 'b => 'a*
where *COMBK P Q == P*

definition *COMBB* :: *('b => 'c) => ('a => 'b) => 'a => 'c*
where *COMBB P Q R == P (Q R)*

definition *COMBC* :: *('a => 'b => 'c) => 'b => 'a => 'c*
where *COMBC P Q R == P R Q*

definition *COMBS* :: *('a => 'b => 'c) => ('a => 'b) => 'a => 'c*
where *COMBS P Q R == P R (Q R)*

definition *fequal* :: *'a => 'a => bool*
where *fequal X Y == (X=Y)*

lemma *fequal-imp-equal*: *fequal X Y ==> X=Y*
⟨*proof*⟩

lemma *equal-imp-fequal*: *X=Y ==> fequal X Y*
⟨*proof*⟩

These two represent the equivalence between Boolean equality and iff. They can't be converted to clauses automatically, as the iff would be expanded...

lemma *iff-positive*: $P \mid Q \mid P{=}Q$
⟨*proof*⟩

lemma *iff-negative*: $\sim P \mid \sim Q \mid P{=}Q$
⟨*proof*⟩

Theorems for translation to combinators

lemma *abs-S*: $(\%x.\ (f\ x)\ (g\ x)) == COMBS\ f\ g$
⟨*proof*⟩

lemma *abs-I*: $(\%x.\ x) == COMBI$
⟨*proof*⟩

lemma *abs-K*: $(\%x.\ y) == COMBK\ y$
⟨*proof*⟩

lemma *abs-B*: $(\%x.\ a\ (g\ x)) == COMBB\ a\ g$
⟨*proof*⟩

lemma *abs-C*: $(\%x.\ (f\ x)\ b) == COMBC\ f\ b$
⟨*proof*⟩

⟨*ML*⟩

39.1 Setup for Vampire, E prover and SPASS

⟨*ML*⟩

39.2 The Metis prover

⟨*ML*⟩

end

40 List: The datatype of finite lists

theory *List*
imports *ATP-Linkup*
uses *Tools/string-syntax.ML*
begin

datatype *'a list* =
    Nil ([])
  | *Cons 'a 'a list* (**infixr** # *65*)

40.1 Basic list processing functions

consts
filter:: ('*a* => *bool*) => '*a list* => '*a list*
concat:: '*a list list* => '*a list*
foldl :: ('*b* => '*a* => '*b*) => '*b* => '*a list* => '*b*
foldr :: ('*a* => '*b* => '*b*) => '*a list* => '*b* => '*b*
hd:: '*a list* => '*a*
tl:: '*a list* => '*a list*
last:: '*a list* => '*a*
butlast :: '*a list* => '*a list*
set :: '*a list* => '*a set*
map :: ('*a*=>'*b*) => ('*a list* => '*b list*)
listsum :: '*a list* => '*a*::*monoid-add*
nth :: '*a list* => *nat* => '*a* (**infixl** ! *100*)
list-update :: '*a list* => *nat* => '*a* => '*a list*
take:: *nat* => '*a list* => '*a list*
drop:: *nat* => '*a list* => '*a list*
takeWhile :: ('*a* => *bool*) => '*a list* => '*a list*
dropWhile :: ('*a* => *bool*) => '*a list* => '*a list*
rev :: '*a list* => '*a list*
zip :: '*a list* => '*b list* => ('*a* * '*b*) *list*
upt :: *nat* => *nat* => *nat list* ((*1*[-..</-']))
remdups :: '*a list* => '*a list*
remove1 :: '*a* => '*a list* => '*a list*
distinct:: '*a list* => *bool*
replicate :: *nat* => '*a* => '*a list*
splice :: '*a list* ⇒ '*a list* ⇒ '*a list*

nonterminals *lupdbinds lupdbind*

syntax
— list Enumeration
@*list* :: *args* => '*a list* ([(-)])

— Special syntax for filter
@*filter* :: [*pttrn*, '*a list*, *bool*] => '*a list* ((*1*[-<−-./ -]))

— list update
-*lupdbind*:: ['*a*, '*a*] => *lupdbind* ((*2*- :=/ -))
:: *lupdbind* => *lupdbinds* (-)
-*lupdbinds* :: [*lupdbind*, *lupdbinds*] => *lupdbinds* (-,/ -)
-*LUpdate* :: ['*a*, *lupdbinds*] => '*a* (-/[(-)] [*900*,*0*] *900*)

translations
[*x*, *xs*] == *x*#[*xs*]
[*x*] == *x*#[]
[*x*<−*xs* . *P*]== *filter* (%*x*. *P*) *xs*

-LUpdate xs (-lupdbinds b bs)== -LUpdate (-LUpdate xs b) bs
xs[i:=x] == list-update xs i x

syntax (*xsymbols*)
@filter :: [pttrn, 'a list, bool] => 'a list((1[-←- ./ -]))
syntax (*HTML* **output**)
@filter :: [pttrn, 'a list, bool] => 'a list((1[-←- ./ -]))

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

abbreviation
length :: 'a list => nat **where**
length == size

primrec
hd(x#xs) = x

primrec
tl([]) = []
tl(x#xs) = xs

primrec
last(x#xs) = (if xs=[] then x else last xs)

primrec
butlast []= []
butlast(x#xs) = (if xs=[] then [] else x#butlast xs)

primrec
set [] = {}
set (x#xs) = insert x (set xs)

primrec
map f [] = []
map f (x#xs) = f(x)#map f xs

primrec
append :: 'a list ⇒ 'a list ⇒ 'a list (**infixr** @ *65*)
where
append-Nil:[] @ *ys = ys*
| *append-Cons*: *(x#xs) @ ys = x # xs @ ys*

primrec
rev([]) = []
rev(x#xs) = rev(xs) @ [x]

primrec
filter P [] = []

filter P (x#xs) = (if P x then x#filter P xs else filter P xs)

primrec
foldl-Nil:*foldl f a [] = a*
foldl-Cons: *foldl f a (x#xs) = foldl f (f a x) xs*

primrec
foldr f [] a = a
foldr f (x#xs) a = f x (foldr f xs a)

primrec
concat([]) = []
concat(x#xs) = x @ concat(xs)

primrec
listsum [] = 0
listsum (x # xs) = x + listsum xs

primrec
drop-Nil:*drop n [] = []*
drop-Cons: *drop n (x#xs) = (case n of 0 => x#xs | Suc(m) => drop m xs)*
— Warning: simpset does not contain this definition, but separate theorems for *n = 0* and *n = Suc k*

primrec
take-Nil:*take n [] = []*
take-Cons: *take n (x#xs) = (case n of 0 => [] | Suc(m) => x # take m xs)*
— Warning: simpset does not contain this definition, but separate theorems for *n = 0* and *n = Suc k*

primrec
nth-Cons:*(x#xs)!n = (case n of 0 => x | (Suc k) => xs!k)*
— Warning: simpset does not contain this definition, but separate theorems for *n = 0* and *n = Suc k*

primrec
[][i:=v] = []
(x#xs)[i:=v] = (case i of 0 => v # xs | Suc j => x # xs[j:=v])

primrec
takeWhile P [] = []
takeWhile P (x#xs) = (if P x then x#takeWhile P xs else [])

primrec
dropWhile P [] = []
dropWhile P (x#xs) = (if P x then dropWhile P xs else x#xs)

primrec
zip xs [] = []

zip-Cons: *zip xs* (*y*#*ys*) = (*case xs of* [] => [] | *z*#*zs* => (*z*,*y*)#*zip zs ys*)
— Warning: simpset does not contain this definition, but separate theorems for *xs* = [] and *xs* = *z* # *zs*

primrec
upt-0: [*i*..<*0*] = []
upt-Suc: [*i*..<(*Suc j*)] = (*if i* <= *j then* [*i*..<*j*] @ [*j*] *else* [])

primrec
distinct [] = *True*
distinct (*x*#*xs*) = (*x* ~: *set xs* ∧ *distinct xs*)

primrec
remdups [] = []
remdups (*x*#*xs*) = (*if x* : *set xs then remdups xs else x* # *remdups xs*)

primrec
remove1 x [] = []
remove1 x (*y*#*xs*) = (*if x*=*y then xs else y* # *remove1 x xs*)

primrec
replicate-0: *replicate 0 x* = []
replicate-Suc: *replicate* (*Suc n*) *x* = *x* # *replicate n x*

definition
rotate1 :: *'a list* ⇒ *'a list* **where**
rotate1 xs = (*case xs of* [] ⇒ [] | *x*#*xs* ⇒ *xs* @ [*x*])

definition
rotate :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
rotate n = *rotate1* ^ *n*

definition
list-all2 :: (*'a* => *'b* => *bool*) => *'a list* => *'b list* => *bool* **where**
[*code func del*]: *list-all2 P xs ys* =
(*length xs* = *length ys* ∧ (∀(*x*, *y*) ∈ *set* (*zip xs ys*). *P x y*))

definition
sublist :: *'a list* => *nat set* => *'a list* **where**
sublist xs A = *map fst* (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*0*..<*size xs*]))

primrec
splice [] *ys* = *ys*
splice (*x*#*xs*) *ys* = (*if ys*=[] *then x*#*xs else x* # *hd ys* # *splice xs* (*tl ys*))
— Warning: simpset does not contain the second eqn but a derived one.

Figure 1 shows charachteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient

$[a, b] @ [c, d] = [a, b, c, d]$
length $[a, b, c] = 3$
set $[a, b, c] = \{a, b, c\}$
map f $[a, b, c] = [f\ a, f\ b, f\ c]$
rev $[a, b, c] = [c, b, a]$
hd $[a, b, c, d] = a$
tl $[a, b, c, d] = [b, c, d]$
last $[a, b, c, d] = d$
butlast $[a, b, c, d] = [a, b, c]$
filter $(\lambda n{::}nat.\ n{<}2)\ [0,2,1] = [0,1]$
concat $[[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]$
foldl f x $[a, b, c] = f\ (f\ (f\ x\ a)\ b)\ c$
foldr f $[a, b, c]\ x = f\ a\ (f\ b\ (f\ c\ x))$
zip $[a, b, c]\ [x, y, z] = [(a, x), (b, y), (c, z)]$
zip $[a, b]\ [x, y, z] = [(a, x), (b, y)]$
splice $[a, b, c]\ [x, y, z] = [a, x, b, y, c, z]$
splice $[a, b, c, d]\ [x, y] = [a, x, b, y, c, d]$
take 2 $[a, b, c, d] = [a, b]$
take 6 $[a, b, c, d] = [a, b, c, d]$
drop 2 $[a, b, c, d] = [c, d]$
drop 6 $[a, b, c, d] = []$
takeWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [1, 2]$
dropWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [3, 0]$
distinct $[2, 0, 1]$
remdups $[2, 0, 2, 1, 2] = [0, 1, 2]$
remove1 2 $[2, 0, 2, 1, 2] = [0, 2, 1, 2]$
$[a, b, c, d]\ !\ 2 = c$
$[a, b, c, d][2 := x] = [a, b, x, d]$
sublist $[a, b, c, d, e]\ \{0, 2, 3\} = [a, c, d]$
rotate1 $[a, b, c, d] = [b, c, d, a]$
rotate 3 $[a, b, c, d] = [d, a, b, c]$
replicate 4 a $= [a, a, a, a]$
$[2..{<}5] = [2, 3, 4]$
listsum $[1, 2, 3] = 6$

Figure 1: Characteristic examples

implementations.

context *linorder*
begin

fun *sorted* :: *'a list* ⇒ *bool* **where**
sorted [] ⟷ *True* |
sorted [*x*] ⟷ *True* |
sorted (*x*#*y*#*zs*) ⟷ *x* <= *y* ∧ *sorted* (*y*#*zs*)

primrec *insort* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
insort x [] = [*x*] |
insort x (*y*#*ys*) = (*if x* <= *y then* (*x*#*y*#*ys*) *else y*#(*insort x ys*))

primrec *sort* :: *'a list* ⇒ *'a list* **where**
sort [] = [] |
sort (*x*#*xs*) = *insort x* (*sort xs*)

end

40.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: [(*x*,*y*). *x* ← *xs*, *y* ← *ys*, *x* ≠ *y*], the list of all pairs of distinct elements from *xs* and *ys*. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle's set comprehension): [*e*. *x* ← *xs*, ...] rather than `[e| x <- xs, ...]`.

The qualifiers after the dot are

generators *p* ← *xs*, where *p* is a pattern and *xs* an expression of list type, or

guards *b*, where *b* is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of [*e*. *x* ← *xs*] is optmized to *map* (λ*x*. *e*) *xs*.

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

nonterminals *lc-qual lc-quals*

syntax
-listcompr :: *'a* ⇒ *lc-qual* ⇒ *lc-quals* ⇒ *'a list* ([- . --)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- <− -)

-lc-test :: *bool* ⇒ *lc-qual* (-)

-lc-end :: *lc-quals* (])
-lc-quals :: *lc-qual* ⇒ *lc-quals* ⇒ *lc-quals* (, --)
-lc-abs :: *'a* => *'b list* => *'b list*

syntax (*xsymbols*)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)
syntax (*HTML* **output**)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)

⟨*ML*⟩

40.1.2 [] and *op #*

lemma *not-Cons-self* [*simp*]:
xs ≠ *x* # *xs*
⟨*proof*⟩

lemmas *not-Cons-self2* [*simp*] = *not-Cons-self* [*symmetric*]

lemma *neq-Nil-conv*: (*xs* ≠ []) = (∃ *y ys*. *xs* = *y* # *ys*)
⟨*proof*⟩

lemma *length-induct*:
(⋀*xs*. ∀ *ys*. *length ys* < *length xs* ⟶ *P ys* ⟹ *P xs*) ⟹ *P xs*
⟨*proof*⟩

40.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: *length* (*xs* @ *ys*) = *length xs* + *length ys*
⟨*proof*⟩

lemma *length-map* [*simp*]: *length* (*map f xs*) = *length xs*
⟨*proof*⟩

lemma *length-rev* [*simp*]: *length* (*rev xs*) = *length xs*
⟨*proof*⟩

lemma *length-tl* [*simp*]: *length* (*tl xs*) = *length xs* − *1*
⟨*proof*⟩

lemma *length-0-conv* [*iff*]: (*length xs* = *0*) = (*xs* = [])
⟨*proof*⟩

lemma *length-greater-0-conv* [*iff*]: (*0* < *length xs*) = (*xs* ≠ [])

⟨*proof*⟩

lemma *length-pos-if-in-set*: $x : set\ xs \Longrightarrow length\ xs > 0$
⟨*proof*⟩

lemma *length-Suc-conv*:
$(length\ xs = Suc\ n) = (\exists y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
⟨*proof*⟩

lemma *Suc-length-conv*:
$(Suc\ n = length\ xs) = (\exists y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
⟨*proof*⟩

lemma *impossible-Cons*: $length\ xs <= length\ ys ==> xs = x\ \#\ ys = False$
⟨*proof*⟩

lemma *list-induct2* [*consumes 1*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow P\ []\ [] \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (x\#xs)\ (y\#ys))$
$\Longrightarrow P\ xs\ ys$
⟨*proof*⟩

lemma *list-induct3* [*consumes 2*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow P\ []\ []\ [] \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys\ z\ zs.\ length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow P\ xs\ ys\ zs$
$\Longrightarrow P\ (x\#xs)\ (y\#ys)\ (z\#zs))$
$\Longrightarrow P\ xs\ ys\ zs$
⟨*proof*⟩

lemma *list-induct2'*:
$[\![\ P\ []\ [];$
$\bigwedge x\ xs.\ P\ (x\#xs)\ [];$
$\bigwedge y\ ys.\ P\ []\ (y\#ys);$
$\bigwedge x\ xs\ y\ ys.\ P\ xs\ ys \Longrightarrow P\ (x\#xs)\ (y\#ys)\ ]\!]$
$\Longrightarrow P\ xs\ ys$
⟨*proof*⟩

lemma *neq-if-length-neq*: $length\ xs \neq length\ ys \Longrightarrow (xs = ys) == False$
⟨*proof*⟩

⟨*ML*⟩

40.1.4 @ – append

lemma *append-assoc* [*simp*]: $(xs\ @\ ys)\ @\ zs = xs\ @\ (ys\ @\ zs)$
⟨*proof*⟩

lemma *append-Nil2* [*simp*]: $xs\ @\ [] = xs$
⟨*proof*⟩

interpretation *semigroup-append*: *semigroup-add* [*op* @]
⟨*proof*⟩
interpretation *monoid-append*: *monoid-add* [[] *op* @]
⟨*proof*⟩

lemma *append-is-Nil-conv* [*iff*]: $(xs \mathbin{@} ys = []) = (xs = [] \wedge ys = [])$
⟨*proof*⟩

lemma *Nil-is-append-conv* [*iff*]: $([] = xs \mathbin{@} ys) = (xs = [] \wedge ys = [])$
⟨*proof*⟩

lemma *append-self-conv* [*iff*]: $(xs \mathbin{@} ys = xs) = (ys = [])$
⟨*proof*⟩

lemma *self-append-conv* [*iff*]: $(xs = xs \mathbin{@} ys) = (ys = [])$
⟨*proof*⟩

lemma *append-eq-append-conv* [*simp*, *noatp*]:
$length\ xs = length\ ys \vee length\ us = length\ vs$
$\Longrightarrow (xs@us = ys@vs) = (xs{=}ys \wedge us{=}vs)$
⟨*proof*⟩

lemma *append-eq-append-conv2*: $(xs \mathbin{@} ys = zs \mathbin{@} ts) =$
$(EX\ us.\ xs = zs \mathbin{@} us\ \&\ us \mathbin{@} ys = ts \mid xs \mathbin{@} us = zs\ \&\ ys = us@\ ts)$
⟨*proof*⟩

lemma *same-append-eq* [*iff*]: $(xs \mathbin{@} ys = xs \mathbin{@} zs) = (ys = zs)$
⟨*proof*⟩

lemma *append1-eq-conv* [*iff*]: $(xs \mathbin{@} [x] = ys \mathbin{@} [y]) = (xs = ys \wedge x = y)$
⟨*proof*⟩

lemma *append-same-eq* [*iff*]: $(ys \mathbin{@} xs = zs \mathbin{@} xs) = (ys = zs)$
⟨*proof*⟩

lemma *append-self-conv2* [*iff*]: $(xs \mathbin{@} ys = ys) = (xs = [])$
⟨*proof*⟩

lemma *self-append-conv2* [*iff*]: $(ys = xs \mathbin{@} ys) = (xs = [])$
⟨*proof*⟩

lemma *hd-Cons-tl* [*simp*,*noatp*]: $xs \neq [] \Longrightarrow hd\ xs \mathbin{\#} tl\ xs = xs$
⟨*proof*⟩

lemma *hd-append*: $hd\ (xs \mathbin{@} ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
⟨*proof*⟩

lemma *hd-append2* [*simp*]: $xs \neq [] \Longrightarrow hd\ (xs \mathbin{@} ys) = hd\ xs$

⟨*proof*⟩

lemma *tl-append*: *tl* (*xs* @ *ys*) = (*case xs of* [] => *tl ys* | *z*#*zs* => *zs* @ *ys*)
⟨*proof*⟩

lemma *tl-append2* [*simp*]: *xs* ≠ [] ==> *tl* (*xs* @ *ys*) = *tl xs* @ *ys*
⟨*proof*⟩

lemma *Cons-eq-append-conv*: *x*#*xs* = *ys*@*zs* =
(*ys* = [] & *x*#*xs* = *zs* | (*EX ys'*. *x*#*ys'* = *ys* & *xs* = *ys'*@*zs*))
⟨*proof*⟩

lemma *append-eq-Cons-conv*: (*ys*@*zs* = *x*#*xs*) =
(*ys* = [] & *zs* = *x*#*xs* | (*EX ys'*. *ys* = *x*#*ys'* & *ys'*@*zs* = *xs*))
⟨*proof*⟩

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: *xs* = *ys* ==> *xs* = [] @ *ys*
⟨*proof*⟩

lemma *Cons-eq-appendI*:
[| *x* # *xs1* = *ys*; *xs* = *xs1* @ *zs* |] ==> *x* # *xs* = *ys* @ *zs*
⟨*proof*⟩

lemma *append-eq-appendI*:
[| *xs* @ *xs1* = *zs*; *ys* = *xs1* @ *us* |] ==> *xs* @ *ys* = *zs* @ *us*
⟨*proof*⟩

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

⟨*ML*⟩

40.1.5 *map*

lemma *map-ext*: (!!*x*. *x* : *set xs* --> *f x* = *g x*) ==> *map f xs* = *map g xs*
⟨*proof*⟩

lemma *map-ident* [*simp*]: *map* (λ*x*. *x*) = (λ*xs*. *xs*)
⟨*proof*⟩

lemma *map-append* [*simp*]: *map f* (*xs* @ *ys*) = *map f xs* @ *map f ys*
⟨*proof*⟩

lemma *map-compose*: *map* (*f o g*) *xs* = *map f* (*map g xs*)
⟨*proof*⟩

lemma *rev-map*: *rev* (*map f xs*) = *map f* (*rev xs*)

⟨*proof*⟩

lemma *map-eq-conv*[*simp*]: (*map f xs* = *map g xs*) = (!*x* : *set xs*. *f x* = *g x*)
⟨*proof*⟩

lemma *map-cong* [*fundef-cong*, *recdef-cong*]:
xs = *ys* ==> (!!*x*. *x* : *set ys* ==> *f x* = *g x*) ==> *map f xs* = *map g ys*
— a congruence rule for *map*
⟨*proof*⟩

lemma *map-is-Nil-conv* [*iff*]: (*map f xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *Nil-is-map-conv* [*iff*]: ([] = *map f xs*) = (*xs* = [])
⟨*proof*⟩

lemma *map-eq-Cons-conv*:
(*map f xs* = *y*#*ys*) = (∃ *z zs*. *xs* = *z*#*zs* ∧ *f z* = *y* ∧ *map f zs* = *ys*)
⟨*proof*⟩

lemma *Cons-eq-map-conv*:
(*x*#*xs* = *map f ys*) = (∃ *z zs*. *ys* = *z*#*zs* ∧ *x* = *f z* ∧ *xs* = *map f zs*)
⟨*proof*⟩

lemmas *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]
lemmas *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]
declare *map-eq-Cons-D* [*dest*!] *Cons-eq-map-D* [*dest*!]

lemma *ex-map-conv*:
(*EX xs*. *ys* = *map f xs*) = (*ALL y* : *set ys*. *EX x*. *y* = *f x*)
⟨*proof*⟩

lemma *map-eq-imp-length-eq*:
assumes *map f xs* = *map f ys*
shows *length xs* = *length ys*
⟨*proof*⟩

lemma *map-inj-on*:
[| *map f xs* = *map f ys*; *inj-on f* (*set xs Un set ys*) |]
==> *xs* = *ys*
⟨*proof*⟩

lemma *inj-on-map-eq-map*:
inj-on f (*set xs Un set ys*) ⟹ (*map f xs* = *map f ys*) = (*xs* = *ys*)
⟨*proof*⟩

lemma *map-injective*:
map f xs = *map f ys* ==> *inj f* ==> *xs* = *ys*
⟨*proof*⟩

lemma *inj-map-eq-map*[*simp*]: *inj f* $\Longrightarrow$ (*map f xs* = *map f ys*) = (*xs* = *ys*)
⟨*proof*⟩

lemma *inj-mapI*: *inj f* ==> *inj* (*map f*)
⟨*proof*⟩

lemma *inj-mapD*: *inj* (*map f*) ==> *inj f*
⟨*proof*⟩

lemma *inj-map*[*iff*]: *inj* (*map f*) = *inj f*
⟨*proof*⟩

lemma *inj-on-mapI*: *inj-on f* (⋃(*set* ' *A*)) $\Longrightarrow$ *inj-on* (*map f*) *A*
⟨*proof*⟩

lemma *map-idI*: (⋀*x*. *x* ∈ *set xs* $\Longrightarrow$ *f x* = *x*) $\Longrightarrow$ *map f xs* = *xs*
⟨*proof*⟩

lemma *map-fun-upd* [*simp*]: *y* ∉ *set xs* $\Longrightarrow$ *map* (*f*(*y*:=*v*)) *xs* = *map f xs*
⟨*proof*⟩

lemma *map-fst-zip*[*simp*]:
length xs = *length ys* $\Longrightarrow$ *map fst* (*zip xs ys*) = *xs*
⟨*proof*⟩

lemma *map-snd-zip*[*simp*]:
length xs = *length ys* $\Longrightarrow$ *map snd* (*zip xs ys*) = *ys*
⟨*proof*⟩

40.1.6 *rev*

lemma *rev-append* [*simp*]: *rev* (*xs* @ *ys*) = *rev ys* @ *rev xs*
⟨*proof*⟩

lemma *rev-rev-ident* [*simp*]: *rev* (*rev xs*) = *xs*
⟨*proof*⟩

lemma *rev-swap*: (*rev xs* = *ys*) = (*xs* = *rev ys*)
⟨*proof*⟩

lemma *rev-is-Nil-conv* [*iff*]: (*rev xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *Nil-is-rev-conv* [*iff*]: ([] = *rev xs*) = (*xs* = [])
⟨*proof*⟩

lemma *rev-singleton-conv* [*simp*]: (*rev xs* = [*x*]) = (*xs* = [*x*])
⟨*proof*⟩

lemma *singleton-rev-conv* [*simp*]: $([x] = rev\ xs) = (xs = [x])$
⟨*proof*⟩

lemma *rev-is-rev-conv* [*iff*]: $(rev\ xs = rev\ ys) = (xs = ys)$
⟨*proof*⟩

lemma *inj-on-rev*[*iff*]: *inj-on rev A*
⟨*proof*⟩

lemma *rev-induct* [*case-names Nil snoc*]:
$[|\ P\ []; !!x\ xs.\ P\ xs ==> P\ (xs\ @\ [x])\ |] ==> P\ xs$
⟨*proof*⟩

lemma *rev-exhaust* [*case-names Nil snoc*]:
$(xs = [] ==> P) ==>(!!ys\ y.\ xs = ys\ @\ [y] ==> P) ==> P$
⟨*proof*⟩

lemmas *rev-cases = rev-exhaust*

lemma *rev-eq-Cons-iff*[*iff*]: $(rev\ xs = y\#ys) = (xs = rev\ ys\ @\ [y])$
⟨*proof*⟩

40.1.7 *set*

lemma *finite-set* [*iff*]: *finite* $(set\ xs)$
⟨*proof*⟩

lemma *set-append* [*simp*]: $set\ (xs\ @\ ys) = (set\ xs \cup set\ ys)$
⟨*proof*⟩

lemma *hd-in-set*[*simp*]: $xs \neq [] \Longrightarrow hd\ xs : set\ xs$
⟨*proof*⟩

lemma *set-subset-Cons*: $set\ xs \subseteq set\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *set-ConsD*: $y \in set\ (x\ \#\ xs) \Longrightarrow y{=}x \vee y \in set\ xs$
⟨*proof*⟩

lemma *set-empty* [*iff*]: $(set\ xs = \{\}) = (xs = [])$
⟨*proof*⟩

lemma *set-empty2*[*iff*]: $(\{\} = set\ xs) = (xs = [])$
⟨*proof*⟩

lemma *set-rev* [*simp*]: $set\ (rev\ xs) = set\ xs$
⟨*proof*⟩

lemma *set-map* [*simp*]: *set* (*map f xs*) = *f*‘(*set xs*)
⟨*proof*⟩

lemma *set-filter* [*simp*]: *set* (*filter P xs*) = {*x*. *x* : *set xs* ∧ *P x*}
⟨*proof*⟩

lemma *set-upt* [*simp*]: *set*[*i*..<*j*] = {*k*. *i* ≤ *k* ∧ *k* < *j*}
⟨*proof*⟩

lemma *split-list*: *x* : *set xs* ⟹ ∃ *ys zs*. *xs* = *ys* @ *x* # *zs*
⟨*proof*⟩

lemma *in-set-conv-decomp*: *x* ∈ *set xs* ⟷ (∃ *ys zs*. *xs* = *ys* @ *x* # *zs*)
⟨*proof*⟩

lemma *split-list-first*: *x* : *set xs* ⟹ ∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set ys*
⟨*proof*⟩

lemma *in-set-conv-decomp-first*:
(*x* : *set xs*) = (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set ys*)
⟨*proof*⟩

lemma *split-list-last*: *x* : *set xs* ⟹ ∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set zs*
⟨*proof*⟩

lemma *in-set-conv-decomp-last*:
(*x* : *set xs*) = (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set zs*)
⟨*proof*⟩

lemma *split-list-prop*: ∃ *x* ∈ *set xs*. *P x* ⟹ ∃ *ys x zs*. *xs* = *ys* @ *x* # *zs* & *P x*
⟨*proof*⟩

lemma *split-list-propE*:
assumes ∃ *x* ∈ *set xs*. *P x*
obtains *ys x zs* **where** *xs* = *ys* @ *x* # *zs* **and** *P x*
⟨*proof*⟩

lemma *split-list-first-prop*:
∃ *x* ∈ *set xs*. *P x* ⟹
∃ *ys x zs*. *xs* = *ys*@*x*#*zs* ∧ *P x* ∧ (∀ *y* ∈ *set ys*. ¬ *P y*)
⟨*proof*⟩

lemma *split-list-first-propE*:
assumes ∃ *x* ∈ *set xs*. *P x*
obtains *ys x zs* **where** *xs* = *ys* @ *x* # *zs* **and** *P x* **and** ∀ *y* ∈ *set ys*. ¬ *P y*
⟨*proof*⟩

lemma *split-list-first-prop-iff*:

$(\exists\, x \in set\ xs.\ P\ x) \longleftrightarrow$
$(\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, y \in set\ ys.\ \neg\ P\ y))$
⟨*proof*⟩

lemma *split-list-last-prop*:
$\exists\, x \in set\ xs.\ P\ x \Longrightarrow$
$\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, z \in set\ zs.\ \neg\ P\ z)$
⟨*proof*⟩

lemma *split-list-last-propE*:
assumes $\exists\, x \in set\ xs.\ P\ x$
obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $P\ x$ **and** $\forall\, z \in set\ zs.\ \neg\ P\ z$
⟨*proof*⟩

lemma *split-list-last-prop-iff*:
$(\exists\, x \in set\ xs.\ P\ x) \longleftrightarrow$
$(\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, z \in set\ zs.\ \neg\ P\ z))$
⟨*proof*⟩

lemma *finite-list*: *finite A ==> EX xs. set xs = A*
⟨*proof*⟩

lemma *card-length*: $card\ (set\ xs) \leq length\ xs$
⟨*proof*⟩

lemma *set-minus-filter-out*:
$set\ xs - \{y\} = set\ (filter\ (\lambda x.\ \neg\ (x = y))\ xs)$
⟨*proof*⟩

40.1.8 *filter*

lemma *filter-append* [*simp*]: $filter\ P\ (xs\ @\ ys) = filter\ P\ xs\ @\ filter\ P\ ys$
⟨*proof*⟩

lemma *rev-filter*: $rev\ (filter\ P\ xs) = filter\ P\ (rev\ xs)$
⟨*proof*⟩

lemma *filter-filter* [*simp*]: $filter\ P\ (filter\ Q\ xs) = filter\ (\lambda x.\ Q\ x \wedge P\ x)\ xs$
⟨*proof*⟩

lemma *length-filter-le* [*simp*]: $length\ (filter\ P\ xs) \leq length\ xs$
⟨*proof*⟩

lemma *sum-length-filter-compl*:
$length(filter\ P\ xs) + length(filter\ (\%x.\ {}^\sim P\ x)\ xs) = length\ xs$
⟨*proof*⟩

lemma *filter-True* [*simp*]: $\forall\, x \in set\ xs.\ P\ x$ *==> filter P xs = xs*
⟨*proof*⟩

lemma *filter-False* [*simp*]: $\forall x \in set\ xs.\ \neg\ P\ x \Longrightarrow filter\ P\ xs = []$
⟨*proof*⟩

lemma *filter-empty-conv*: $(filter\ P\ xs = []) = (\forall x{\in}set\ xs.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *filter-id-conv*: $(filter\ P\ xs = xs) = (\forall x{\in}set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *filter-map*:
$filter\ P\ (map\ f\ xs) = map\ f\ (filter\ (P\ o\ f)\ xs)$
⟨*proof*⟩

lemma *length-filter-map*[*simp*]:
$length\ (filter\ P\ (map\ f\ xs)) = length(filter\ (P\ o\ f)\ xs)$
⟨*proof*⟩

lemma *filter-is-subset* [*simp*]: $set\ (filter\ P\ xs) \leq set\ xs$
⟨*proof*⟩

lemma *length-filter-less*:
$[\![\ x : set\ xs; \sim\ P\ x\ ]\!] \Longrightarrow length(filter\ P\ xs) < length\ xs$
⟨*proof*⟩

lemma *length-filter-conv-card*:
$length(filter\ p\ xs) = card\{i.\ i < length\ xs\ \&\ p(xs!i)\}$
⟨*proof*⟩

lemma *Cons-eq-filterD*:
$x\#xs = filter\ P\ ys \Longrightarrow$
$\exists us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
(**is** - $\Longrightarrow \exists us\ vs.\ ?P\ ys\ us\ vs$)
⟨*proof*⟩

lemma *filter-eq-ConsD*:
$filter\ P\ ys = x\#xs \Longrightarrow$
$\exists us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
⟨*proof*⟩

lemma *filter-eq-Cons-iff*:
$(filter\ P\ ys = x\#xs) =$
$(\exists us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
⟨*proof*⟩

lemma *Cons-eq-filter-iff*:
$(x\#xs = filter\ P\ ys) =$
$(\exists us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
⟨*proof*⟩

lemma *filter-cong*[*fundef-cong*, *recdef-cong*]:
$xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ ys \Longrightarrow P\ x = Q\ x) \Longrightarrow filter\ P\ xs = filter\ Q\ ys$
⟨*proof*⟩

40.1.9 List partitioning

primrec *partition* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \times 'a\ list$ **where**
  partition P [] = ([], [])
  | *partition P* (*x* # *xs*) =
      (*let* (*yes*, *no*) = *partition P xs*
      in if P x then (*x* # *yes*, *no*) *else* (*yes*, *x* # *no*))

lemma *partition-filter1*:
    fst (*partition P xs*) = *filter P xs*
⟨*proof*⟩

lemma *partition-filter2*:
    snd (*partition P xs*) = *filter* (*Not o P*) *xs*
⟨*proof*⟩

lemma *partition-P*:
  assumes *partition P xs* = (*yes*, *no*)
  shows $(\forall p \in set\ yes.\ P\ p) \wedge (\forall p \in set\ no.\ \neg\ P\ p)$
⟨*proof*⟩

lemma *partition-set*:
  assumes *partition P xs* = (*yes*, *no*)
  shows *set yes* ∪ *set no* = *set xs*
⟨*proof*⟩

40.1.10 *concat*

lemma *concat-append* [*simp*]: *concat* (*xs* @ *ys*) = *concat xs* @ *concat ys*
⟨*proof*⟩

lemma *concat-eq-Nil-conv* [*simp*]: (*concat xss* = []) = $(\forall xs \in set\ xss.\ xs = [])$
⟨*proof*⟩

lemma *Nil-eq-concat-conv* [*simp*]: ([] = *concat xss*) = $(\forall xs \in set\ xss.\ xs = [])$
⟨*proof*⟩

lemma *set-concat* [*simp*]: *set* (*concat xs*) = (*UN x*:*set xs*. *set x*)
⟨*proof*⟩

lemma *concat-map-singleton*[*simp*]: *concat*(*map* (%*x*. [*f x*]) *xs*) = *map f xs*
⟨*proof*⟩

lemma *map-concat*: *map f* (*concat xs*) = *concat* (*map* (*map f*) *xs*)
⟨*proof*⟩

lemma *filter-concat*: *filter p (concat xs) = concat (map (filter p) xs)*
⟨*proof*⟩

lemma *rev-concat*: *rev (concat xs) = concat (map rev (rev xs))*
⟨*proof*⟩

40.1.11 *nth*

lemma *nth-Cons-0* [*simp*]: *(x # xs)!0 = x*
⟨*proof*⟩

lemma *nth-Cons-Suc* [*simp*]: *(x # xs)!(Suc n) = xs!n*
⟨*proof*⟩

declare *nth.simps* [*simp del*]

lemma *nth-append*:
(xs @ ys)!n = (if n < length xs then xs!n else ys!(n − length xs))
⟨*proof*⟩

lemma *nth-append-length* [*simp*]: *(xs @ x # ys) ! length xs = x*
⟨*proof*⟩

lemma *nth-append-length-plus*[*simp*]: *(xs @ ys) ! (length xs + n) = ys ! n*
⟨*proof*⟩

lemma *nth-map* [*simp*]: *n < length xs ==> (map f xs)!n = f(xs!n)*
⟨*proof*⟩

lemma *hd-conv-nth*: *xs ≠ [] ⟹ hd xs = xs!0*
⟨*proof*⟩

lemma *list-eq-iff-nth-eq*:
(xs = ys) = (length xs = length ys ∧ (ALL i<length xs. xs!i = ys!i))
⟨*proof*⟩

lemma *set-conv-nth*: *set xs = {xs!i | i. i < length xs}*
⟨*proof*⟩

lemma *in-set-conv-nth*: *(x ∈ set xs) = (∃ i < length xs. xs!i = x)*
⟨*proof*⟩

lemma *list-ball-nth*: *[| n < length xs; !x : set xs. P x|] ==> P(xs!n)*
⟨*proof*⟩

lemma *nth-mem* [*simp*]: *n < length xs ==> xs!n : set xs*
⟨*proof*⟩

lemma *all-nth-imp-all-set*:
[| !i < *length xs*. P(*xs*!i); x : *set xs*|] ==> P x
⟨*proof*⟩

lemma *all-set-conv-all-nth*:
(∀ x ∈ *set xs*. P x) = (∀ i. i < *length xs* −−> P (*xs* ! i))
⟨*proof*⟩

lemma *rev-nth*:
n < *size xs* ⟹ *rev xs* ! n = *xs* ! (*length xs* − *Suc n*)
⟨*proof*⟩

40.1.12 *list-update*

lemma *length-list-update* [*simp*]: *length*(*xs*[i:=x]) = *length xs*
⟨*proof*⟩

lemma *nth-list-update*:
i < *length xs*==> (*xs*[i:=x])!j = (*if* i = j *then* x *else xs*!j)
⟨*proof*⟩

lemma *nth-list-update-eq* [*simp*]: i < *length xs* ==> (*xs*[i:=x])!i = x
⟨*proof*⟩

lemma *nth-list-update-neq* [*simp*]: i ≠ j ==> *xs*[i:=x]!j = *xs*!j
⟨*proof*⟩

lemma *list-update-id*[*simp*]: *xs*[i := *xs*!i] = *xs*
⟨*proof*⟩

lemma *list-update-beyond*[*simp*]: *length xs* ≤ i ⟹ *xs*[i:=x] = *xs*
⟨*proof*⟩

lemma *list-update-same-conv*:
i < *length xs* ==> (*xs*[i := x] = *xs*) = (*xs*!i = x)
⟨*proof*⟩

lemma *list-update-append1*:
i < *size xs* ⟹ (*xs* @ *ys*)[i:=x] = *xs*[i:=x] @ *ys*
⟨*proof*⟩

lemma *list-update-append*:
(*xs* @ *ys*) [n:= x] =
(*if* n < *length xs then xs*[n:= x] @ *ys else xs* @ (*ys* [n−*length xs*:= x]))
⟨*proof*⟩

lemma *list-update-length* [*simp*]:
(*xs* @ x # *ys*)[*length xs* := y] = (*xs* @ y # *ys*)

⟨*proof*⟩

lemma *update-zip*:
length xs = length ys ==>
(zip xs ys)[i:=xy] = zip (xs[i:=fst xy]) (ys[i:=snd xy])
⟨*proof*⟩

lemma *set-update-subset-insert*: *set(xs[i:=x]) <= insert x (set xs)*
⟨*proof*⟩

lemma *set-update-subsetI*: *[| set xs <= A; x:A |] ==> set(xs[i := x]) <= A*
⟨*proof*⟩

lemma *set-update-memI*: *n < length xs ⟹ x ∈ set (xs[n := x])*
⟨*proof*⟩

lemma *list-update-overwrite*:
xs [i := x, i := y] = xs [i := y]
⟨*proof*⟩

lemma *list-update-swap*:
i ≠ i' ⟹ xs [i := x, i' := x'] = xs [i' := x', i := x]
⟨*proof*⟩

40.1.13 *last* and *butlast*

lemma *last-snoc* [*simp*]: *last (xs @ [x]) = x*
⟨*proof*⟩

lemma *butlast-snoc* [*simp*]: *butlast (xs @ [x]) = xs*
⟨*proof*⟩

lemma *last-ConsL*: *xs = [] ⟹ last(x#xs) = x*
⟨*proof*⟩

lemma *last-ConsR*: *xs ≠ [] ⟹ last(x#xs) = last xs*
⟨*proof*⟩

lemma *last-append*: *last(xs @ ys) = (if ys = [] then last xs else last ys)*
⟨*proof*⟩

lemma *last-appendL*[*simp*]: *ys = [] ⟹ last(xs @ ys) = last xs*
⟨*proof*⟩

lemma *last-appendR*[*simp*]: *ys ≠ [] ⟹ last(xs @ ys) = last ys*
⟨*proof*⟩

lemma *hd-rev*: *xs ≠ [] ⟹ hd(rev xs) = last xs*
⟨*proof*⟩

lemma *last-rev*: $xs \neq [] \Longrightarrow last(rev\ xs) = hd\ xs$
⟨*proof*⟩

lemma *last-in-set*[*simp*]: $as \neq [] \Longrightarrow last\ as \in set\ as$
⟨*proof*⟩

lemma *length-butlast* [*simp*]: $length\ (butlast\ xs) = length\ xs - 1$
⟨*proof*⟩

lemma *butlast-append*:
$butlast\ (xs\ @\ ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs\ @\ butlast\ ys)$
⟨*proof*⟩

lemma *append-butlast-last-id* [*simp*]:
$xs \neq [] ==> butlast\ xs\ @\ [last\ xs] = xs$
⟨*proof*⟩

lemma *in-set-butlastD*: $x : set\ (butlast\ xs) ==> x : set\ xs$
⟨*proof*⟩

lemma *in-set-butlast-appendI*:
$x : set\ (butlast\ xs) \mid x : set\ (butlast\ ys) ==> x : set\ (butlast\ (xs\ @\ ys))$
⟨*proof*⟩

lemma *last-drop*[*simp*]: $n < length\ xs \Longrightarrow last\ (drop\ n\ xs) = last\ xs$
⟨*proof*⟩

lemma *last-conv-nth*: $xs \neq [] \Longrightarrow last\ xs = xs!(length\ xs - 1)$
⟨*proof*⟩

lemma *butlast-conv-take*: $butlast\ xs = take\ (length\ xs - 1)\ xs$
⟨*proof*⟩

40.1.14 *take* and *drop*

lemma *take-0* [*simp*]: $take\ 0\ xs = []$
⟨*proof*⟩

lemma *drop-0* [*simp*]: $drop\ 0\ xs = xs$
⟨*proof*⟩

lemma *take-Suc-Cons* [*simp*]: $take\ (Suc\ n)\ (x\ \#\ xs) = x\ \#\ take\ n\ xs$
⟨*proof*⟩

lemma *drop-Suc-Cons* [*simp*]: $drop\ (Suc\ n)\ (x\ \#\ xs) = drop\ n\ xs$
⟨*proof*⟩

declare *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

lemma *take-Suc*: *xs ~= [] ==> take (Suc n) xs = hd xs # take n (tl xs)*
⟨*proof*⟩

lemma *drop-Suc*: *drop (Suc n) xs = drop n (tl xs)*
⟨*proof*⟩

lemma *take-tl*: *take n (tl xs) = tl (take (Suc n) xs)*
⟨*proof*⟩

lemma *drop-tl*: *drop n (tl xs) = tl(drop n xs)*
⟨*proof*⟩

lemma *tl-take*: *tl (take n xs) = take (n - 1) (tl xs)*
⟨*proof*⟩

lemma *tl-drop*: *tl (drop n xs) = drop n (tl xs)*
⟨*proof*⟩

lemma *nth-via-drop*: *drop n xs = y#ys ⟹ xs!n = y*
⟨*proof*⟩

lemma *take-Suc-conv-app-nth*:
i < length xs ⟹ take (Suc i) xs = take i xs @ [xs!i]
⟨*proof*⟩

lemma *drop-Suc-conv-tl*:
i < length xs ⟹ (xs!i) # (drop (Suc i) xs) = drop i xs
⟨*proof*⟩

lemma *length-take* [*simp*]: *length (take n xs) = min (length xs) n*
⟨*proof*⟩

lemma *length-drop* [*simp*]: *length (drop n xs) = (length xs - n)*
⟨*proof*⟩

lemma *take-all* [*simp*]: *length xs <= n ==> take n xs = xs*
⟨*proof*⟩

lemma *drop-all* [*simp*]: *length xs <= n ==> drop n xs = []*
⟨*proof*⟩

lemma *take-append* [*simp*]:
take n (xs @ ys) = (take n xs @ take (n - length xs) ys)
⟨*proof*⟩

lemma *drop-append* [*simp*]:
drop n (xs @ ys) = drop n xs @ drop (n - length xs) ys
⟨*proof*⟩

lemma *take-take* [*simp*]: *take n* (*take m xs*) = *take* (*min n m*) *xs*
⟨*proof*⟩

lemma *drop-drop* [*simp*]: *drop n* (*drop m xs*) = *drop* (*n* + *m*) *xs*
⟨*proof*⟩

lemma *take-drop*: *take n* (*drop m xs*) = *drop m* (*take* (*n* + *m*) *xs*)
⟨*proof*⟩

lemma *drop-take*: *drop n* (*take m xs*) = *take* (*m*−*n*) (*drop n xs*)
⟨*proof*⟩

lemma *append-take-drop-id* [*simp*]: *take n xs* @ *drop n xs* = *xs*
⟨*proof*⟩

lemma *take-eq-Nil*[*simp*]: (*take n xs* = []) = (*n* = *0* ∨ *xs* = [])
⟨*proof*⟩

lemma *drop-eq-Nil*[*simp*]: (*drop n xs* = []) = (*length xs* <= *n*)
⟨*proof*⟩

lemma *take-map*: *take n* (*map f xs*) = *map f* (*take n xs*)
⟨*proof*⟩

lemma *drop-map*: *drop n* (*map f xs*) = *map f* (*drop n xs*)
⟨*proof*⟩

lemma *rev-take*: *rev* (*take i xs*) = *drop* (*length xs* − *i*) (*rev xs*)
⟨*proof*⟩

lemma *rev-drop*: *rev* (*drop i xs*) = *take* (*length xs* − *i*) (*rev xs*)
⟨*proof*⟩

lemma *nth-take* [*simp*]: *i* < *n* ==> (*take n xs*)!*i* = *xs*!*i*
⟨*proof*⟩

lemma *nth-drop* [*simp*]:
n + *i* <= *length xs* ==> (*drop n xs*)!*i* = *xs*!(*n* + *i*)
⟨*proof*⟩

lemma *butlast-take*:
n <= *length xs* ==> *butlast* (*take n xs*) = *take* (*n* − *1*) *xs*
⟨*proof*⟩

lemma *butlast-drop*: *butlast* (*drop n xs*) = *drop n* (*butlast xs*)
⟨*proof*⟩

lemma *take-butlast*: *n* < *length xs* ==> *take n* (*butlast xs*) = *take n xs*

⟨*proof*⟩

lemma *drop-butlast*: *drop n* (*butlast xs*) = *butlast* (*drop n xs*)
⟨*proof*⟩

lemma *hd-drop-conv-nth*: ⟦ *xs* ≠ []; *n* < *length xs* ⟧ ⟹ *hd*(*drop n xs*) = *xs*!*n*
⟨*proof*⟩

lemma *set-take-subset*: *set*(*take n xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *set-drop-subset*: *set*(*drop n xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *in-set-takeD*: *x* : *set*(*take n xs*) ⟹ *x* : *set xs*
⟨*proof*⟩

lemma *in-set-dropD*: *x* : *set*(*drop n xs*) ⟹ *x* : *set xs*
⟨*proof*⟩

lemma *append-eq-conv-conj*:
(*xs* @ *ys* = *zs*) = (*xs* = *take* (*length xs*) *zs* ∧ *ys* = *drop* (*length xs*) *zs*)
⟨*proof*⟩

lemma *take-add*:
i+*j* ≤ *length*(*xs*) ⟹ *take* (*i*+*j*) *xs* = *take i xs* @ *take j* (*drop i xs*)
⟨*proof*⟩

lemma *append-eq-append-conv-if*:
(*xs*$_1$ @ *xs*$_2$ = *ys*$_1$ @ *ys*$_2$) =
(*if size xs*$_1$ ≤ *size ys*$_1$
then xs$_1$ = *take* (*size xs*$_1$) *ys*$_1$ ∧ *xs*$_2$ = *drop* (*size xs*$_1$) *ys*$_1$ @ *ys*$_2$
else take (*size ys*$_1$) *xs*$_1$ = *ys*$_1$ ∧ *drop* (*size ys*$_1$) *xs*$_1$ @ *xs*$_2$ = *ys*$_2$)
⟨*proof*⟩

lemma *take-hd-drop*:
n < *length xs* ⟹ *take n xs* @ [*hd* (*drop n xs*)] = *take* (*n*+1) *xs*
⟨*proof*⟩

lemma *id-take-nth-drop*:
i < *length xs* ⟹ *xs* = *take i xs* @ *xs*!*i* # *drop* (*Suc i*) *xs*
⟨*proof*⟩

lemma *upd-conv-take-nth-drop*:
i < *length xs* ⟹ *xs*[*i*:=*a*] = *take i xs* @ *a* # *drop* (*Suc i*) *xs*
⟨*proof*⟩

lemma *nth-drop'*:
i < *length xs* ⟹ *xs* ! *i* # *drop* (*Suc i*) *xs* = *drop i xs*

⟨*proof*⟩

40.1.15 *takeWhile* and *dropWhile*

lemma *takeWhile-dropWhile-id* [*simp*]: *takeWhile P xs* @ *dropWhile P xs* = *xs*
⟨*proof*⟩

lemma *takeWhile-append1* [*simp*]:
[| *x*:*set xs*; ~*P*(*x*)|] ==> *takeWhile P* (*xs* @ *ys*) = *takeWhile P xs*
⟨*proof*⟩

lemma *takeWhile-append2* [*simp*]:
(!!*x*. *x* : *set xs* ==> *P x*) ==> *takeWhile P* (*xs* @ *ys*) = *xs* @ *takeWhile P ys*
⟨*proof*⟩

lemma *takeWhile-tail*: ¬ *P x* ==> *takeWhile P* (*xs* @ (*x*#*l*)) = *takeWhile P xs*
⟨*proof*⟩

lemma *dropWhile-append1* [*simp*]:
[| *x* : *set xs*; ~*P*(*x*)|] ==> *dropWhile P* (*xs* @ *ys*) = (*dropWhile P xs*)@*ys*
⟨*proof*⟩

lemma *dropWhile-append2* [*simp*]:
(!!*x*. *x*:*set xs* ==> *P*(*x*)) ==> *dropWhile P* (*xs* @ *ys*) = *dropWhile P ys*
⟨*proof*⟩

lemma *set-takeWhileD*: *x* : *set* (*takeWhile P xs*) ==> *x* : *set xs* ∧ *P x*
⟨*proof*⟩

lemma *takeWhile-eq-all-conv*[*simp*]:
(*takeWhile P xs* = *xs*) = (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemma *dropWhile-eq-Nil-conv*[*simp*]:
(*dropWhile P xs* = []) = (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemma *dropWhile-eq-Cons-conv*:
(*dropWhile P xs* = *y*#*ys*) = (*xs* = *takeWhile P xs* @ *y* # *ys* & ¬ *P y*)
⟨*proof*⟩

The following two lemmmas could be generalized to an arbitrary property.

lemma *takeWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
takeWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *rev* (*tl* (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*))
⟨*proof*⟩

lemma *dropWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
dropWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *x* # *rev* (*takeWhile* (λ*y*. *y* ≠ *x*) *xs*)
⟨*proof*⟩

lemma *takeWhile-not-last*:
⟦ *xs* ≠ []; *distinct xs*⟧ ⟹ *takeWhile* (λ*y*. *y* ≠ *last xs*) *xs* = *butlast xs*
⟨*proof*⟩

lemma *takeWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *takeWhile P l* = *takeWhile Q k*
⟨*proof*⟩

lemma *dropWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *dropWhile P l* = *dropWhile Q k*
⟨*proof*⟩

40.1.16 *zip*

lemma *zip-Nil* [*simp*]: *zip* [] *ys* = []
⟨*proof*⟩

lemma *zip-Cons-Cons* [*simp*]: *zip* (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip xs ys*
⟨*proof*⟩

declare *zip-Cons* [*simp del*]

lemma *zip-Cons1*:
zip (*x*#*xs*) *ys* = (*case ys of* [] ⇒ [] | *y*#*ys* ⇒ (*x*,*y*)#*zip xs ys*)
⟨*proof*⟩

lemma *length-zip* [*simp*]:
length (*zip xs ys*) = *min* (*length xs*) (*length ys*)
⟨*proof*⟩

lemma *zip-append1*:
zip (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
⟨*proof*⟩

lemma *zip-append2*:
zip xs (*ys* @ *zs*) =
zip (*take* (*length ys*) *xs*) *ys* @ *zip* (*drop* (*length ys*) *xs*) *zs*
⟨*proof*⟩

lemma *zip-append* [*simp*]:
[| *length xs* = *length us*; *length ys* = *length vs* |] ==>
zip (*xs*@*ys*) (*us*@*vs*) = *zip xs us* @ *zip ys vs*
⟨*proof*⟩

lemma *zip-rev*:

length xs = length ys ==> zip (rev xs) (rev ys) = rev (zip xs ys)
⟨*proof*⟩

lemma *map-zip-map*:
map f (zip (map g xs) ys) = map (%(x,y). f(g x, y)) (zip xs ys)
⟨*proof*⟩

lemma *map-zip-map2*:
map f (zip xs (map g ys)) = map (%(x,y). f(x, g y)) (zip xs ys)
⟨*proof*⟩

lemma *nth-zip* [*simp*]:
[| i < length xs; i < length ys|] ==> (zip xs ys)!i = (xs!i, ys!i)
⟨*proof*⟩

lemma *set-zip*:
set (zip xs ys) = {(xs!i, ys!i) | i. i < min (length xs) (length ys)}
⟨*proof*⟩

lemma *zip-update*:
length xs = length ys ==> zip (xs[i:=x]) (ys[i:=y]) = (zip xs ys)[i:=(x,y)]
⟨*proof*⟩

lemma *zip-replicate* [*simp*]:
zip (replicate i x) (replicate j y) = replicate (min i j) (x,y)
⟨*proof*⟩

lemma *take-zip*:
take n (zip xs ys) = zip (take n xs) (take n ys)
⟨*proof*⟩

lemma *drop-zip*:
drop n (zip xs ys) = zip (drop n xs) (drop n ys)
⟨*proof*⟩

lemma *set-zip-leftD*:
(x,y)∈ set (zip xs ys) ⟹ x ∈ set xs
⟨*proof*⟩

lemma *set-zip-rightD*:
(x,y)∈ set (zip xs ys) ⟹ y ∈ set ys
⟨*proof*⟩

lemma *in-set-zipE*:
(x,y) : set(zip xs ys) ⟹ (⟦ x : set xs; y : set ys ⟧ ⟹ R) ⟹ R
⟨*proof*⟩

40.1.17 *list-all2*

lemma *list-all2-lengthD* [*intro?*]:
list-all2 P xs ys ==> length xs = length ys
⟨*proof*⟩

lemma *list-all2-Nil* [*iff*, *code*]: *list-all2 P [] ys = (ys = [])*
⟨*proof*⟩

lemma *list-all2-Nil2* [*iff*, *code*]: *list-all2 P xs [] = (xs = [])*
⟨*proof*⟩

lemma *list-all2-Cons* [*iff*, *code*]:
list-all2 P (x # xs) (y # ys) = (P x y ∧ list-all2 P xs ys)
⟨*proof*⟩

lemma *list-all2-Cons1*:
list-all2 P (x # xs) ys = (∃ z zs. ys = z # zs ∧ P x z ∧ list-all2 P xs zs)
⟨*proof*⟩

lemma *list-all2-Cons2*:
list-all2 P xs (y # ys) = (∃ z zs. xs = z # zs ∧ P z y ∧ list-all2 P zs ys)
⟨*proof*⟩

lemma *list-all2-rev* [*iff*]:
list-all2 P (rev xs) (rev ys) = list-all2 P xs ys
⟨*proof*⟩

lemma *list-all2-rev1*:
list-all2 P (rev xs) ys = list-all2 P xs (rev ys)
⟨*proof*⟩

lemma *list-all2-append1*:
list-all2 P (xs @ ys) zs =
(EX us vs. zs = us @ vs ∧ length us = length xs ∧ length vs = length ys ∧
list-all2 P xs us ∧ list-all2 P ys vs)
⟨*proof*⟩

lemma *list-all2-append2*:
list-all2 P xs (ys @ zs) =
(EX us vs. xs = us @ vs ∧ length us = length ys ∧ length vs = length zs ∧
list-all2 P us ys ∧ list-all2 P vs zs)
⟨*proof*⟩

lemma *list-all2-append*:
length xs = length ys ⟹
list-all2 P (xs@us) (ys@vs) = (list-all2 P xs ys ∧ list-all2 P us vs)
⟨*proof*⟩

lemma *list-all2-appendI* [*intro?*, *trans*]:

⟦ *list-all2 P a b*; *list-all2 P c d* ⟧ ⟹ *list-all2 P* (*a*@*c*) (*b*@*d*)
⟨*proof*⟩

lemma *list-all2-conv-all-nth*:
list-all2 P xs ys =
(*length xs* = *length ys* ∧ (∀ *i* < *length xs*. *P* (*xs*!*i*) (*ys*!*i*)))
⟨*proof*⟩

lemma *list-all2-trans*:
assumes *tr*: !!*a b c*. *P1 a b* ==> *P2 b c* ==> *P3 a c*
shows !!*bs cs*. *list-all2 P1 as bs* ==> *list-all2 P2 bs cs* ==> *list-all2 P3 as cs*
(**is** !!*bs cs*. *PROP ?Q as bs cs*)
⟨*proof*⟩

lemma *list-all2-all-nthI* [*intro?*]:
length a = *length b* ⟹ (⋀*n*. *n* < *length a* ⟹ *P* (*a*!*n*) (*b*!*n*)) ⟹ *list-all2 P a b*
⟨*proof*⟩

lemma *list-all2I*:
∀ *x* ∈ *set* (*zip a b*). *split P x* ⟹ *length a* = *length b* ⟹ *list-all2 P a b*
⟨*proof*⟩

lemma *list-all2-nthD*:
⟦ *list-all2 P xs ys*; *p* < *size xs* ⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
⟨*proof*⟩

lemma *list-all2-nthD2*:
⟦*list-all2 P xs ys*; *p* < *size ys*⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
⟨*proof*⟩

lemma *list-all2-map1*:
list-all2 P (*map f as*) *bs* = *list-all2* (λ*x y*. *P* (*f x*) *y*) *as bs*
⟨*proof*⟩

lemma *list-all2-map2*:
list-all2 P as (*map f bs*) = *list-all2* (λ*x y*. *P x* (*f y*)) *as bs*
⟨*proof*⟩

lemma *list-all2-refl* [*intro?*]:
(⋀*x*. *P x x*) ⟹ *list-all2 P xs xs*
⟨*proof*⟩

lemma *list-all2-update-cong*:
⟦ *i*<*size xs*; *list-all2 P xs ys*; *P x y* ⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
⟨*proof*⟩

lemma *list-all2-update-cong2*:
⟦*list-all2 P xs ys*; *P x y*; *i* < *length ys*⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
⟨*proof*⟩

lemma *list-all2-takeI* [*simp*,*intro?*]:
list-all2 P xs ys $\Longrightarrow$ *list-all2 P* (*take n xs*) (*take n ys*)
⟨*proof*⟩

lemma *list-all2-dropI* [*simp*,*intro?*]:
list-all2 P as bs $\Longrightarrow$ *list-all2 P* (*drop n as*) (*drop n bs*)
⟨*proof*⟩

lemma *list-all2-mono* [*intro?*]:
list-all2 P xs ys $\Longrightarrow$ ($\bigwedge$*xs ys*. *P xs ys* $\Longrightarrow$ *Q xs ys*) $\Longrightarrow$ *list-all2 Q xs ys*
⟨*proof*⟩

lemma *list-all2-eq*:
xs = *ys* $\longleftrightarrow$ *list-all2* (*op* =) *xs ys*
⟨*proof*⟩

40.1.18 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:
foldl f a (*xs* @ *ys*) = *foldl f* (*foldl f a xs*) *ys*
⟨*proof*⟩

lemma *foldr-append*[*simp*]: *foldr f* (*xs* @ *ys*) *a* = *foldr f xs* (*foldr f ys a*)
⟨*proof*⟩

lemma *foldr-map*: *foldr g* (*map f xs*) *a* = *foldr* (*g o f*) *xs a*
⟨*proof*⟩

For efficient code generation: avoid intermediate list.

lemma *foldl-map*[*code unfold*]:
foldl g a (*map f xs*) = *foldl* (%*a x*. *g a* (*f x*)) *a xs*
⟨*proof*⟩

lemma *foldl-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f a x* = *g a x* |]
==> *foldl f a l* = *foldl g b k*
⟨*proof*⟩

lemma *foldr-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f x a* = *g x a* |]
==> *foldr f l a* = *foldr g k b*
⟨*proof*⟩

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows *foldl op+* (*x*+*y*) *zs* = *x* + (*foldl op+ y zs*)
⟨*proof*⟩

lemma (**in** *monoid-add*) *foldl-absorb0*:

shows $x + (foldl\ op+\ 0\ zs) = foldl\ op+\ x\ zs$
⟨*proof*⟩

The "First Duality Theorem" in Bird & Wadler:

lemma *foldl-foldr1-lemma*:
$foldl\ op\ +\ a\ xs = a + foldr\ op\ +\ xs\ (0::'a::monoid\text{-}add)$
⟨*proof*⟩

corollary *foldl-foldr1*:
$foldl\ op\ +\ 0\ xs = foldr\ op\ +\ xs\ (0::'a::monoid\text{-}add)$
⟨*proof*⟩

The "Third Duality Theorem" in Bird & Wadler:

lemma *foldr-foldl*: $foldr\ f\ xs\ a = foldl\ (\%x\ y.\ f\ y\ x)\ a\ (rev\ xs)$
⟨*proof*⟩

lemma *foldl-foldr*: $foldl\ f\ a\ xs = foldr\ (\%x\ y.\ f\ y\ x)\ (rev\ xs)\ a$
⟨*proof*⟩

lemma (**in** *ab-semigroup-add*) *foldr-conv-foldl*: $foldr\ op\ +\ xs\ a = foldl\ op\ +\ a\ xs$
⟨*proof*⟩

Note: $n \leq foldl\ (op\ +)\ n\ ns$ looks simpler, but is more difficult to use because it requires an additional transitivity step.

lemma *start-le-sum*: $(m::nat) <= n ==> m <= foldl\ (op\ +)\ n\ ns$
⟨*proof*⟩

lemma *elem-le-sum*: $(n::nat) : set\ ns ==> n <= foldl\ (op\ +)\ 0\ ns$
⟨*proof*⟩

lemma *sum-eq-0-conv* [*iff*]:
$(foldl\ (op\ +)\ (m::nat)\ ns = 0) = (m = 0 \wedge (\forall n \in set\ ns.\ n = 0))$
⟨*proof*⟩

lemma *foldr-invariant*:
$[\![Q\ x\ ;\ \forall\ x{\in}\ set\ xs.\ P\ x;\ \forall\ x\ y.\ P\ x \wedge Q\ y \longrightarrow Q\ (f\ x\ y)\ ]\!] \Longrightarrow Q\ (foldr\ f\ xs\ x)$
⟨*proof*⟩

lemma *foldl-invariant*:
$[\![Q\ x\ ;\ \forall\ x{\in}\ set\ xs.\ P\ x;\ \forall\ x\ y.\ P\ x \wedge Q\ y \longrightarrow Q\ (f\ y\ x)\ ]\!] \Longrightarrow Q\ (foldl\ f\ x\ xs)$
⟨*proof*⟩

foldl and *concat*

lemma *concat-conv-foldl*: $concat\ xss = foldl\ op@\ []\ xss$
⟨*proof*⟩

lemma *foldl-conv-concat*:
$foldl\ (op\ @)\ xs\ xxs = xs\ @\ (concat\ xxs)$
⟨*proof*⟩

40.1.19 List summation: *listsum* and $\sum$

lemma *listsum-append* [*simp*]: *listsum* (*xs* @ *ys*) = *listsum xs* + *listsum ys*
⟨*proof*⟩

lemma *listsum-rev* [*simp*]:
fixes *xs* :: *'a*::*comm-monoid-add list*
shows *listsum* (*rev xs*) = *listsum xs*
⟨*proof*⟩

lemma *listsum-foldr*: *listsum xs* = *foldr* (*op* +) *xs 0*
⟨*proof*⟩

lemma *length-concat*: *length* (*concat xss*) = *listsum* (*map length xss*)
⟨*proof*⟩

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

lemma *listsum*[*code unfold*]: *listsum xs* = *foldl* (*op* +) *0 xs*
⟨*proof*⟩

Some syntactic sugar for summing a function over a list:

syntax
-listsum :: *pttrn* => *'a list* => *'b* => *'b* (*3SUM -<−-. -*) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*$\sum$ *-←-. -*) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*$\sum$ *-←-. -*) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM x<−xs. b == *CONST listsum* (*map* (%*x. b*) *xs*)
$\sum$ *x←xs. b* == *CONST listsum* (*map* (%*x. b*) *xs*)

lemma *listsum-triv*: ($\sum$ *x←xs. r*) = *of-nat* (*length xs*) * *r*
⟨*proof*⟩

lemma *listsum-0* [*simp*]: ($\sum$ *x←xs. 0*) = *0*
⟨*proof*⟩

For non-Abelian groups *xs* needs to be reversed on one side:

lemma *uminus-listsum-map*:
fixes *f* :: *'a* ⇒ *'b*::*ab-group-add*
shows − *listsum* (*map f xs*) = (*listsum* (*map* (*uminus o f*) *xs*))
⟨*proof*⟩

40.1.20 *upt*

lemma *upt-rec*[*code*]: [*i..<j*] = (*if i<j then i#*[*Suc i..<j*] *else* []) — simp does not terminate!
⟨*proof*⟩

lemma *upt-conv-Nil* [*simp*]: $j <= i ==> [i..<j] = []$
⟨*proof*⟩

lemma *upt-eq-Nil-conv*[*simp*]: $([i..<j] = []) = (j = 0 \vee j <= i)$
⟨*proof*⟩

lemma *upt-eq-Cons-conv*:
$([i..<j] = x\#xs) = (i < j \ \& \ i = x \ \& \ [i+1..<j] = xs)$
⟨*proof*⟩

lemma *upt-Suc-append*: $i <= j ==> [i..<(Suc\ j)] = [i..<j]@[j]$
— Only needed if *upt-Suc* is deleted from the simpset.
⟨*proof*⟩

lemma *upt-conv-Cons*: $i < j ==> [i..<j] = i \# [Suc\ i..<j]$
⟨*proof*⟩

lemma *upt-add-eq-append*: $i<=j ==> [i..<j+k] = [i..<j]@[j..<j+k]$
— LOOPS as a simprule, since $j <= j$.
⟨*proof*⟩

lemma *length-upt* [*simp*]: $length\ [i..<j] = j - i$
⟨*proof*⟩

lemma *nth-upt* [*simp*]: $i + k < j ==> [i..<j] ! k = i + k$
⟨*proof*⟩

lemma *hd-upt*[*simp*]: $i < j \Longrightarrow hd[i..<j] = i$
⟨*proof*⟩

lemma *last-upt*[*simp*]: $i < j \Longrightarrow last[i..<j] = j - 1$
⟨*proof*⟩

lemma *take-upt* [*simp*]: $i+m <= n ==> take\ m\ [i..<n] = [i..<i+m]$
⟨*proof*⟩

lemma *drop-upt*[*simp*]: $drop\ m\ [i..<j] = [i+m..<j]$
⟨*proof*⟩

lemma *map-Suc-upt*: $map\ Suc\ [m..<n] = [Suc\ m..<Suc\ n]$
⟨*proof*⟩

lemma *nth-map-upt*: $i < n-m ==> (map\ f\ [m..<n]) ! i = f(m+i)$
⟨*proof*⟩

lemma *nth-take-lemma*:
$k <= length\ xs ==> k <= length\ ys ==>$

(!!*i*. *i* < *k* −−> *xs*!*i* = *ys*!*i*) ==> *take k xs* = *take k ys*
⟨*proof*⟩

lemma *nth-equalityI*:
[| *length xs* = *length ys*; *ALL i* < *length xs*. *xs*!*i* = *ys*!*i* |] ==> *xs* = *ys*
⟨*proof*⟩

lemma *map-nth*:
map (λ*i*. *xs* ! *i*) [*0*..<*length xs*] = *xs*
⟨*proof*⟩

lemma *list-all2-antisym*:
⟦ (⋀*x y*. ⟦*P x y*; *Q y x*⟧ ⟹ *x* = *y*); *list-all2 P xs ys*; *list-all2 Q ys xs* ⟧
⟹ *xs* = *ys*
⟨*proof*⟩

lemma *take-equalityI*: (∀ *i*. *take i xs* = *take i ys*) ==> *xs* = *ys*
— The famous take-lemma.
⟨*proof*⟩

lemma *take-Cons'*:
take n (*x* # *xs*) = (*if n* = *0 then* [] *else x* # *take* (*n* − *1*) *xs*)
⟨*proof*⟩

lemma *drop-Cons'*:
drop n (*x* # *xs*) = (*if n* = *0 then x* # *xs else drop* (*n* − *1*) *xs*)
⟨*proof*⟩

lemma *nth-Cons'*: (*x* # *xs*)!*n* = (*if n* = *0 then x else xs*!(*n* − *1*))
⟨*proof*⟩

lemmas *take-Cons-number-of* = *take-Cons'*[*of number-of v*,*standard*]
lemmas *drop-Cons-number-of* = *drop-Cons'*[*of number-of v*,*standard*]
lemmas *nth-Cons-number-of* = *nth-Cons'*[*of - - number-of v*,*standard*]

declare *take-Cons-number-of* [*simp*]
drop-Cons-number-of [*simp*]
nth-Cons-number-of [*simp*]

40.1.21 *distinct* and *remdups*

lemma *distinct-append* [*simp*]:
distinct (*xs* @ *ys*) = (*distinct xs* ∧ *distinct ys* ∧ *set xs* ∩ *set ys* = {})
⟨*proof*⟩

lemma *distinct-rev*[*simp*]: *distinct*(*rev xs*) = *distinct xs*
⟨*proof*⟩

lemma *set-remdups* [*simp*]: *set* (*remdups xs*) = *set xs*
⟨*proof*⟩

lemma *distinct-remdups* [*iff*]: *distinct* (*remdups xs*)
⟨*proof*⟩

lemma *distinct-remdups-id*: *distinct xs* ==> *remdups xs* = *xs*
⟨*proof*⟩

lemma *remdups-id-iff-distinct* [*simp*]: *remdups xs* = *xs* ⟷ *distinct xs*
⟨*proof*⟩

lemma *finite-distinct-list*: *finite A* ⟹ *EX xs*. *set xs* = *A* & *distinct xs*
⟨*proof*⟩

lemma *remdups-eq-nil-iff* [*simp*]: (*remdups x* = []) = (*x* = [])
⟨*proof*⟩

lemma *remdups-eq-nil-right-iff* [*simp*]: ([] = *remdups x*) = (*x* = [])
⟨*proof*⟩

lemma *length-remdups-leq*[*iff*]: *length*(*remdups xs*) <= *length xs*
⟨*proof*⟩

lemma *length-remdups-eq*[*iff*]:
(*length* (*remdups xs*) = *length xs*) = (*remdups xs* = *xs*)
⟨*proof*⟩

lemma *distinct-map*:
distinct(*map f xs*) = (*distinct xs* & *inj-on f* (*set xs*))
⟨*proof*⟩

lemma *distinct-filter* [*simp*]: *distinct xs* ==> *distinct* (*filter P xs*)
⟨*proof*⟩

lemma *distinct-upt*[*simp*]: *distinct*[*i*..<*j*]
⟨*proof*⟩

lemma *distinct-take*[*simp*]: *distinct xs* ⟹ *distinct* (*take i xs*)
⟨*proof*⟩

lemma *distinct-drop*[*simp*]: *distinct xs* ⟹ *distinct* (*drop i xs*)
⟨*proof*⟩

lemma *distinct-list-update*:
assumes *d*: *distinct xs* **and** *a*: *a* ∉ *set xs* − {*xs*!*i*}

shows *distinct* $(xs[i:=a])$
⟨*proof*⟩

It is best to avoid this indexed version of distinct, but sometimes it is useful.

lemma *distinct-conv-nth*:
$distinct\ xs = (\forall i < size\ xs.\ \forall j < size\ xs.\ i \neq j \longrightarrow xs!i \neq xs!j)$
⟨*proof*⟩

lemma *nth-eq-iff-index-eq*:
$[\![\ distinct\ xs;\ i < length\ xs;\ j < length\ xs\ ]\!] \Longrightarrow (xs!i = xs!j) = (i = j)$
⟨*proof*⟩

lemma *distinct-card*: $distinct\ xs \Longrightarrow card\ (set\ xs) = size\ xs$
⟨*proof*⟩

lemma *card-distinct*: $card\ (set\ xs) = size\ xs \Longrightarrow distinct\ xs$
⟨*proof*⟩

lemma *not-distinct-decomp*: $\sim distinct\ ws \Longrightarrow EX\ xs\ ys\ zs\ y.\ ws = xs@[y]@ys@[y]@zs$
⟨*proof*⟩

lemma *length-remdups-concat*:
$length(remdups(concat\ xss)) = card(\bigcup xs \in set\ xss.\ set\ xs)$
⟨*proof*⟩

40.1.22 *remove1*

lemma *remove1-append*:
$remove1\ x\ (xs\ @\ ys) =$
$(if\ x \in set\ xs\ then\ remove1\ x\ xs\ @\ ys\ else\ xs\ @\ remove1\ x\ ys)$
⟨*proof*⟩

lemma *in-set-remove1*[*simp*]:
$a \neq b \Longrightarrow a : set(remove1\ b\ xs) = (a : set\ xs)$
⟨*proof*⟩

lemma *set-remove1-subset*: $set(remove1\ x\ xs) <= set\ xs$
⟨*proof*⟩

lemma *set-remove1-eq* [*simp*]: $distinct\ xs \Longrightarrow set(remove1\ x\ xs) = set\ xs - \{x\}$
⟨*proof*⟩

lemma *length-remove1*:
$length(remove1\ x\ xs) = (if\ x : set\ xs\ then\ length\ xs - 1\ else\ length\ xs)$
⟨*proof*⟩

lemma *remove1-filter-not*[*simp*]:
$\neg\ P\ x \Longrightarrow remove1\ x\ (filter\ P\ xs) = filter\ P\ xs$
⟨*proof*⟩

lemma *notin-set-remove1*[*simp*]: *x* ~: *set xs* ==> *x* ~: *set*(*remove1 y xs*)
⟨*proof*⟩

lemma *distinct-remove1*[*simp*]: *distinct xs* ==> *distinct*(*remove1 x xs*)
⟨*proof*⟩

40.1.23 *replicate*

lemma *length-replicate* [*simp*]: *length* (*replicate n x*) = *n*
⟨*proof*⟩

lemma *map-replicate* [*simp*]: *map f* (*replicate n x*) = *replicate n* (*f x*)
⟨*proof*⟩

lemma *replicate-app-Cons-same*:
(*replicate n x*) @ (*x* # *xs*) = *x* # *replicate n x* @ *xs*
⟨*proof*⟩

lemma *rev-replicate* [*simp*]: *rev* (*replicate n x*) = *replicate n x*
⟨*proof*⟩

lemma *replicate-add*: *replicate* (*n* + *m*) *x* = *replicate n x* @ *replicate m x*
⟨*proof*⟩

Courtesy of Matthias Daum:

lemma *append-replicate-commute*:
replicate n x @ *replicate k x* = *replicate k x* @ *replicate n x*
⟨*proof*⟩

lemma *hd-replicate* [*simp*]: *n* ≠ *0* ==> *hd* (*replicate n x*) = *x*
⟨*proof*⟩

lemma *tl-replicate* [*simp*]: *n* ≠ *0* ==> *tl* (*replicate n x*) = *replicate* (*n* − *1*) *x*
⟨*proof*⟩

lemma *last-replicate* [*simp*]: *n* ≠ *0* ==> *last* (*replicate n x*) = *x*
⟨*proof*⟩

lemma *nth-replicate*[*simp*]: *i* < *n* ==> (*replicate n x*)!*i* = *x*
⟨*proof*⟩

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[*simp*]: *take i* (*replicate k x*) = *replicate* (*min i k*) *x*
⟨*proof*⟩

lemma *drop-replicate*[*simp*]: *drop i* (*replicate k x*) = *replicate* (*k*−*i*) *x*
⟨*proof*⟩

lemma *set-replicate-Suc*: *set* (*replicate* (*Suc n*) *x*) = {*x*}
⟨*proof*⟩

lemma *set-replicate* [*simp*]: *n* ≠ *0* ==> *set* (*replicate n x*) = {*x*}
⟨*proof*⟩

lemma *set-replicate-conv-if*: *set* (*replicate n x*) = (*if n* = *0 then* {} *else* {*x*})
⟨*proof*⟩

lemma *in-set-replicateD*: *x* : *set* (*replicate n y*) ==> *x* = *y*
⟨*proof*⟩

lemma *replicate-append-same*:
replicate i x @ [*x*] = *x* # *replicate i x*
⟨*proof*⟩

lemma *map-replicate-trivial*:
map (λ*i*. *x*) [*0*..<*i*] = *replicate i x*
⟨*proof*⟩

40.1.24 *rotate1* and *rotate*

lemma *rotate-simps*[*simp*]: *rotate1* [] = [] ∧ *rotate1* (*x*#*xs*) = *xs* @ [*x*]
⟨*proof*⟩

lemma *rotate0*[*simp*]: *rotate 0* = *id*
⟨*proof*⟩

lemma *rotate-Suc*[*simp*]: *rotate* (*Suc n*) *xs* = *rotate1*(*rotate n xs*)
⟨*proof*⟩

lemma *rotate-add*:
rotate (*m*+*n*) = *rotate m o rotate n*
⟨*proof*⟩

lemma *rotate-rotate*: *rotate m* (*rotate n xs*) = *rotate* (*m*+*n*) *xs*
⟨*proof*⟩

lemma *rotate1-rotate-swap*: *rotate1* (*rotate n xs*) = *rotate n* (*rotate1 xs*)
⟨*proof*⟩

lemma *rotate1-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate1 xs* = *xs*
⟨*proof*⟩

lemma *rotate-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate n xs* = *xs*
⟨*proof*⟩

lemma *rotate1-hd-tl*: *xs* ≠ [] ⟹ *rotate1 xs* = *tl xs* @ [*hd xs*]

⟨*proof*⟩

lemma *rotate-drop-take*:
rotate n xs = *drop* (*n mod length xs*) *xs* @ *take* (*n mod length xs*) *xs*
⟨*proof*⟩

lemma *rotate-conv-mod*: *rotate n xs* = *rotate* (*n mod length xs*) *xs*
⟨*proof*⟩

lemma *rotate-id*[*simp*]: *n mod length xs* = *0* ⟹ *rotate n xs* = *xs*
⟨*proof*⟩

lemma *length-rotate1*[*simp*]: *length*(*rotate1 xs*) = *length xs*
⟨*proof*⟩

lemma *length-rotate*[*simp*]: *length*(*rotate n xs*) = *length xs*
⟨*proof*⟩

lemma *distinct1-rotate*[*simp*]: *distinct*(*rotate1 xs*) = *distinct xs*
⟨*proof*⟩

lemma *distinct-rotate*[*simp*]: *distinct*(*rotate n xs*) = *distinct xs*
⟨*proof*⟩

lemma *rotate-map*: *rotate n* (*map f xs*) = *map f* (*rotate n xs*)
⟨*proof*⟩

lemma *set-rotate1*[*simp*]: *set*(*rotate1 xs*) = *set xs*
⟨*proof*⟩

lemma *set-rotate*[*simp*]: *set*(*rotate n xs*) = *set xs*
⟨*proof*⟩

lemma *rotate1-is-Nil-conv*[*simp*]: (*rotate1 xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *rotate-is-Nil-conv*[*simp*]: (*rotate n xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *rotate-rev*:
rotate n (*rev xs*) = *rev*(*rotate* (*length xs* − (*n mod length xs*)) *xs*)
⟨*proof*⟩

lemma *hd-rotate-conv-nth*: *xs* ≠ [] ⟹ *hd*(*rotate n xs*) = *xs*!(*n mod length xs*)
⟨*proof*⟩

40.1.25 *sublist* — a generalization of *nth* to sets

lemma *sublist-empty* [*simp*]: *sublist xs* {} = []

⟨*proof*⟩

lemma *sublist-nil* [*simp*]: *sublist* [] *A* = []
⟨*proof*⟩

lemma *length-sublist*:
length(*sublist xs I*) = *card*{*i*. *i* < *length xs* ∧ *i* : *I*}
⟨*proof*⟩

lemma *sublist-shift-lemma-Suc*:
map fst (*filter* (%*p*. *P*(*Suc*(*snd p*))) (*zip xs is*)) =
map fst (*filter* (%*p*. *P*(*snd p*)) (*zip xs* (*map Suc is*)))
⟨*proof*⟩

lemma *sublist-shift-lemma*:
map fst [*p*<−*zip xs* [*i*..<*i* + *length xs*] . *snd p* : *A*] =
map fst [*p*<−*zip xs* [0..<*length xs*] . *snd p* + *i* : *A*]
⟨*proof*⟩

lemma *sublist-append*:
sublist (*l* @ *l'*) *A* = *sublist l A* @ *sublist l'* {*j*. *j* + *length l* : *A*}
⟨*proof*⟩

lemma *sublist-Cons*:
sublist (*x* # *l*) *A* = (*if* 0:*A then* [*x*] *else* []) @ *sublist l* {*j*. *Suc j* : *A*}
⟨*proof*⟩

lemma *set-sublist*: *set*(*sublist xs I*) = {*xs*!*i*|*i*. *i*<*size xs* ∧ *i* ∈ *I*}
⟨*proof*⟩

lemma *set-sublist-subset*: *set*(*sublist xs I*) ⊆ *set xs*
⟨*proof*⟩

lemma *notin-set-sublistI*[*simp*]: *x* ∉ *set xs* ⟹ *x* ∉ *set*(*sublist xs I*)
⟨*proof*⟩

lemma *in-set-sublistD*: *x* ∈ *set*(*sublist xs I*) ⟹ *x* ∈ *set xs*
⟨*proof*⟩

lemma *sublist-singleton* [*simp*]: *sublist* [*x*] *A* = (*if* 0 : *A then* [*x*] *else* [])
⟨*proof*⟩

lemma *distinct-sublistI*[*simp*]: *distinct xs* ⟹ *distinct*(*sublist xs I*)
⟨*proof*⟩

lemma *sublist-upt-eq-take* [*simp*]: *sublist l* {..<*n*} = *take n l*
⟨*proof*⟩

lemma *filter-in-sublist*:
distinct xs $\Longrightarrow$ *filter* (%*x*. *x* $\in$ *set*(*sublist xs s*)) *xs* = *sublist xs s*
⟨*proof*⟩

40.1.26 *splice*

lemma *splice-Nil2* [*simp*, *code*]:
splice xs [] = *xs*
⟨*proof*⟩

lemma *splice-Cons-Cons* [*simp*, *code*]:
splice (*x*#*xs*) (*y*#*ys*) = *x* # *y* # *splice xs ys*
⟨*proof*⟩

declare *splice.simps*(*2*) [*simp del*, *code del*]

lemma *length-splice*[*simp*]: *length*(*splice xs ys*) = *length xs* + *length ys*
⟨*proof*⟩

40.2 Sorting

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context *linorder*
begin

lemma *sorted-Cons*: *sorted* (*x*#*xs*) = (*sorted xs* & (*ALL y*:*set xs*. *x* <= *y*))
⟨*proof*⟩

lemma *sorted-append*:
sorted (*xs*@*ys*) = (*sorted xs* & *sorted ys* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *x*≤*y*))
⟨*proof*⟩

lemma *set-insort*: *set*(*insort x xs*) = *insert x* (*set xs*)
⟨*proof*⟩

lemma *set-sort*[*simp*]: *set*(*sort xs*) = *set xs*
⟨*proof*⟩

lemma *distinct-insort*: *distinct* (*insort x xs*) = (*x* ∉ *set xs* ∧ *distinct xs*)
⟨*proof*⟩

lemma *distinct-sort*[*simp*]: *distinct* (*sort xs*) = *distinct xs*
⟨*proof*⟩

lemma *sorted-insort*: *sorted* (*insort x xs*) = *sorted xs*
⟨*proof*⟩

theorem *sorted-sort*[*simp*]: *sorted* (*sort xs*)
⟨*proof*⟩

lemma *insort-is-Cons*: ∀ *x*∈*set xs*. *a* ≤ *x* ⟹ *insort a xs* = *a* # *xs*
⟨*proof*⟩

lemma *sorted-remove1*: *sorted xs* ⟹ *sorted* (*remove1 a xs*)
⟨*proof*⟩

lemma *insort-remove1*: ⟦ *a* ∈ *set xs*; *sorted xs* ⟧ ⟹ *insort a* (*remove1 a xs*) = *xs*
⟨*proof*⟩

lemma *sorted-remdups*[*simp*]:
sorted l ⟹ *sorted* (*remdups l*)
⟨*proof*⟩

lemma *sorted-distinct-set-unique*:
assumes *sorted xs distinct xs sorted ys distinct ys set xs* = *set ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *finite-sorted-distinct-unique*:
shows *finite A* ⟹ *EX*! *xs*. *set xs* = *A* & *sorted xs* & *distinct xs*
⟨*proof*⟩

end

lemma *sorted-upt*[*simp*]: *sorted*[*i*..<*j*]
⟨*proof*⟩

40.2.1 *sorted-list-of-set*

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

context *linorder*
begin

definition
sorted-list-of-set :: *'a set* ⇒ *'a list* **where**
sorted-list-of-set A == *THE xs*. *set xs* = *A* & *sorted xs* & *distinct xs*

lemma *sorted-list-of-set*[*simp*]: *finite A* ⟹
set(*sorted-list-of-set A*) = *A* &
sorted(*sorted-list-of-set A*) & *distinct*(*sorted-list-of-set A*)

⟨*proof*⟩

lemma *sorted-list-of-empty*[*simp*]: *sorted-list-of-set* {} = []
⟨*proof*⟩

end

40.2.2 *upto*: the generic interval-list

class *finite-intvl-succ* = *linorder* +
fixes *successor* :: *'a* ⇒ *'a*
assumes *finite-intvl*: *finite*{*a..b*}
and *successor-incr*: *a* < *successor a*
and *ord-discrete*: ¬(∃ *x*. *a* < *x* & *x* < *successor a*)

context *finite-intvl-succ*
begin

definition
upto :: *'a* ⇒ *'a* ⇒ *'a list* ((*1*[-../-])) **where**
upto i j == *sorted-list-of-set* {*i..j*}

lemma *upto*[*simp*]: *set*[*a..b*] = {*a..b*} & *sorted*[*a..b*] & *distinct*[*a..b*]
⟨*proof*⟩

lemma *insert-intvl*: *i* ≤ *j* ⟹ *insert i* {*successor i..j*} = {*i..j*}
⟨*proof*⟩

lemma *sorted-list-of-set-rec*: *i* ≤ *j* ⟹
sorted-list-of-set {*i..j*} = *i* # *sorted-list-of-set* {*successor i..j*}
⟨*proof*⟩

lemma *upto-rec*[*code*]: [*i..j*] = (*if i* ≤ *j then i* # [*successor i..j*] *else* [])
⟨*proof*⟩

end

The integers are an instance of the above class:

instantiation *int*:: *finite-intvl-succ*
begin

definition
successor-int-def: *successor* = (%*i*::*int*. *i*+*1*)

instance
⟨*proof*⟩

end

Now [*i..j*] is defined for integers.

hide (**open**) *const successor*

40.2.3 *lists*: the list-forming operator over sets

inductive-set
  lists :: *'a set* => *'a list set*
  for *A* :: *'a set*
where
    Nil [*intro*!]: []: *lists A*
  | *Cons* [*intro*!,*noatp*]: [| *a*: *A*;*l*: *lists A*|] ==> *a*#*l* : *lists A*

inductive-cases *listsE* [*elim*!,*noatp*]: *x*#*l* : *lists A*
inductive-cases *listspE* [*elim*!,*noatp*]: *listsp A* (*x* # *l*)

lemma *listsp-mono* [*mono*]: *A* ≤ *B* ==> *listsp A* ≤ *listsp B*
⟨*proof*⟩

lemmas *lists-mono* = *listsp-mono* [*to-set pred-subset-eq*]

lemma *listsp-infI*:
  assumes *l*: *listsp A l* **shows** *listsp B l* ==> *listsp* (*inf A B*) *l* ⟨*proof*⟩

lemmas *lists-IntI* = *listsp-infI* [*to-set*]

lemma *listsp-inf-eq* [*simp*]: *listsp* (*inf A B*) = *inf* (*listsp A*) (*listsp B*)
⟨*proof*⟩

lemmas *listsp-conj-eq* [*simp*] = *listsp-inf-eq* [*simplified inf-fun-eq inf-bool-eq*]

lemmas *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set pred-equals-eq*]

lemma *append-in-listsp-conv* [*iff*]:
    (*listsp A* (*xs* @ *ys*)) = (*listsp A xs* ∧ *listsp A ys*)
⟨*proof*⟩

lemmas *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

lemma *in-listsp-conv-set*: (*listsp A xs*) = (∀ *x* ∈ *set xs*. *A x*)
— eliminate *listsp* in favour of *set*
⟨*proof*⟩

lemmas *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest*!,*noatp*]: *listsp A xs* ==> ∀ *x*∈*set xs*. *A x*
⟨*proof*⟩

lemmas *in-listsD* [*dest*!,*noatp*] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro*!,*noatp*]: ∀ *x*∈*set xs*. *A x* ==> *listsp A xs*

⟨*proof*⟩

lemmas *in-listsI* [*intro*!,*noatp*] = *in-listspI* [*to-set*]

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
⟨*proof*⟩

40.2.4 Inductive definition for membership

inductive *ListMem* :: *'a* ⇒ *'a list* ⇒ *bool*
where
  elem: *ListMem x* (*x* # *xs*)
| *insert*: *ListMem x xs* ⟹ *ListMem x* (*y* # *xs*)

lemma *ListMem-iff*: (*ListMem x xs*) = (*x* ∈ *set xs*)
⟨*proof*⟩

40.2.5 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from *A* and tail drawn from *Xs*.

constdefs
  set-Cons :: *'a set* ⇒ *'a list set* ⇒ *'a list set*
  set-Cons A XS == {*z*. ∃*x xs*. *z* = *x*#*xs* & *x* ∈ *A* & *xs* ∈ *XS*}

lemma *set-Cons-sing-Nil* [*simp*]: *set-Cons A* {[]} = (%*x*. [*x*])‘*A*
⟨*proof*⟩

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

consts *listset* :: *'a set list* ⇒ *'a list set*
primrec
  listset [] = {[]}
  listset(*A*#*As*) = *set-Cons A* (*listset As*)

40.3 Relations on Lists

40.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

consts *lexn* :: (*'a* * *'a*)*set* => *nat* => (*'a list* * *'a list*)*set*
      — The lexicographic ordering for lists of the specified length
primrec
  lexn r 0 = {}
  lexn r (*Suc n*) =
    (*prod-fun* (%(*x*,*xs*). *x*#*xs*) (%(*x*,*xs*). *x*#*xs*) ‘ (*r* <*lex*> *lexn r n*)) *Int*
    {(*xs*,*ys*). *length xs* = *Suc n* ∧ *length ys* = *Suc n*}

constdefs
  lex :: (*'a* × *'a*) *set* => (*'a list* × *'a list*) *set*
    lex r == ⋃ *n*. *lexn r n*
      — Holds only between lists of the same length

  lenlex :: (*'a* × *'a*) *set* => (*'a list* × *'a list*) *set*
    lenlex r == *inv-image* (*less-than* <*lex*> *lex r*) (%*xs*. (*length xs*, *xs*))
      — Compares lists by their length and then lexicographically

lemma *wf-lexn*: *wf r* ==> *wf* (*lexn r n*)
⟨*proof*⟩

lemma *lexn-length*:
  (*xs*, *ys*) : *lexn r n* ==> *length xs* = *n* ∧ *length ys* = *n*
⟨*proof*⟩

lemma *wf-lex* [*intro*!]: *wf r* ==> *wf* (*lex r*)
⟨*proof*⟩

lemma *lexn-conv*:
  lexn r n =
    {(*xs*,*ys*). *length xs* = *n* ∧ *length ys* = *n* ∧
    (∃ *xys x y xs' ys'*. *xs*= *xys* @ *x*#*xs'* ∧ *ys*= *xys* @ *y* # *ys'* ∧ (*x*, *y*):*r*)}
⟨*proof*⟩

lemma *lex-conv*:
  lex r =
    {(*xs*,*ys*). *length xs* = *length ys* ∧
    (∃ *xys x y xs' ys'*. *xs* = *xys* @ *x* # *xs'* ∧ *ys* = *xys* @ *y* # *ys'* ∧ (*x*, *y*):*r*)}
⟨*proof*⟩

lemma *wf-lenlex* [*intro*!]: *wf r* ==> *wf* (*lenlex r*)
⟨*proof*⟩

lemma *lenlex-conv*:
    lenlex r = {(*xs*,*ys*). *length xs* < *length ys* |
                length xs = *length ys* ∧ (*xs*, *ys*) : *lex r*}
⟨*proof*⟩

lemma *Nil-notin-lex* [*iff*]: ([], *ys*) ∉ *lex r*
⟨*proof*⟩

lemma *Nil2-notin-lex* [*iff*]: (*xs*, []) ∉ *lex r*
⟨*proof*⟩

lemma *Cons-in-lex* [*simp*]:
    ((*x* # *xs*, *y* # *ys*) : *lex r*) =

$((x,\ y) : r \wedge \mathit{length}\ xs = \mathit{length}\ ys \mid x = y \wedge (xs,\ ys) : \mathit{lex}\ r)$
⟨*proof*⟩

40.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" ¡ "ab" ¡ "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

constdefs
lexord :: $('a * 'a)set \Rightarrow ('a\ list * 'a\ list)\ set$
lexord $r == \{(x,y).\ \exists\ a\ v.\ y = x\ @\ a\ \#\ v \vee$
$(\exists\ u\ a\ b\ v\ w.\ (a,b) \in r \wedge x = u\ @\ (a\ \#\ v) \wedge y = u\ @\ (b\ \#\ w))\}$

lemma *lexord-Nil-left*[*simp*]: $([],y) \in \mathit{lexord}\ r = (\exists\ a\ x.\ y = a\ \#\ x)$
⟨*proof*⟩

lemma *lexord-Nil-right*[*simp*]: $(x,[]) \notin \mathit{lexord}\ r$
⟨*proof*⟩

lemma *lexord-cons-cons*[*simp*]:
$((a\ \#\ x,\ b\ \#\ y) \in \mathit{lexord}\ r) = ((a,b) \in r \mid (a = b\ \&\ (x,y) \in \mathit{lexord}\ r))$
⟨*proof*⟩

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-append-rightI*: $\exists\ b\ z.\ y = b\ \#\ z \Longrightarrow (x,\ x\ @\ y) \in \mathit{lexord}\ r$
⟨*proof*⟩

lemma *lexord-append-left-rightI*:
$(a,b) \in r \Longrightarrow (u\ @\ a\ \#\ x,\ u\ @\ b\ \#\ y) \in \mathit{lexord}\ r$
⟨*proof*⟩

lemma *lexord-append-leftI*: $(u,v) \in \mathit{lexord}\ r \Longrightarrow (x\ @\ u,\ x\ @\ v) \in \mathit{lexord}\ r$
⟨*proof*⟩

lemma *lexord-append-leftD*:
$[\![\ (x\ @\ u,\ x\ @\ v) \in \mathit{lexord}\ r;\ (!\ a.\ (a,a) \notin r)\ ]\!] \Longrightarrow (u,v) \in \mathit{lexord}\ r$
⟨*proof*⟩

lemma *lexord-take-index-conv*:
$((x,y) : \mathit{lexord}\ r) =$
$((\mathit{length}\ x < \mathit{length}\ y \wedge \mathit{take}\ (\mathit{length}\ x)\ y = x) \vee$
$(\exists i.\ i < \mathit{min}(\mathit{length}\ x)(\mathit{length}\ y)\ \&\ \mathit{take}\ i\ x = \mathit{take}\ i\ y\ \&\ (x!i,y!i) \in r))$
⟨*proof*⟩
lemma *lexord-lex*: $(x,y) \in \mathit{lex}\ r = ((x,y) \in \mathit{lexord}\ r \wedge \mathit{length}\ x = \mathit{length}\ y)$
⟨*proof*⟩

lemma *lexord-irreflexive*: $(!\ x.\ (x,x) \notin r) \Longrightarrow (y,y) \notin \mathit{lexord}\ r$
⟨*proof*⟩

lemma *lexord-trans*:
⟦ (*x*, *y*) ∈ *lexord r*; (*y*, *z*) ∈ *lexord r*; *trans r* ⟧ ⟹ (*x*, *z*) ∈ *lexord r*
⟨*proof*⟩

lemma *lexord-transI*: *trans r* ⟹ *trans* (*lexord r*)
⟨*proof*⟩

lemma *lexord-linear*: (! *a b*. (*a*,*b*)∈ *r* | *a* = *b* | (*b*,*a*) ∈ *r*) ⟹ (*x*,*y*) : *lexord r* | *x* = *y* | (*y*,*x*) : *lexord r*
⟨*proof*⟩

40.4 Lexicographic combination of measure functions

These are useful for termination proofs

definition
measures fs = *inv-image* (*lex less-than*) (%*a*. *map* (%*f*. *f a*) *fs*)

lemma *wf-measures*[*recdef-wf*, *simp*]: *wf* (*measures fs*)
⟨*proof*⟩

lemma *in-measures*[*simp*]:
(*x*, *y*) ∈ *measures* [] = *False*
(*x*, *y*) ∈ *measures* (*f* # *fs*)
= (*f x* < *f y* ∨ (*f x* = *f y* ∧ (*x*, *y*) ∈ *measures fs*))
⟨*proof*⟩

lemma *measures-less*: *f x* < *f y* ==> (*x*, *y*) ∈ *measures* (*f*#*fs*)
⟨*proof*⟩

lemma *measures-lesseq*: *f x* <= *f y* ==> (*x*, *y*) ∈ *measures fs* ==> (*x*, *y*) ∈ *measures* (*f*#*fs*)
⟨*proof*⟩

40.4.1 Lifting a Relation on List Elements to the Lists

inductive-set
listrel :: ('*a* ∗ '*a*)*set* => ('*a list* ∗ '*a list*)*set*
for *r* :: ('*a* ∗ '*a*)*set*
where
Nil: ([],[]) ∈ *listrel r*
| *Cons*: [| (*x*,*y*) ∈ *r*; (*xs*,*ys*) ∈ *listrel r* |] ==> (*x*#*xs*, *y*#*ys*) ∈ *listrel r*

inductive-cases *listrel-Nil1* [*elim*!]: ([],*xs*) ∈ *listrel r*
inductive-cases *listrel-Nil2* [*elim*!]: (*xs*,[]) ∈ *listrel r*
inductive-cases *listrel-Cons1* [*elim*!]: (*y*#*ys*,*xs*) ∈ *listrel r*
inductive-cases *listrel-Cons2* [*elim*!]: (*xs*,*y*#*ys*) ∈ *listrel r*

lemma *listrel-mono*: *r* ⊆ *s* ⟹ *listrel r* ⊆ *listrel s*

⟨*proof*⟩

lemma *listrel-subset*: $r \subseteq A \times A \Longrightarrow$ *listrel* $r \subseteq$ *lists* $A \times$ *lists* A
⟨*proof*⟩

lemma *listrel-refl*: *refl* A $r \Longrightarrow$ *refl* (*lists* A) (*listrel* r)
⟨*proof*⟩

lemma *listrel-sym*: *sym* $r \Longrightarrow$ *sym* (*listrel* r)
⟨*proof*⟩

lemma *listrel-trans*: *trans* $r \Longrightarrow$ *trans* (*listrel* r)
⟨*proof*⟩

theorem *equiv-listrel*: *equiv* A $r \Longrightarrow$ *equiv* (*lists* A) (*listrel* r)
⟨*proof*⟩

lemma *listrel-Nil* [*simp*]: *listrel* r `` $\{[]\} = \{[]\}$
⟨*proof*⟩

lemma *listrel-Cons*:
listrel r `` $\{x\#xs\} =$ *set-Cons* $(r``\{x\})$ (*listrel* r `` $\{xs\}$)
⟨*proof*⟩

40.5 Miscellany

40.5.1 Characters and strings

datatype *nibble* =
Nibble0 | *Nibble1* | *Nibble2* | *Nibble3* | *Nibble4* | *Nibble5* | *Nibble6* | *Nibble7*
| *Nibble8* | *Nibble9* | *NibbleA* | *NibbleB* | *NibbleC* | *NibbleD* | *NibbleE* | *NibbleF*

lemma *UNIV-nibble*:
UNIV = {*Nibble0*, *Nibble1*, *Nibble2*, *Nibble3*, *Nibble4*, *Nibble5*, *Nibble6*, *Nibble7*, *Nibble8*, *Nibble9*, *NibbleA*, *NibbleB*, *NibbleC*, *NibbleD*, *NibbleE*, *NibbleF*} (**is** - = *?A*)
⟨*proof*⟩

instance *nibble* :: *finite*
⟨*proof*⟩

datatype *char* = *Char nibble nibble*
— Note: canonical order of character encoding coincides with standard term ordering

lemma *UNIV-char*:
UNIV = *image* (*split Char*) (*UNIV* × *UNIV*)
⟨*proof*⟩

instance *char* :: *finite*

⟨*proof*⟩

types *string* = *char list*

syntax
-*Char* :: *xstr* => *char* (*CHR* -)
-*String* :: *xstr* => *string* (-)

⟨*ML*⟩

40.6 Size function

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (*list-size f*)
⟨*proof*⟩

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (*option-size f*)
⟨*proof*⟩

lemma *list-size-estimation*[*termination-simp*]:
x ∈ *set xs* ⟹ *y* < *f x* ⟹ *y* < *list-size f xs*
⟨*proof*⟩

lemma *list-size-estimation'*[*termination-simp*]:
x ∈ *set xs* ⟹ *y* ≤ *f x* ⟹ *y* ≤ *list-size f xs*
⟨*proof*⟩

lemma *list-size-map*[*simp*]: *list-size f* (*map g xs*) = *list-size* (*f o g*) *xs*
⟨*proof*⟩

lemma *list-size-pointwise*[*termination-simp*]:
(⋀*x*. *x* ∈ *set xs* ⟹ *f x* < *g x*) ⟹ *list-size f xs* ≤ *list-size g xs*
⟨*proof*⟩

40.7 Code generator

40.7.1 Setup

types-code
list (- *list*)
attach (*term-of*) ⟪
fun term-of-list f T = *HOLogic.mk-list T o map f*;
⟫
attach (*test*) ⟪
fun gen-list' aG aT i j = *frequency*
[(*i*, *fn* () =>
let
val (*x*, *t*) = *aG j*;
val (*xs*, *ts*) = *gen-list' aG aT* (*i*−*1*) *j*
in (*x* :: *xs*, *fn* () => *HOLogic.cons-const aT* $ *t* () $ *ts* ()) *end*),
(*1*, *fn* () => ([], *fn* () => *HOLogic.nil-const aT*))] ()

```
and gen-list aG aT i = gen-list' aG aT i i;
⟩⟩
  char (string)
attach (term-of) ⟨⟨
val term-of-char = HOLogic.mk-char o ord;
⟩⟩
attach (test) ⟨⟨
fun gen-char i =
  let val j = random-range (ord a) (Int.min (ord a + i, ord z))
  in (chr j, fn () => HOLogic.mk-char j) end;
⟩⟩

consts-code Cons ((- ::/ -))

code-type list
  (SML - list)
  (OCaml - list)
  (Haskell ![-])

code-reserved SML
  list

code-reserved OCaml
  list

code-const Nil
  (SML [])
  (OCaml [])
  (Haskell [])

⟨ML⟩

code-instance list :: eq
  (Haskell -)

code-const op = :: 'a::eq list ⇒ 'a list ⇒ bool
  (Haskell infixl 4 ==)

⟨ML⟩
```

40.7.2 Generation of efficient code

```
primrec
  member :: 'a ⇒ 'a list ⇒ bool (infixl mem 55)
where
  x mem [] ⟷ False
  | x mem (y#ys) ⟷ (if y = x then True else x mem ys)

primrec
```

```
  null:: 'a list ⇒ bool
where
  null [] = True
  | null (x#xs) = False

primrec
  list-inter :: 'a list ⇒ 'a list ⇒ 'a list
where
  list-inter [] bs = []
  | list-inter (a#as) bs =
    (if a ∈ set bs then a # list-inter as bs else list-inter as bs)

primrec
  list-all :: ('a ⇒ bool) ⇒ ('a list ⇒ bool)
where
  list-all P [] = True
  | list-all P (x#xs) = (P x ∧ list-all P xs)

primrec
  list-ex :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
where
  list-ex P [] = False
  | list-ex P (x#xs) = (P x ∨ list-ex P xs)

primrec
  filtermap :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list
where
  filtermap f [] = []
  | filtermap f (x#xs) =
    (case f x of None ⇒ filtermap f xs
     | Some y ⇒ y # filtermap f xs)

primrec
  map-filter :: ('a ⇒ 'b) ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ 'b list
where
  map-filter f P [] = []
  | map-filter f P (x#xs) =
    (if P x then f x # map-filter f P xs else map-filter f P xs)
```

Only use *mem* for generating executable code. Otherwise use $x \in set\ xs$ instead — it is much easier to reason about. The same is true for *list-all* and *list-ex*: write $\forall x{\in}set\ xs$ and $\exists x{\in}set\ xs$ instead because the HOL quantifiers are aleady known to the automatic provers. In fact, the declarations in the code subsection make sure that $\in$, $\forall x{\in}set\ xs$ and $\exists x{\in}set\ xs$ are implemented efficiently.

Efficient emptyness check is implemented by *null*.

The functions *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

lemma *rev-foldl-cons* [*code*]:
rev xs = *foldl* (λ*xs x*. *x* # *xs*) [] *xs*
⟨*proof*⟩

lemma *mem-iff* [*code post*]:
x mem xs ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemmas *in-set-code* [*code unfold*] = *mem-iff* [*symmetric*]

lemma *empty-null* [*code inline*]:
xs = [] ⟷ *null xs*
⟨*proof*⟩

lemmas *null-empty* [*code post*] =
empty-null [*symmetric*]

lemma *list-inter-conv*:
set (*list-inter xs ys*) = *set xs* ∩ *set ys*
⟨*proof*⟩

lemma *list-all-iff* [*code post*]:
list-all P xs ⟷ (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemmas *list-ball-code* [*code unfold*] = *list-all-iff* [*symmetric*]

lemma *list-all-append* [*simp*]:
list-all P (*xs* @ *ys*) ⟷ (*list-all P xs* ∧ *list-all P ys*)
⟨*proof*⟩

lemma *list-all-rev* [*simp*]:
list-all P (*rev xs*) ⟷ *list-all P xs*
⟨*proof*⟩

lemma *list-all-length*:
list-all P xs ⟷ (∀ *n* < *length xs*. *P* (*xs* ! *n*))
⟨*proof*⟩

lemma *list-ex-iff* [*code post*]:
list-ex P xs ⟷ (∃ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemmas *list-bex-code* [*code unfold*] =
list-ex-iff [*symmetric*]

lemma *list-ex-length*:
list-ex P xs ⟷ (∃ *n* < *length xs*. *P* (*xs* ! *n*))
⟨*proof*⟩

lemma *filtermap-conv*:
filtermap f xs = map (λ*x*. *the* (*f x*)) (*filter* (λ*x*. *f x* ≠ *None*) *xs*)
⟨*proof*⟩

lemma *map-filter-conv* [*simp*]:
map-filter f P xs = map f (*filter P xs*)
⟨*proof*⟩

Code for bounded quantification and summation over nats.

lemma *atMost-upto* [*code unfold*]:
{..*n*} = *set* [*0*..<*Suc n*]
⟨*proof*⟩

lemma *atLeast-upt* [*code unfold*]:
{..<*n*} = *set* [*0*..<*n*]
⟨*proof*⟩

lemma *greaterThanLessThan-upt* [*code unfold*]:
{*n*<..<*m*} = *set* [*Suc n*..<*m*]
⟨*proof*⟩

lemma *atLeastLessThan-upt* [*code unfold*]:
{*n*..<*m*} = *set* [*n*..<*m*]
⟨*proof*⟩

lemma *greaterThanAtMost-upto* [*code unfold*]:
{*n*<..*m*} = *set* [*Suc n*..<*Suc m*]
⟨*proof*⟩

lemma *atLeastAtMost-upto* [*code unfold*]:
{*n*..*m*} = *set* [*n*..<*Suc m*]
⟨*proof*⟩

lemma *all-nat-less-eq* [*code unfold*]:
(∀ *m*<*n*::*nat*. *P m*) ⟷ (∀ *m* ∈ {*0*..<*n*}. *P m*)
⟨*proof*⟩

lemma *ex-nat-less-eq* [*code unfold*]:
(∃ *m*<*n*::*nat*. *P m*) ⟷ (∃ *m* ∈ {*0*..<*n*}. *P m*)
⟨*proof*⟩

lemma *all-nat-less* [*code unfold*]:
(∀ *m*≤*n*::*nat*. *P m*) ⟷ (∀ *m* ∈ {*0*..*n*}. *P m*)
⟨*proof*⟩

lemma *ex-nat-less* [*code unfold*]:
(∃ *m*≤*n*::*nat*. *P m*) ⟷ (∃ *m* ∈ {*0*..*n*}. *P m*)
⟨*proof*⟩

lemma *setsum-set-upt-conv-listsum* [*code unfold*]:
setsum f (*set* [*k*..<*n*]) = *listsum* (*map f* [*k*..<*n*])
⟨*proof*⟩

end

41 Map: Maps

theory *Map*
imports *List*
begin

types ('*a*,'*b*) ~=> = '*a* => '*b option* (**infixr** *0*)
translations (*type*) *a* ~=> *b* <= (*type*) *a* => *b option*

syntax (*xsymbols*)
~=> :: [*type*, *type*] => *type* (**infixr** ⇀ *0*)

abbreviation
empty :: '*a* ~=> '*b* **where**
empty == %*x*. *None*

definition
map-comp :: ('*b* ~=> '*c*) => ('*a* ~=> '*b*) => ('*a* ~=> '*c*) (**infixl** *o'-m 55*)
where
f o-m g = (λ*k*. *case g k of None* ⇒ *None* | *Some v* ⇒ *f v*)

notation (*xsymbols*)
map-comp (**infixl** ∘$_m$ *55*)

definition
map-add :: ('*a* ~=> '*b*) => ('*a* ~=> '*b*) => ('*a* ~=> '*b*) (**infixl** ++ *100*)
where
m1 ++ *m2* = (λ*x*. *case m2 x of None* => *m1 x* | *Some y* => *Some y*)

definition
restrict-map :: ('*a* ~=> '*b*) => '*a set* => ('*a* ~=> '*b*) (**infixl** |` *110*) **where**
m|`*A* = (λ*x*. *if x* : *A then m x else None*)

notation (*latex* **output**)
restrict-map (-|`- [*111*,*110*] *110*)

definition
dom :: ('*a* ~=> '*b*) => '*a set* **where**
dom m = {*a*. *m a* ~= *None*}

definition

ran :: ('*a* ~=> '*b*) => '*b set* **where**
ran m = {*b*. *EX a*. *m a* = *Some b*}

definition
map-le :: ('*a* ~=> '*b*) => ('*a* ~=> '*b*) => *bool* (**infix** $\subseteq_m$ *50*) **where**
($m_1 \subseteq_m m_2$) = ($\forall a \in dom\ m_1.\ m_1\ a = m_2\ a$)

consts
map-of :: ('*a* * '*b*) *list* => '*a* ~=> '*b*
map-upds :: ('*a* ~=> '*b*) => '*a list* => '*b list* => ('*a* ~=> '*b*)

nonterminals
maplets maplet

syntax
-maplet :: ['*a*, '*a*] => *maplet* (- /|->/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[|->]/ -)
:: *maplet* => *maplets* (-)
-Maplets :: [*maplet*, *maplets*] => *maplets* (-,/ -)
-MapUpd :: ['*a* ~=> '*b*, *maplets*] => '*a* ~=> '*b* (-/'(-') [*900*,*0*]*900*)
-Map :: *maplets* => '*a* ~=> '*b* ((*1*[-]))

syntax (*xsymbols*)
-maplet :: ['*a*, '*a*] => *maplet* (- /↦/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[↦]/ -)

translations
-MapUpd m (*-Maplets xy ms*) == *-MapUpd* (*-MapUpd m xy*) *ms*
-MapUpd m (*-maplet x y*) == *m*(*x*:=*Some y*)
-MapUpd m (*-maplets x y*) == *map-upds m x y*
-Map ms == *-MapUpd* (*CONST empty*) *ms*
-Map (*-Maplets ms1 ms2*) <= *-MapUpd* (*-Map ms1*) *ms2*
-Maplets ms1 (*-Maplets ms2 ms3*) <= *-Maplets* (*-Maplets ms1 ms2*) *ms3*

primrec
map-of [] = *empty*
map-of (*p*#*ps*) = (*map-of ps*)(*fst p* |-> *snd p*)

declare *map-of.simps* [*code del*]

lemma *map-of-Cons-code* [*code*]:
map-of [] *k* = *None*
map-of ((*l*, *v*) # *ps*) *k* = (*if l* = *k then Some v else map-of ps k*)
⟨*proof*⟩

defs
map-upds-def [*code func*]: *m*(*xs* [|->] *ys*) == *m* ++ *map-of* (*rev*(*zip xs ys*))

41.1 *empty*

lemma *empty-upd-none* [*simp*]: $empty(x := None) = empty$
⟨*proof*⟩

41.2 *map-upd*

lemma *map-upd-triv*: $t\ k = Some\ x \Longrightarrow t(k|{->}x) = t$
⟨*proof*⟩

lemma *map-upd-nonempty* [*simp*]: $t(k|{->}x) \neq empty$
⟨*proof*⟩

lemma *map-upd-eqD1*:
assumes $m(a{\mapsto}x) = n(a{\mapsto}y)$
shows $x = y$
⟨*proof*⟩

lemma *map-upd-Some-unfold*:
$((m(a|{->}b))\ x = Some\ y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = Some\ y)$
⟨*proof*⟩

lemma *image-map-upd* [*simp*]: $x \notin A \Longrightarrow m(x \mapsto y)\ `\ A = m\ `\ A$
⟨*proof*⟩

lemma *finite-range-updI*: $finite\ (range\ f) \Longrightarrow finite\ (range\ (f(a|{->}b)))$
⟨*proof*⟩

41.3 *map-of*

lemma *map-of-eq-None-iff*:
$(map\text{-}of\ xys\ x = None) = (x \notin fst\ `\ (set\ xys))$
⟨*proof*⟩

lemma *map-of-is-SomeD*: $map\text{-}of\ xys\ x = Some\ y \Longrightarrow (x,y) \in set\ xys$
⟨*proof*⟩

lemma *map-of-eq-Some-iff* [*simp*]:
$distinct(map\ fst\ xys) \Longrightarrow (map\text{-}of\ xys\ x = Some\ y) = ((x,y) \in set\ xys)$
⟨*proof*⟩

lemma *Some-eq-map-of-iff* [*simp*]:
$distinct(map\ fst\ xys) \Longrightarrow (Some\ y = map\text{-}of\ xys\ x) = ((x,y) \in set\ xys)$
⟨*proof*⟩

lemma *map-of-is-SomeI* [*simp*]: $[\![\ distinct(map\ fst\ xys);\ (x,y) \in set\ xys\ ]\!]$
$\Longrightarrow map\text{-}of\ xys\ x = Some\ y$
⟨*proof*⟩

lemma *map-of-zip-is-None* [*simp*]:

length xs = length ys ⟹ (map-of (zip xs ys) x = None) = (x ∉ set xs)
⟨*proof*⟩

lemma *map-of-zip-is-Some*:
assumes *length xs = length ys*
shows *x ∈ set xs ⟷ (∃ y. map-of (zip xs ys) x = Some y)*
⟨*proof*⟩

lemma *map-of-zip-upd*:
fixes *x* :: *'a* **and** *xs* :: *'a list* **and** *ys zs* :: *'b list*
assumes *length ys = length xs*
and *length zs = length xs*
and *x ∉ set xs*
and *map-of (zip xs ys)(x ↦ y) = map-of (zip xs zs)(x ↦ z)*
shows *map-of (zip xs ys) = map-of (zip xs zs)*
⟨*proof*⟩

lemma *map-of-zip-inject*:
assumes *length ys = length xs*
and *length zs = length xs*
and *dist*: *distinct xs*
and *map-of*: *map-of (zip xs ys) = map-of (zip xs zs)*
shows *ys = zs*
⟨*proof*⟩

lemma *finite-range-map-of*: *finite (range (map-of xys))*
⟨*proof*⟩

lemma *map-of-SomeD*: *map-of xs k = Some y ⟹ (k, y) ∈ set xs*
⟨*proof*⟩

lemma *map-of-mapk-SomeI*:
inj f ==> map-of t k = Some x ==>
map-of (map (split (%k. Pair (f k))) t) (f k) = Some x
⟨*proof*⟩

lemma *weak-map-of-SomeI*: *(k, x) : set l ==> ∃ x. map-of l k = Some x*
⟨*proof*⟩

lemma *map-of-filter-in*:
map-of xs k = Some z ⟹ P k z ⟹ map-of (filter (split P) xs) k = Some z
⟨*proof*⟩

lemma *map-of-map*: *map-of (map (%(a,b). (a,f b)) xs) x = option-map f (map-of xs x)*
⟨*proof*⟩

41.4 *option-map* related

lemma *option-map-o-empty* [*simp*]: *option-map f o empty = empty*
⟨*proof*⟩

lemma *option-map-o-map-upd* [*simp*]:
option-map f o m(a|->b) = (option-map f o m)(a|->f b)
⟨*proof*⟩

41.5 *map-comp* related

lemma *map-comp-empty* [*simp*]:
m ∘m empty = empty
empty ∘m m = empty
⟨*proof*⟩

lemma *map-comp-simps* [*simp*]:
m2 k = None ⟹ (m1 ∘m m2) k = None
m2 k = Some k' ⟹ (m1 ∘m m2) k = m1 k'
⟨*proof*⟩

lemma *map-comp-Some-iff*:
((m1 ∘m m2) k = Some v) = (∃ k'. m2 k = Some k' ∧ m1 k' = Some v)
⟨*proof*⟩

lemma *map-comp-None-iff*:
((m1 ∘m m2) k = None) = (m2 k = None ∨ (∃ k'. m2 k = Some k' ∧ m1 k' = None))
⟨*proof*⟩

41.6 ++

lemma *map-add-empty*[*simp*]: *m ++ empty = m*
⟨*proof*⟩

lemma *empty-map-add*[*simp*]: *empty ++ m = m*
⟨*proof*⟩

lemma *map-add-assoc*[*simp*]: *m1 ++ (m2 ++ m3) = (m1 ++ m2) ++ m3*
⟨*proof*⟩

lemma *map-add-Some-iff*:
((m ++ n) k = Some x) = (n k = Some x | n k = None & m k = Some x)
⟨*proof*⟩

lemma *map-add-SomeD* [*dest*!]:
(m ++ n) k = Some x ⟹ n k = Some x ∨ n k = None ∧ m k = Some x
⟨*proof*⟩

lemma *map-add-find-right* [*simp*]: !!*xx*. *n k = Some xx ==> (m ++ n) k = Some*

xx
⟨*proof*⟩

lemma *map-add-None* [*iff*]: $((m\ ++\ n)\ k = None) = (n\ k = None\ \&\ m\ k = None)$
⟨*proof*⟩

lemma *map-add-upd*[*simp*]: $f\ ++\ g(x|{->}y) = (f\ ++\ g)(x|{->}y)$
⟨*proof*⟩

lemma *map-add-upds*[*simp*]: $m1\ ++\ (m2(xs[\mapsto]ys)) = (m1{+}{+}m2)(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *map-of-append*[*simp*]: $map\text{-}of\ (xs\ @\ ys) = map\text{-}of\ ys\ ++\ map\text{-}of\ xs$
⟨*proof*⟩

lemma *finite-range-map-of-map-add*:
$finite\ (range\ f) ==> finite\ (range\ (f\ ++\ map\text{-}of\ l))$
⟨*proof*⟩

lemma *inj-on-map-add-dom* [*iff*]:
$inj\text{-}on\ (m\ ++\ m')\ (dom\ m') = inj\text{-}on\ m'\ (dom\ m')$
⟨*proof*⟩

41.7 *restrict-map*

lemma *restrict-map-to-empty* [*simp*]: $m|`\{\} = empty$
⟨*proof*⟩

lemma *restrict-map-empty* [*simp*]: $empty|`D = empty$
⟨*proof*⟩

lemma *restrict-in* [*simp*]: $x \in A \Longrightarrow (m|`A)\ x = m\ x$
⟨*proof*⟩

lemma *restrict-out* [*simp*]: $x \notin A \Longrightarrow (m|`A)\ x = None$
⟨*proof*⟩

lemma *ran-restrictD*: $y \in ran\ (m|`A) \Longrightarrow \exists x{\in}A.\ m\ x = Some\ y$
⟨*proof*⟩

lemma *dom-restrict* [*simp*]: $dom\ (m|`A) = dom\ m \cap A$
⟨*proof*⟩

lemma *restrict-upd-same* [*simp*]: $m(x\mapsto y)|`(-\{x\}) = m|`(-\{x\})$
⟨*proof*⟩

lemma *restrict-restrict* [*simp*]: $m|`A|`B = m|`(A{\cap}B)$
⟨*proof*⟩

lemma *restrict-fun-upd* [*simp*]:
$m(x := y)|`D = (\mathit{if}\ x \in D\ \mathit{then}\ (m|`(D-\{x\}))(x := y)\ \mathit{else}\ m|`D)$
⟨*proof*⟩

lemma *fun-upd-None-restrict* [*simp*]:
$(m|`D)(x := \mathit{None}) = (\mathit{if}\ x{:}D\ \mathit{then}\ m|`(D - \{x\})\ \mathit{else}\ m|`D)$
⟨*proof*⟩

lemma *fun-upd-restrict*: $(m|`D)(x := y) = (m|`(D-\{x\}))(x := y)$
⟨*proof*⟩

lemma *fun-upd-restrict-conv* [*simp*]:
$x \in D \Longrightarrow (m|`D)(x := y) = (m|`(D-\{x\}))(x := y)$
⟨*proof*⟩

41.8 *map-upds*

lemma *map-upds-Nil1* [*simp*]: $m([]\ [|->]\ bs) = m$
⟨*proof*⟩

lemma *map-upds-Nil2* [*simp*]: $m(as\ [|->]\ []) = m$
⟨*proof*⟩

lemma *map-upds-Cons* [*simp*]: $m(a\#as\ [|->]\ b\#bs) = (m(a|->b))(as[|->]bs)$
⟨*proof*⟩

lemma *map-upds-append1* [*simp*]: $\bigwedge ys\ m.\ \mathit{size}\ xs < \mathit{size}\ ys \Longrightarrow$
$m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys)(x \mapsto ys!\mathit{size}\ xs)$
⟨*proof*⟩

lemma *map-upds-list-update2-drop* [*simp*]:
$[\![\mathit{size}\ xs \le i;\ i < \mathit{size}\ ys]\!]$
$\Longrightarrow m(xs[\mapsto]ys[i:=y]) = m(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *map-upd-upds-conv-if*:
$(f(x|->y))(xs\ [|->]\ ys) =$
$(\mathit{if}\ x : \mathit{set}(\mathit{take}\ (\mathit{length}\ ys)\ xs)\ \mathit{then}\ f(xs\ [|->]\ ys)$
$\mathit{else}\ (f(xs\ [|->]\ ys))(x|->y))$
⟨*proof*⟩

lemma *map-upds-twist* [*simp*]:
$a\ ^\sim\!: \mathit{set}\ as ==> m(a|->b)(as[|->]bs) = m(as[|->]bs)(a|->b)$
⟨*proof*⟩

lemma *map-upds-apply-nontin* [*simp*]:
$x\ ^\sim\!: \mathit{set}\ xs ==> (f(xs[|->]ys))\ x = f\ x$
⟨*proof*⟩

lemma *fun-upds-append-drop* [*simp*]:
size xs = *size ys* ⟹ *m*(*xs*@*zs*[↦]*ys*) = *m*(*xs*[↦]*ys*)
⟨*proof*⟩

lemma *fun-upds-append2-drop* [*simp*]:
size xs = *size ys* ⟹ *m*(*xs*[↦]*ys*@*zs*) = *m*(*xs*[↦]*ys*)
⟨*proof*⟩

lemma *restrict-map-upds*[*simp*]:
⟦ *length xs* = *length ys*; *set xs* ⊆ *D* ⟧
⟹ *m*(*xs* [↦] *ys*)|‘*D* = (*m*|‘(*D* − *set xs*))(*xs* [↦] *ys*)
⟨*proof*⟩

41.9 *dom*

lemma *domI*: *m a* = *Some b* ==> *a* : *dom m*
⟨*proof*⟩

lemma *domD*: *a* : *dom m* ==> ∃ *b*. *m a* = *Some b*
⟨*proof*⟩

lemma *domIff* [*iff*, *simp del*]: (*a* : *dom m*) = (*m a* ~= *None*)
⟨*proof*⟩

lemma *dom-empty* [*simp*]: *dom empty* = {}
⟨*proof*⟩

lemma *dom-fun-upd* [*simp*]:
dom(*f*(*x* := *y*)) = (*if y*=*None then dom f* − {*x*} *else insert x* (*dom f*))
⟨*proof*⟩

lemma *dom-map-of*: *dom*(*map-of xys*) = {*x*. ∃ *y*. (*x*,*y*) : *set xys*}
⟨*proof*⟩

lemma *dom-map-of-conv-image-fst*:
dom(*map-of xys*) = *fst* ‘ (*set xys*)
⟨*proof*⟩

lemma *dom-map-of-zip* [*simp*]: [| *length xs* = *length ys*; *distinct xs* |] ==>
dom(*map-of*(*zip xs ys*)) = *set xs*
⟨*proof*⟩

lemma *finite-dom-map-of*: *finite* (*dom* (*map-of l*))
⟨*proof*⟩

lemma *dom-map-upds* [*simp*]:

dom(*m*(*xs*[|−>]*ys*)) = *set*(*take* (*length ys*) *xs*) *Un dom m*
⟨*proof*⟩

lemma *dom-map-add* [*simp*]: *dom*(*m*++*n*) = *dom n Un dom m*
⟨*proof*⟩

lemma *dom-override-on* [*simp*]:
dom(*override-on f g A*) =
(*dom f* − {*a*. *a* : *A* − *dom g*}) *Un* {*a*. *a* : *A Int dom g*}
⟨*proof*⟩

lemma *map-add-comm*: *dom m1* ∩ *dom m2* = {} ⟹ *m1*++*m2* = *m2*++*m1*
⟨*proof*⟩

lemma *finite-map-freshness*:
finite (*dom* (*f* :: 'a ⇀ 'b)) ⟹ ¬ *finite* (*UNIV* :: 'a *set*) ⟹
∃ *x*. *f x* = *None*
⟨*proof*⟩

41.10 *ran*

lemma *ranI*: *m a* = *Some b* ==> *b* : *ran m*
⟨*proof*⟩

lemma *ran-empty* [*simp*]: *ran empty* = {}
⟨*proof*⟩

lemma *ran-map-upd* [*simp*]: *m a* = *None* ==> *ran*(*m*(*a*|−>*b*)) = *insert b* (*ran m*)
⟨*proof*⟩

41.11 *map-le*

lemma *map-le-empty* [*simp*]: *empty* ⊆$_m$ *g*
⟨*proof*⟩

lemma *upd-None-map-le* [*simp*]: *f*(*x* := *None*) ⊆$_m$ *f*
⟨*proof*⟩

lemma *map-le-upd*[*simp*]: *f* ⊆$_m$ *g* ==> *f*(*a* := *b*) ⊆$_m$ *g*(*a* := *b*)
⟨*proof*⟩

lemma *map-le-imp-upd-le* [*simp*]: *m1* ⊆$_m$ *m2* ⟹ *m1*(*x* := *None*) ⊆$_m$ *m2*(*x* ↦ *y*)
⟨*proof*⟩

lemma *map-le-upds* [*simp*]:
f ⊆$_m$ *g* ==> *f*(*as* [|−>] *bs*) ⊆$_m$ *g*(*as* [|−>] *bs*)

⟨*proof*⟩

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \Longrightarrow (dom\ f \subseteq dom\ g)$
⟨*proof*⟩

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
⟨*proof*⟩

lemma *map-le-trans*[*trans*]: $[\![m1 \subseteq_m m2;\ m2 \subseteq_m m3]\!] \Longrightarrow m1 \subseteq_m m3$
⟨*proof*⟩

lemma *map-le-antisym*: $[\![f \subseteq_m g;\ g \subseteq_m f\ ]\!] \Longrightarrow f = g$
⟨*proof*⟩

lemma *map-le-map-add* [*simp*]: $f \subseteq_m (g +\!\!+ f)$
⟨*proof*⟩

lemma *map-le-iff-map-add-commute*: $(f \subseteq_m f +\!\!+ g) = (f +\!\!+ g = g +\!\!+ f)$
⟨*proof*⟩

lemma *map-add-le-mapE*: $f +\!\!+ g \subseteq_m h \Longrightarrow g \subseteq_m h$
⟨*proof*⟩

lemma *map-add-le-mapI*: $[\![f \subseteq_m h;\ g \subseteq_m h;\ f \subseteq_m f +\!\!+ g\ ]\!] \Longrightarrow f +\!\!+ g \subseteq_m h$
⟨*proof*⟩

end

42 Main: Main HOL

theory *Main*
imports *Map*
begin

⟨*ML*⟩

end

References