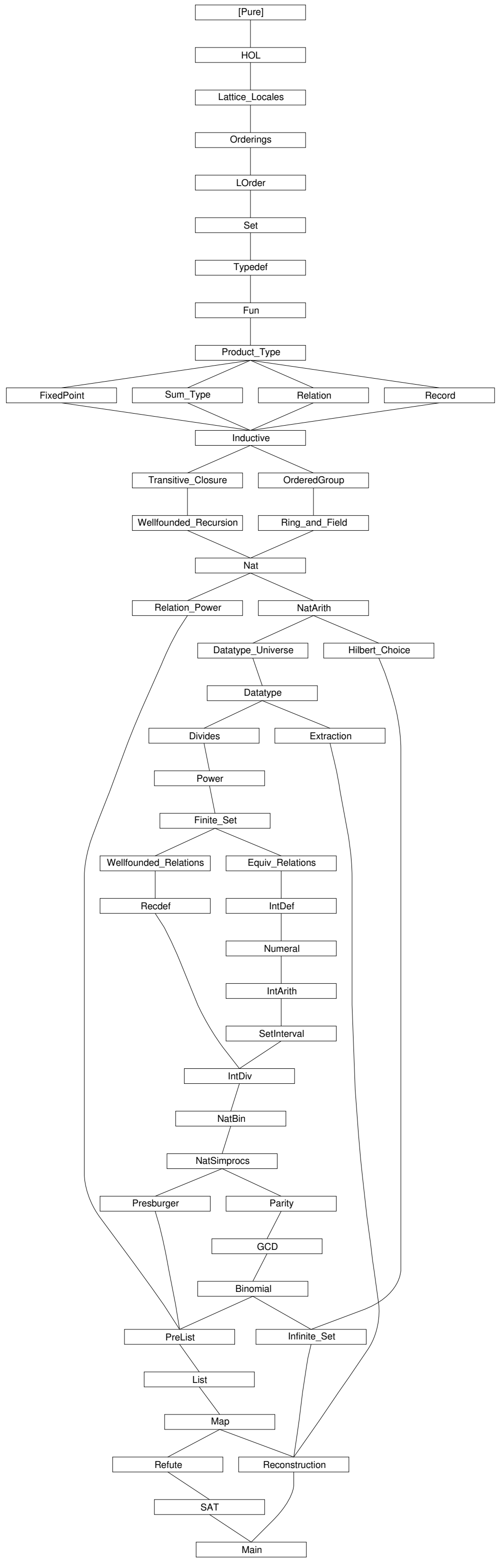
Isabelle/HOL — Higher-Order Logic

October 1, 2005

Contents


[Pure]
HOL
Lattice_Locales
Orderings
LOrder
Set
Typedef
Fun
Product_Type
FixedPoint
Sum_Type
Relation
Record
Inductive
Transitive_Closure
OrderedGroup
Wellfounded_Recursion
Ring_and_Field
Nat
Relation_Power
NatArith
Datatype_Universe
Hilbert_Choice
Datatype
Divides
Extraction
Power
Finite_Set
Wellfounded_Relations
Equiv_Relations
Recdef
IntDef
Numeral
IntArith
SetInterval
IntDiv
NatBin
NatSimprocs
Presburger
Parity
GCD
Binomial
PreList
Infinite_Set
List
Map
Refute
Reconstruction
SAT
Main

1 HOL: The basis of Higher-Order Logic

theory *HOL*
imports *CPure*
uses (*cladata.ML*) (*blastdata.ML*) (*simpdata.ML*) (*eqrule-HOL-data.ML*)
(~~/*src*/*Provers*/*eqsubst.ML*)

begin

1.1 Primitive logic

1.1.1 Core syntax

classes *type*
defaultsort *type*

global

typedecl *bool*

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

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*

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

local

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

1.1.2 Additional concrete syntax

nonterminals

```
letbinds  letbind
case-syn  cases-syn
```

syntax

```
-not-equal  :: ['a, 'a] => bool                  (infixl ~= 50)
-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

```
x ~= y                 == ~ (x = y)
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 (**output**)

```
=           :: ['a, 'a] => bool                  (infix 50)
-not-equal  :: ['a, 'a] => bool                  (infix ~= 50)
```

syntax (*xsymbols*)

```
Not         :: bool => bool                      (¬ - [40] 40)
op &        :: [bool, bool] => bool              (infixr ∧ 35)
op |        :: [bool, bool] => bool              (infixr ∨ 30)
op -->      :: [bool, bool] => bool              (infixr ⟶ 25)
-not-equal  :: ['a, 'a] => bool                  (infix ≠ 50)
ALL         :: [idts, bool] => bool              ((3∀ -./ -) [0, 10] 10)
EX          :: [idts, bool] => bool              ((3∃ -./ -) [0, 10] 10)
EX!         :: [idts, bool] => bool              ((3∃!-./ -) [0, 10] 10)
-case1      :: ['a, 'b] => case-syn              ((2- ⇒/ -) 10)
```

syntax (*xsymbols* **output**)

```
-not-equal  :: ['a, 'a] => bool                  (infix ≠ 50)
```

syntax (*HTML* **output**)

```
-not-equal  :: ['a, 'a] => bool                  (infix ≠ 50)
Not         :: bool => bool                      (¬ - [40] 40)
```

$op\ \&$ :: $[bool, bool] => bool$ (**infixr** $\wedge$ 35)
$op\ |$ :: $[bool, bool] => bool$ (**infixr** $\vee$ 30)
$\text{-}not\text{-}equal$ :: $['a, 'a] => bool$ (**infix** $\neq$ 50)
ALL :: $[idts, bool] => bool$ $((3\forall \text{-./ -}) [0, 10]\ 10)$
EX :: $[idts, bool] => bool$ $((3\exists \text{-./ -}) [0, 10]\ 10)$
$EX!$ :: $[idts, bool] => bool$ $((3\exists!\text{-./ -}) [0, 10]\ 10)$

syntax (*HOL*)
ALL :: $[idts, bool] => bool$ $((3!\ \text{-./ -}) [0, 10]\ 10)$
EX :: $[idts, bool] => bool$ $((3?\ \text{-./ -}) [0, 10]\ 10)$
$EX!$ :: $[idts, bool] => bool$ $((3?!\ \text{-./ -}) [0, 10]\ 10)$

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: $[|\ P{-}{-}{>}Q;\ P\ |] ==> Q$

Thanks to Stephan Merz

theorem *subst*:
assumes *eq*: $s = t$ **and** *p*: $P(s)$
shows $P(t::'a)$
⟨*proof*⟩

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

1.1.4 Generic algebraic operations

axclass *zero < type*
axclass *one < type*
axclass *plus < type*
axclass *minus < type*
axclass *times < type*
axclass *inverse < type*

global

consts
0 :: *'a::zero* (*0*)
1 :: *'a::one* (*1*)
+ :: *['a::plus, 'a] => 'a* (**infixl** *65*)
− :: *['a::minus, 'a] => 'a* (**infixl** *65*)
uminus :: *['a::minus] => 'a* (− - [*81*] *80*)
∗ :: *['a::times, 'a] => 'a* (**infixl** *70*)

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

local

⟨*ML*⟩

consts
abs :: *'a::minus => 'a*
inverse :: *'a::inverse => 'a*
divide :: *['a::inverse, 'a] => 'a* (**infixl** '/ *70*)

syntax (*xsymbols*)
abs :: *'a::minus => 'a* (|-|)
syntax (*HTML* **output**)
abs :: *'a::minus => 'a* (|-|)

1.2 Equality

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

lemmas *ssubst* = *sym* [*THEN subst*, *standard*]

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

lemma *def-imp-eq*: **assumes** *meq*: A == B **shows** $A = B$
⟨*proof*⟩

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

For calculational reasoning:

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

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

1.3 Congruence rules for application

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

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

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

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

1.4 Equality of booleans – iff

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

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

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

⟨*proof*⟩

lemmas *iffD1* = *sym* [*THEN iffD2*, *standard*]
lemmas *rev-iffD1* = *sym* [*THEN* [*2*] *rev-iffD2*, *standard*]

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

1.5 True

lemma *TrueI*: *True*
⟨*proof*⟩

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

lemma *eqTrueE*: *P*=*True* ==> *P*
⟨*proof*⟩

1.6 Universal quantifier

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

lemma *spec*: *ALL x*::'*a*. *P*(*x*) ==> *P*(*x*)
⟨*proof*⟩

lemma *allE*:
  assumes *major*: *ALL x*. *P*(*x*)
    and *minor*: *P*(*x*) ==> *R*
  shows *R*
⟨*proof*⟩

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

1.7 False

lemma *FalseE*: *False* ==> *P*
⟨*proof*⟩

lemma *False-neq-True*: *False*=*True* ==> *P*
⟨*proof*⟩

1.8 Negation

lemma *notI*:
 assumes *p*: *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*⟩

lemmas *notI2* = *notE* [*THEN notI*, *standard*]

1.9 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 *rev-contrapos*:
 assumes *pq*: *P* ==> *Q*
  and *nq*: ~*Q*

shows $\sim P$
⟨*proof*⟩

1.10 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*⟩

1.11 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 *prems*: P P ==> Q **shows** P & Q
⟨*proof*⟩

1.12 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*⟩

1.13 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*⟩

1.14 Unique existence

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

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

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

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

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

1.15 THE: definite description operator

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

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

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

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

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

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

1.16 Classical intro rules for disjunction and existential quantifiers

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

lemma *excluded-middle*: $\tilde{}P\ |\ P$
⟨*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*: $P ==> Q$
    and *prem2*: $\tilde{}P ==> Q$
  shows Q
⟨*proof*⟩

lemma *impCE*:
  assumes *major*: $P-->Q$
    and *minor*: $\tilde{}P ==> R\ Q ==> R$

shows *R*
⟨*proof*⟩

lemma *impCE′*:
assumes *major*: *P-->Q*
and *minor*: *Q ==> R ~P ==> R*
shows *R*
⟨*proof*⟩

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

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

1.17 Theory and package setup

⟨*ML*⟩

theorems *case-split* = *case-split-thm* [*case-names True False*]

1.17.1 Intuitionistic Reasoning

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

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

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

lemmas [*Pure.elim!*] = *disjE iffE FalseE conjE exE*

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*

1.17.2 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*]:
(!!C. (A ==> B ==> *PROP* C) ==> *PROP* C) == *Trueprop* (A & B)
⟨*proof*⟩

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

1.17.3 Classical Reasoner setup

⟨*ML*⟩

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

⟨*ML*⟩

1.17.4 Simplifier setup

lemma *meta-eq-to-obj-eq*: x == y ==> x = y
⟨*proof*⟩

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*

$(x = x) = True$
$(\sim True) = False$ $(\sim False) = True$
$(\sim P) \sim= P$ $P \sim= (\sim P)$
$(True{=}P) = P$ $(P{=}True) = P$ $(False{=}P) = (\sim P)$ $(P{=}False) = (\sim P)$
$(True \dashrightarrow P) = P$ $(False \dashrightarrow P) = True$
$(P \dashrightarrow True) = True$ $(P \dashrightarrow P) = True$
$(P \dashrightarrow False) = (\sim P)$ $(P \dashrightarrow \sim P) = (\sim P)$
$(P \ \& \ True) = P$ $(True \ \& \ P) = P$
$(P \ \& \ False) = False$ $(False \ \& \ P) = False$
$(P \ \& \ P) = P$ $(P \ \& \ (P \ \& \ Q)) = (P \ \& \ Q)$
$(P \ \& \ \sim P) = False$ $(\sim P \ \& \ P) = False$
$(P \mid True) = True$ $(True \mid P) = True$
$(P \mid False) = P$ $(False \mid P) = P$
$(P \mid P) = P$ $(P \mid (P \mid Q)) = (P \mid Q)$ **and**
$(ALL\ x.\ P) = P$ $(EX\ x.\ P) = P$ $EX\ x.\ x{=}t$ $EX\ x.\ t{=}x$
— needed for the one-point-rule quantifier simplification procs
— essential for termination!! **and**
$!!P.\ (EX\ x.\ x{=}t \ \&\ P(x)) = P(t)$
$!!P.\ (EX\ x.\ t{=}x \ \&\ P(x)) = P(t)$
$!!P.\ (ALL\ x.\ x{=}t \dashrightarrow P(x)) = P(t)$
$!!P.\ (ALL\ x.\ t{=}x \dashrightarrow P(x)) = P(t)$
⟨*proof*⟩

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

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

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

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*⟩

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

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

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

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

lemma *de-Morgan-disj*: $(\tilde{}(P \mid Q)) = (\tilde{}P \ \& \ \tilde{}Q)$ ⟨*proof*⟩
lemma *de-Morgan-conj*: $(\tilde{}(P \ \& \ Q)) = (\tilde{}P \mid \tilde{}Q)$ ⟨*proof*⟩
lemma *not-imp*: $(\tilde{}(P \ {-}{-}{>} \ Q)) = (P \ \& \ \tilde{}Q)$ ⟨*proof*⟩
lemma *not-iff*: $(P\tilde{}{=}Q) = (P = (\tilde{}Q))$ ⟨*proof*⟩
lemma *disj-not1*: $(\tilde{}P \mid Q) = (P \ {-}{-}{>} \ Q)$ ⟨*proof*⟩
lemma *disj-not2*: $(P \mid \tilde{}Q) = (Q \ {-}{-}{>} \ P)$ — 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 *ex-disj-distrib*: $(?\ x.\ P(x) \mid Q(x)) = ((?\ x.\ P(x)) \mid (?\ x.\ Q(x)))$ ⟨*proof*⟩
lemma *all-conj-distrib*: $(!x.\ P(x)\ \&\ Q(x)) = ((!\ x.\ P(x))\ \&\ (!\ x.\ Q(x)))$ ⟨*proof*⟩

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

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

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

The | congruence rule: not included by default!

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

lemma *eq-sym-conv*: $(x = y) = (y = x)$
⟨*proof*⟩

if-then-else rules

lemma *if-True*: $(\mathit{if}\ \mathit{True}\ \mathit{then}\ x\ \mathit{else}\ y) = x$
⟨*proof*⟩

lemma *if-False*: $(\mathit{if}\ \mathit{False}\ \mathit{then}\ x\ \mathit{else}\ y) = y$
⟨*proof*⟩

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

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

⟨*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* = *split-if split-if-asm*

lemma *if-def2*: (*if* Q *then* x *else* y) = ((Q --> x) & (~ Q --> y))
⟨*proof*⟩

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*: $f\ x \equiv g\ x \Longrightarrow$ *Let* x f $\equiv$ *Let* x g
⟨*proof*⟩

lemma *Let-unfold*: $f\ x \equiv g \Longrightarrow$ *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*)
simp-implies $\equiv$ *op* ==>

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

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

lemma *simp-implies-cong*:
  assumes PP' :*PROP P == PROP* P'
  and $P'QQ'$: *PROP* P' ==> (*PROP Q == PROP* Q')
  shows (*PROP P =simp=> PROP Q*) == (*PROP* P' =*simp*=> *PROP* Q')
⟨*proof*⟩

Actual Installation of the Simplifier.

⟨*ML*⟩

Lucas Dixon's eqstep tactic.

⟨*ML*⟩

1.17.5 Code generator setup

types-code
  bool (*bool*)
attach (*term-of*) ⟪
fun term-of-bool b = *if b then HOLogic.true-const else HOLogic.false-const*;
⟫
attach (*test*) ⟪
fun gen-bool i = *one-of* [*false*, *true*];
⟫

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

⟨*ML*⟩

1.18 Other simple lemmas

declare *disj-absorb* [*simp*] *conj-absorb* [*simp*]

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

theorem *choice-eq*: (*ALL x*. *EX*! *y*. *P x y*) = (*EX*! *f*. *ALL x*. *P x* (*f x*))

⟨*proof*⟩

Needs only HOL-lemmas:

lemma *mk-left-commute*:
assumes a: $\bigwedge x\ y\ z.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$ **and**
c: $\bigwedge x\ y.\ f\ x\ y = f\ y\ x$
shows $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$
⟨*proof*⟩

1.19 Generic cases and induction

constdefs
induct-forall :: ($'a$ => *bool*) => *bool*
induct-forall P == $\forall x.\ P\ x$
induct-implies :: *bool* => *bool* => *bool*
induct-implies $A\ B$ == A −−> B
induct-equal :: $'a$ => $'a$ => *bool*
induct-equal $x\ y$ == $x = y$
induct-conj :: *bool* => *bool* => *bool*
induct-conj $A\ B$ == A & B

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

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

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

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

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

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

lemma *induct-impliesI*: (A ==> B) ==> *induct-implies* $A\ B$
⟨*proof*⟩

lemmas *induct-atomize* = *atomize-conj induct-forall-eq induct-implies-eq induct-equal-eq*
lemmas *induct-rulify1* [*symmetric*, *standard*] = *induct-forall-eq induct-implies-eq induct-equal-eq*
lemmas *induct-rulify2* = *induct-forall-def induct-implies-def induct-equal-def induct-conj-def*

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*⟩

1.19.1 Tags, for the ATP Linkup

constdefs
  tag :: *bool* => *bool*
   tag P == *P*

These label the distinguished literals of introduction and elimination rules.

lemma *tagI*: *P* ==> *tag P*
⟨*proof*⟩

lemma *tagD*: *tag P* ==> *P*
⟨*proof*⟩

Applications of "tag" to True and False must go!

lemma *tag-True*: *tag True* = *True*
⟨*proof*⟩

lemma *tag-False*: *tag False* = *False*
⟨*proof*⟩

end

2 Lattice-Locales: Lattices via Locales

theory *Lattice-Locales*
imports *HOL*
begin

2.1 Lattices

This theory of lattice locales only defines binary sup and inf operations. The extension to finite sets is done in theory *Finite-Set*. In the longer term it may be better to define arbitrary sups and infs via *THE*.

locale *partial-order* =
  fixes *below* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sqsubseteq$ *50*)
  assumes *refl*[*iff*]: $x \sqsubseteq x$
  and *trans*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
  and *antisym*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq x \Longrightarrow x = y$

locale *lower-semilattice* = *partial-order* +
fixes *inf* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcap$ *70*)
assumes *inf-le1*: $x \sqcap y \sqsubseteq x$ **and** *inf-le2*: $x \sqcap y \sqsubseteq y$
and *inf-least*: $x \sqsubseteq y \Longrightarrow x \sqsubseteq z \Longrightarrow x \sqsubseteq y \sqcap z$

locale *upper-semilattice* = *partial-order* +
fixes *sup* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcup$ *65*)
assumes *sup-ge1*: $x \sqsubseteq x \sqcup y$ **and** *sup-ge2*: $y \sqsubseteq x \sqcup y$
and *sup-greatest*: $y \sqsubseteq x \Longrightarrow z \sqsubseteq x \Longrightarrow y \sqcup z \sqsubseteq x$

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

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

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

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

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

lemma (**in** *lower-semilattice*) *inf-idem[simp]*: $x \sqcap x = x$
⟨*proof*⟩

lemma (**in** *upper-semilattice*) *sup-idem[simp]*: $x \sqcup x = x$
⟨*proof*⟩

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

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

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

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

lemma (**in** *lower-semilattice*) *inf-absorb*: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
⟨*proof*⟩

lemma (**in** *upper-semilattice*) *sup-absorb*: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
⟨*proof*⟩

lemma (**in** *lower-semilattice*) *below-inf-conv*[*simp*]:
$x \sqsubseteq y \sqcap z = (x \sqsubseteq y \wedge x \sqsubseteq z)$
$\langle proof \rangle$

lemma (**in** *upper-semilattice*) *above-sup-conv*[*simp*]:
$x \sqcup y \sqsubseteq z = (x \sqsubseteq z \wedge y \sqsubseteq z)$
$\langle proof \rangle$

Towards distributivity: if you have one of them, you have them all.

lemma (**in** *lattice*) *distrib-imp1*:
assumes D: !!$x\ y\ z.\ x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
shows $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
$\langle proof \rangle$

lemma (**in** *lattice*) *distrib-imp2*:
assumes D: !!$x\ y\ z.\ x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
$\langle proof \rangle$

A package of rewrite rules for deciding equivalence wrt ACI:

lemma (**in** *lower-semilattice*) *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
$\langle proof \rangle$

lemma (**in** *upper-semilattice*) *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
$\langle proof \rangle$

lemma (**in** *lower-semilattice*) *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$
$\langle proof \rangle$

lemma (**in** *upper-semilattice*) *sup-left-idem*: $x \sqcup (x \sqcup y) = x \sqcup y$
$\langle proof \rangle$

lemmas (**in** *lower-semilattice*) *inf-ACI* =
inf-commute inf-assoc inf-left-commute inf-left-idem

lemmas (**in** *upper-semilattice*) *sup-ACI* =
sup-commute sup-assoc sup-left-commute sup-left-idem

lemmas (**in** *lattice*) *ACI* = *inf-ACI sup-ACI*

2.2 Distributive lattices

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

lemma (**in** *distrib-lattice*) *sup-inf-distrib2*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$

⟨*proof*⟩

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

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

lemmas (**in** *distrib-lattice*) *distrib* =
sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2

end

3 Orderings: Type classes for ≤

theory *Orderings*
imports *Lattice-Locales*
uses (*antisym-setup.ML*)
begin

3.1 Order signatures and orders

axclass
ord < *type*

syntax
op < :: ['*a*::*ord*, '*a*] => *bool* (*op* <)
op <= :: ['*a*::*ord*, '*a*] => *bool* (*op* <=)

global

consts
op < :: ['*a*::*ord*, '*a*] => *bool* ((-/ < -) [*50*, *51*] *50*)
op <= :: ['*a*::*ord*, '*a*] => *bool* ((-/ <= -) [*50*, *51*] *50*)

local

syntax (*xsymbols*)
op <= :: ['*a*::*ord*, '*a*] => *bool* (*op* ≤)
op <= :: ['*a*::*ord*, '*a*] => *bool* ((-/ ≤ -) [*50*, *51*] *50*)

syntax (*HTML* **output**)
op <= :: ['*a*::*ord*, '*a*] => *bool* (*op* ≤)
op <= :: ['*a*::*ord*, '*a*] => *bool* ((-/ ≤ -) [*50*, *51*] *50*)

Syntactic sugar:

syntax
-gt :: *'a::ord* => *'a* => *bool* (**infixl** > *50*)
-ge :: *'a::ord* => *'a* => *bool* (**infixl** >= *50*)
translations
x > *y* => *y* < *x*
x >= *y* => *y* <= *x*

syntax (*xsymbols*)
-ge :: *'a::ord* => *'a* => *bool* (**infixl** ≥ *50*)

syntax (*HTML* **output**)
-ge :: [*'a::ord*, *'a*] => *bool* (**infixl** ≥ *50*)

3.2 Monotonicity

locale *mono* =
fixes *f*
assumes *mono*: *A* <= *B* ==> *f A* <= *f B*

lemmas *monoI* [*intro?*] = *mono.intro*
and *monoD* [*dest?*] = *mono.mono*

constdefs
min :: [*'a::ord*, *'a*] => *'a*
min a b == (*if a* <= *b then a else b*)
max :: [*'a::ord*, *'a*] => *'a*
max a b == (*if a* <= *b then b else a*)

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

lemma *min-of-mono*:
ALL x y. (*f x* <= *f y*) = (*x* <= *y*) ==> *min* (*f m*) (*f n*) = *f* (*min m n*)
⟨*proof*⟩

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

lemma *max-of-mono*:
ALL x y. (*f x* <= *f y*) = (*x* <= *y*) ==> *max* (*f m*) (*f n*) = *f* (*max m n*)
⟨*proof*⟩

3.3 Orders

axclass *order* < *ord*
order-refl [*iff*]: *x* <= *x*
order-trans: *x* <= *y* ==> *y* <= *z* ==> *x* <= *z*
order-antisym: *x* <= *y* ==> *y* <= *x* ==> *x* = *y*

order-less-le: $(x < y) = (x <= y \ \& \ x \ \tilde{}{=}\ y)$

Connection to locale:

interpretation *order*:
partial-order$[op \leq :: 'a::order \Rightarrow 'a \Rightarrow bool]$
⟨*proof*⟩

Reflexivity.

lemma *order-eq-refl*: $!!x::'a::order.\ x = y ==> x <= y$
— This form is useful with the classical reasoner.
⟨*proof*⟩

lemma *order-less-irrefl* [*iff*]: $\tilde{}\ x < (x::'a::order)$
⟨*proof*⟩

lemma *order-le-less*: $((x::'a::order) <= y) = (x < y \mid x = y)$
— NOT suitable for iff, since it can cause PROOF FAILED.
⟨*proof*⟩

lemmas *order-le-imp-less-or-eq* = *order-le-less* [*THEN iffD1*, *standard*]

lemma *order-less-imp-le*: $!!x::'a::order.\ x < y ==> x <= y$
⟨*proof*⟩

Asymmetry.

lemma *order-less-not-sym*: $(x::'a::order) < y ==> \tilde{}\ (y < x)$
⟨*proof*⟩

lemma *order-less-asym*: $x < (y::'a::order) ==> (\tilde{}P ==> y < x) ==> P$
⟨*proof*⟩

lemma *order-eq-iff*: $!!x::'a::order.\ (x = y) = (x \leq y \ \& \ y \leq x)$
⟨*proof*⟩

lemma *order-antisym-conv*: $(y::'a::order) <= x ==> (x <= y) = (x = y)$
⟨*proof*⟩

Transitivity.

lemma *order-less-trans*: $!!x::'a::order.\ [|\ x < y;\ y < z\ |] ==> x < z$
⟨*proof*⟩

lemma *order-le-less-trans*: $!!x::'a::order.\ [|\ x <= y;\ y < z\ |] ==> x < z$
⟨*proof*⟩

lemma *order-less-le-trans*: $!!x::'a::order.\ [|\ x < y;\ y <= z\ |] ==> x < z$
⟨*proof*⟩

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

lemma *order-less-imp-not-less*: $(x::'a::order) < y ==> (\sim y < x) = True$
⟨*proof*⟩

lemma *order-less-imp-triv*: $(x::'a::order) < y ==> (y < x --> P) = True$
⟨*proof*⟩

lemma *order-less-imp-not-eq*: $(x::'a::order) < y ==> (x = y) = False$
⟨*proof*⟩

lemma *order-less-imp-not-eq2*: $(x::'a::order) < y ==> (y = x) = False$
⟨*proof*⟩

Other operators.

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*⟩

3.4 Transitivity rules for calculational reasoning

lemma *order-neq-le-trans*: $a \sim= b ==> (a::'a::order) <= b ==> a < b$
⟨*proof*⟩

lemma *order-le-neq-trans*: $(a::'a::order) <= b ==> a \sim= b ==> a < b$
⟨*proof*⟩

lemma *order-less-asym'*: $(a::'a::order) < b ==> b < a ==> P$
⟨*proof*⟩

3.5 Least value operator

constdefs
$Least :: ('a::ord => bool) => 'a$ (**binder** $LEAST\ \ 10$)
$Least\ P == THE\ x.\ P\ x\ \&\ (ALL\ y.\ P\ y --> x <= y)$
— We can no longer use LeastM because the latter requires Hilbert-AC.

lemma *LeastI2-order*:
$[|\ P\ (x::'a::order);$
$!!y.\ P\ y ==> x <= y;$
$!!x.\ [|\ P\ x;\ ALL\ y.\ P\ y --> x \leq y\ |] ==> Q\ x\ |]$
$==> Q\ (Least\ P)$
⟨*proof*⟩

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

3.6 Linear / total orders

axclass $linorder < order$
$linorder\text{-}linear$: $x <= y \mid y <= x$

lemma $linorder\text{-}less\text{-}linear$: $!!x::'a::linorder.\ x<y \mid x=y \mid y<x$
⟨*proof*⟩

lemma $linorder\text{-}le\text{-}less\text{-}linear$: $!!x::'a::linorder.\ x\leq y \mid y<x$
⟨*proof*⟩

lemma $linorder\text{-}le\text{-}cases$ [$case\text{-}names\ le\ ge$]:
$((x::'a::linorder) \leq y ==> P) ==> (y \leq x ==> P) ==> P$
⟨*proof*⟩

lemma $linorder\text{-}cases$ [$case\text{-}names\ less\ equal\ greater$]:
$((x::'a::linorder) < y ==> P) ==> (x = y ==> P) ==> (y < x ==> P) ==> P$
⟨*proof*⟩

lemma $linorder\text{-}not\text{-}less$: $!!x::'a::linorder.\ (\sim x < y) = (y <= x)$
⟨*proof*⟩

lemma $linorder\text{-}not\text{-}le$: $!!x::'a::linorder.\ (\sim x <= y) = (y < x)$
⟨*proof*⟩

lemma $linorder\text{-}neq\text{-}iff$: $!!x::'a::linorder.\ (x \sim= y) = (x<y \mid y<x)$
⟨*proof*⟩

lemma $linorder\text{-}neqE$: $x \sim= (y::'a::linorder) ==> (x < y ==> R) ==> (y < x ==> R) ==> R$
⟨*proof*⟩

lemma $linorder\text{-}antisym\text{-}conv1$: $\sim (x::'a::linorder) < y ==> (x <= y) = (x = y)$
⟨*proof*⟩

lemma $linorder\text{-}antisym\text{-}conv2$: $(x::'a::linorder) <= y ==> (\sim x < y) = (x = y)$
⟨*proof*⟩

lemma $linorder\text{-}antisym\text{-}conv3$: $\sim (y::'a::linorder) < x ==> (\sim x < y) = (x = y)$
⟨*proof*⟩

Replacing the old Nat.leI

lemma leI: $\sim x < y ==> y <= (x::'a::linorder)$
⟨*proof*⟩

lemma leD: $y <= (x::'a::linorder) ==> \sim x < y$
⟨*proof*⟩

lemma *not-leE*: $\tilde{}\ y <= (x::'a::linorder) ==> x < y$
⟨*proof*⟩

⟨*ML*⟩

3.7 Setup of transitivity reasoner as Solver

lemma *less-imp-neq*: $[|\ (x::'a::order) < y\ |] ==> x\ \tilde{}= y$
⟨*proof*⟩

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

⟨*ML*⟩

3.8 Min and max on (linear) orders

Instantiate locales:

interpretation *min-max*:
lower-semilattice$[op \leq min :: 'a::linorder \Rightarrow 'a \Rightarrow 'a]$
⟨*proof*⟩

interpretation *min-max*:
upper-semilattice$[op \leq max :: 'a::linorder \Rightarrow 'a \Rightarrow 'a]$
⟨*proof*⟩

interpretation *min-max*:
lattice$[op \leq min :: 'a::linorder \Rightarrow 'a \Rightarrow 'a\ max]$
⟨*proof*⟩

interpretation *min-max*:
distrib-lattice$[op \leq min :: 'a::linorder \Rightarrow 'a \Rightarrow 'a\ max]$
⟨*proof*⟩

lemma *le-max-iff-disj*: $!!z::'a::linorder.\ (z <= max\ x\ y) = (z <= x\ |\ z <= y)$
⟨*proof*⟩

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

lemma *less-max-iff-disj*: $!!z::'a::linorder.\ (z < max\ x\ y) = (z < x\ |\ z < y)$
⟨*proof*⟩

lemma *max-less-iff-conj* [*simp*]:
$!!z::'a::linorder.\ (max\ x\ y < z) = (x < z\ \&\ y < z)$
⟨*proof*⟩

lemma *min-less-iff-conj* [*simp*]:

!!z::'a::linorder. (z < min x y) = (z < x & z < y)
⟨proof⟩

lemma min-le-iff-disj: !!z::'a::linorder. (min x y <= z) = (x <= z | y <= z)
⟨proof⟩

lemma min-less-iff-disj: !!z::'a::linorder. (min x y < z) = (x < z | y < z)
⟨proof⟩

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]

lemma split-min:
P (min (i::'a::linorder) j) = ((i <= j --> P(i)) & (~ i <= j --> P(j)))
⟨proof⟩

lemma split-max:
P (max (i::'a::linorder) j) = ((i <= j --> P(j)) & (~ i <= j --> P(i)))
⟨proof⟩

3.9 Bounded quantifiers

syntax
-lessAll :: [idt, 'a, bool] => bool ((3ALL -<-./ -) [0, 0, 10] 10)
-lessEx :: [idt, 'a, bool] => bool ((3EX -<-./ -) [0, 0, 10] 10)
-leAll :: [idt, 'a, bool] => bool ((3ALL -<=-./ -) [0, 0, 10] 10)
-leEx :: [idt, 'a, bool] => bool ((3EX -<=-./ -) [0, 0, 10] 10)

-gtAll :: [idt, 'a, bool] => bool ((3ALL ->-./ -) [0, 0, 10] 10)
-gtEx :: [idt, 'a, bool] => bool ((3EX ->-./ -) [0, 0, 10] 10)
-geAll :: [idt, 'a, bool] => bool ((3ALL ->=-./ -) [0, 0, 10] 10)
-geEx :: [idt, 'a, bool] => bool ((3EX ->=-./ -) [0, 0, 10] 10)

syntax (xsymbols)
-lessAll :: [idt, 'a, bool] => bool ((3∀ -<-./ -) [0, 0, 10] 10)
-lessEx :: [idt, 'a, bool] => bool ((3∃ -<-./ -) [0, 0, 10] 10)
-leAll :: [idt, 'a, bool] => bool ((3∀ -≤-./ -) [0, 0, 10] 10)
-leEx :: [idt, 'a, bool] => bool ((3∃ -≤-./ -) [0, 0, 10] 10)

-gtAll :: [idt, 'a, bool] => bool ((3∀ ->-./ -) [0, 0, 10] 10)
-gtEx :: [idt, 'a, bool] => bool ((3∃ ->-./ -) [0, 0, 10] 10)
-geAll :: [idt, 'a, bool] => bool ((3∀ -≥-./ -) [0, 0, 10] 10)
-geEx :: [idt, 'a, bool] => bool ((3∃ -≥-./ -) [0, 0, 10] 10)

syntax (HOL)
-lessAll :: [idt, 'a, bool] => bool ((3! -<-./ -) [0, 0, 10] 10)

-lessEx :: [idt, 'a, bool] => bool ((3? -<-./ -) [0, 0, 10] 10)
-leAll :: [idt, 'a, bool] => bool ((3! -<=-./ -) [0, 0, 10] 10)
-leEx :: [idt, 'a, bool] => bool ((3? -<=-./ -) [0, 0, 10] 10)

syntax (*HTML* **output**)
-lessAll :: [idt, 'a, bool] => bool ((3∀ -<-./ -) [0, 0, 10] 10)
-lessEx :: [idt, 'a, bool] => bool ((3∃ -<-./ -) [0, 0, 10] 10)
-leAll :: [idt, 'a, bool] => bool ((3∀ -≤-./ -) [0, 0, 10] 10)
-leEx :: [idt, 'a, bool] => bool ((3∃ -≤-./ -) [0, 0, 10] 10)

-gtAll :: [idt, 'a, bool] => bool ((3∀ ->-./ -) [0, 0, 10] 10)
-gtEx :: [idt, 'a, bool] => bool ((3∃ ->-./ -) [0, 0, 10] 10)
-geAll :: [idt, 'a, bool] => bool ((3∀ -≥-./ -) [0, 0, 10] 10)
-geEx :: [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⟩

3.10 Extra transitivity rules

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

lemma *xt1*: a = b ==> b > c ==> a > c
⟨proof⟩

lemma *xt2*: a > b ==> b = c ==> a > c
⟨proof⟩

lemma *xt3*: a = b ==> b >= c ==> a >= c
⟨proof⟩

lemma *xt4*: a >= b ==> b = c ==> a >= c
⟨proof⟩

lemma *xt5*: (x::'a::order) >= y ==> y >= x ==> x = y
⟨proof⟩

lemma *xt6*: (x::'a::order) >= y ==> y >= z ==> x >= z
⟨proof⟩

lemma *xt7*: (x::'a::order) > y ==> y >= z ==> x > z
⟨*proof*⟩

lemma *xt8*: (x::'a::order) >= y ==> y > z ==> x > z
⟨*proof*⟩

lemma *xt9*: (a::'a::order) > b ==> b > a ==> ?P
⟨*proof*⟩

lemma *xt10*: (x::'a::order) > y ==> y > z ==> x > z
⟨*proof*⟩

lemma *xt11*: (a::'a::order) >= b ==> a ~= b ==> a > b
⟨*proof*⟩

lemma *xt12*: (a::'a::order) ~= b ==> a >= b ==> a > b
⟨*proof*⟩

lemma *xt13*: a = f b ==> b > c ==> (!!x y. x > y ==> f x > f y) ==>
a > f c
⟨*proof*⟩

lemma *xt14*: a > b ==> f b = c ==> (!!x y. x > y ==> f x > f y) ==>
f a > c
⟨*proof*⟩

lemma *xt15*: a = f b ==> b >= c ==> (!!x y. x >= y ==> f x >= f y) ==>
a >= f c
⟨*proof*⟩

lemma *xt16*: a >= b ==> f b = c ==> (!! x y. x >= y ==> f x >= f y) ==>
f a >= c
⟨*proof*⟩

lemma *xt17*: (a::'a::order) >= f b ==> b >= c ==>
(!!x y. x >= y ==> f x >= f y) ==> a >= f c
⟨*proof*⟩

lemma *xt18*: (a::'a::order) >= b ==> (f b::'b::order) >= c ==>
(!!x y. x >= y ==> f x >= f y) ==> f a >= c
⟨*proof*⟩

lemma *xt19*: (a::'a::order) > f b ==> (b::'b::order) >= c ==>
(!!x y. x >= y ==> f x >= f y) ==> a > f c
⟨*proof*⟩

lemma *xt20*: (a::'a::order) > b ==> (f b::'b::order) >= c==>
(!!x y. x > y ==> f x > f y) ==> f a > c
⟨*proof*⟩

lemma *xt21*: $(a::'a::order) >= f\ b ==> b > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt22*: $(a::'a::order) >= b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> f\ a > c$
⟨*proof*⟩

lemma *xt23*: $(a::'a::order) > f\ b ==> (b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt24*: $(a::'a::order) > b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> f\ a > c$
⟨*proof*⟩

lemmas *xtrans* = *xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9 xt10 xt11 xt12 xt13 xt14 xt15 xt15 xt17 xt18 xt19 xt20 xt21 xt22 xt23 xt24*

end

4 LOrder: Lattice Orders

theory *LOrder*
imports *Orderings*
begin

The theory of lattices developed here is taken from the book:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979.

constdefs
is-meet $:: (('a::order) \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
is-meet $m == !\ a\ b\ x.\ m\ a\ b \le a \wedge m\ a\ b \le b \wedge (x \le a \wedge x \le b \longrightarrow x \le m\ a\ b)$
is-join $:: (('a::order) \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
is-join $j == !\ a\ b\ x.\ a \le j\ a\ b \wedge b \le j\ a\ b \wedge (a \le x \wedge b \le x \longrightarrow j\ a\ b \le x)$

lemma *is-meet-unique*:
assumes *is-meet u is-meet v* **shows** $u = v$
⟨*proof*⟩

lemma *is-join-unique*:

assumes *is-join u is-join v* **shows** $u = v$
⟨*proof*⟩

axclass *join-semilorder* < *order*
join-exists: *? j. is-join j*

axclass *meet-semilorder* < *order*
meet-exists: *? m. is-meet m*

axclass *lorder* < *join-semilorder*, *meet-semilorder*

constdefs
meet :: ('*a*::*meet-semilorder*) ⇒ '*a* ⇒ '*a*
meet == *THE m. is-meet m*
join :: ('*a*::*join-semilorder*) ⇒ '*a* ⇒ '*a*
join == *THE j. is-join j*

lemma *is-meet-meet*: *is-meet* (*meet*::'*a* ⇒ '*a* ⇒ ('*a*::*meet-semilorder*))
⟨*proof*⟩

lemma *meet-unique*: (*is-meet m*) = (*m* = *meet*)
⟨*proof*⟩

lemma *is-join-join*: *is-join* (*join*::'*a* ⇒ '*a* ⇒ ('*a*::*join-semilorder*))
⟨*proof*⟩

lemma *join-unique*: (*is-join j*) = (*j* = *join*)
⟨*proof*⟩

lemma *meet-left-le*: *meet a b* ≤ (*a*::'*a*::*meet-semilorder*)
⟨*proof*⟩

lemma *meet-right-le*: *meet a b* ≤ (*b*::'*a*::*meet-semilorder*)
⟨*proof*⟩

lemma *meet-imp-le*: *x* ≤ *a* ⟹ *x* ≤ *b* ⟹ *x* ≤ *meet a* (*b*::'*a*::*meet-semilorder*)
⟨*proof*⟩

lemma *join-left-le*: *a* ≤ *join a* (*b*::'*a*::*join-semilorder*)
⟨*proof*⟩

lemma *join-right-le*: *b* ≤ *join a* (*b*::'*a*::*join-semilorder*)
⟨*proof*⟩

lemma *join-imp-le*: *a* ≤ *x* ⟹ *b* ≤ *x* ⟹ *join a b* ≤ (*x*::'*a*::*join-semilorder*)
⟨*proof*⟩

lemmas *meet-join-le* = *meet-left-le meet-right-le join-left-le join-right-le*

lemma *is-meet-min*: *is-meet* $(min::'a \Rightarrow 'a \Rightarrow ('a::linorder))$
⟨*proof*⟩

lemma *is-join-max*: *is-join* $(max::'a \Rightarrow 'a \Rightarrow ('a::linorder))$
⟨*proof*⟩

instance *linorder* ⊆ *meet-semilorder*
⟨*proof*⟩

instance *linorder* ⊆ *join-semilorder*
⟨*proof*⟩

instance *linorder* ⊆ *lorder* ⟨*proof*⟩

lemma *meet-min*: $meet = (min :: 'a{\Rightarrow}'a{\Rightarrow}('a::linorder))$
⟨*proof*⟩

lemma *join-max*: $join = (max :: 'a{\Rightarrow}'a{\Rightarrow}('a::linorder))$
⟨*proof*⟩

lemma *meet-idempotent*[*simp*]: $meet\ x\ x = x$
⟨*proof*⟩

lemma *join-idempotent*[*simp*]: $join\ x\ x = x$
⟨*proof*⟩

lemma *meet-comm*: $meet\ x\ y = meet\ y\ x$
⟨*proof*⟩

lemma *join-comm*: $join\ x\ y = join\ y\ x$
⟨*proof*⟩

lemma *meet-assoc*: $meet\ (meet\ x\ y)\ z = meet\ x\ (meet\ y\ z)$ (**is** *?l=?r*)
⟨*proof*⟩

lemma *join-assoc*: $join\ (join\ x\ y)\ z = join\ x\ (join\ y\ z)$ (**is** *?l=?r*)
⟨*proof*⟩

lemma *meet-left-comm*: $meet\ a\ (meet\ b\ c) = meet\ b\ (meet\ a\ c)$
⟨*proof*⟩

lemma *meet-left-idempotent*: $meet\ y\ (meet\ y\ x) = meet\ y\ x$
⟨*proof*⟩

lemma *join-left-comm*: $join\ a\ (join\ b\ c) = join\ b\ (join\ a\ c)$
⟨*proof*⟩

lemma *join-left-idempotent*: $join\ y\ (join\ y\ x) = join\ y\ x$
⟨*proof*⟩

lemmas *meet-aci* = *meet-assoc meet-comm meet-left-comm meet-left-idempotent*

lemmas *join-aci* = *join-assoc join-comm join-left-comm join-left-idempotent*

lemma *le-def-meet*: $(x <= y) = (meet\ x\ y = x)$
⟨*proof*⟩

lemma *le-def-join*: $(x <= y) = (join\ x\ y = y)$
⟨*proof*⟩

lemma *meet-join-absorp*: $meet\ x\ (join\ x\ y) = x$
⟨*proof*⟩

lemma *join-meet-absorp*: $join\ x\ (meet\ x\ y) = x$
⟨*proof*⟩

lemma *meet-mono*: $y \leq z \Longrightarrow meet\ x\ y \leq meet\ x\ z$
⟨*proof*⟩

lemma *join-mono*: $y \leq z \Longrightarrow join\ x\ y \leq join\ x\ z$
⟨*proof*⟩

lemma *distrib-join-le*: $join\ x\ (meet\ y\ z) \leq meet\ (join\ x\ y)\ (join\ x\ z)$ (**is** - <= *?r*)
⟨*proof*⟩

lemma *distrib-meet-le*: $join\ (meet\ x\ y)\ (meet\ x\ z) \leq meet\ x\ (join\ y\ z)$ (**is** *?l* <= -)
⟨*proof*⟩

lemma *meet-join-eq-imp-le*: $a = c \lor a = d \lor b = c \lor b = d \Longrightarrow meet\ a\ b \leq join\ c\ d$
⟨*proof*⟩

lemma *modular-le*: $x \leq z \Longrightarrow join\ x\ (meet\ y\ z) \leq meet\ (join\ x\ y)\ z$ (**is** - $\Longrightarrow$ *?t* <= -)
⟨*proof*⟩

end

5 Set: Set theory for higher-order logic

theory *Set*
imports *LOrder*
begin

A set in HOL is simply a predicate.

5.1 Basic syntax

global

typedecl *'a set*
arities *set* :: (*type*) *type*

consts
{} :: *'a set* ({})
UNIV :: *'a set*
insert :: *'a => 'a set => 'a set*
Collect :: (*'a => bool*) *=> 'a set* — comprehension
Int :: *'a set => 'a set => 'a set* (**infixl** *70*)
Un :: *'a set => 'a set => 'a set* (**infixl** *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
image :: (*'a => 'b*) *=> 'a set => 'b set* (**infixr** ' *90*)

syntax
op : :: *'a => 'a set => bool* (*op* :)
consts
op : :: *'a => 'a set => bool* ((-/ : -) [*50*, *51*] *50*) — membership

local

instance *set* :: (*type*) {*ord*, *minus*} ⟨*proof*⟩

5.2 Additional concrete syntax

syntax
range :: (*'a => 'b*) *=> 'b set* — of function

op ~: :: *'a => 'a set => bool* (*op* ~:) — non-membership
op ~: :: *'a => 'a set => bool* ((-/ ~: -) [*50*, *51*] *50*)

@*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* -./ -) *10*)
@*UNION1* :: *pttrns => 'b set => 'b set* ((*3UN* -./ -) *10*)
@*INTER* :: *pttrn => 'a set => 'b set => 'b set* ((*3INT* -:-./ -) *10*)

```
@UNION      :: pttrn => 'a set => 'b set => 'b set  ((3UN -:-./ -) 10)

-Ball       :: pttrn => 'a set => bool => bool      ((3ALL -:-./ -) [0, 0, 10] 10)
-Bex        :: pttrn => 'a set => bool => bool      ((3EX -:-./ -) [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)
```

translations

```
range f      == f`UNIV
x ~: y       == ~ (x : y)
{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)
```

syntax (**output**)

```
-setle      :: 'a set => 'a set => bool        (op <=)
-setle      :: 'a set => 'a set => bool        ((-/ <= -) [50, 51] 50)
-setless    :: 'a set => 'a set => bool        (op <)
-setless    :: 'a set => 'a set => bool        ((-/ < -) [50, 51] 50)
```

syntax (*xsymbols*)

```
-setle      :: 'a set => 'a set => bool        (op ⊆)
-setle      :: 'a set => 'a set => bool        ((-/ ⊆ -) [50, 51] 50)
-setless    :: 'a set => 'a set => bool        (op ⊂)
-setless    :: 'a set => 'a set => bool        ((-/ ⊂ -) [50, 51] 50)
op Int      :: 'a set => 'a set => 'a set      (infixl ∩ 70)
op Un       :: 'a set => 'a set => 'a set      (infixl ∪ 65)
op :        :: 'a => 'a set => bool            (op ∈)
op :        :: 'a => 'a set => bool            ((-/ ∈ -) [50, 51] 50)
op ~:       :: 'a => 'a set => bool            (op ∉)
op ~:       :: 'a => 'a set => bool            ((-/ ∉ -) [50, 51] 50)
Union       :: 'a set set => 'a set            (⋃ - [90] 90)
Inter       :: 'a set set => 'a set            (⋂ - [90] 90)
-Ball       :: pttrn => 'a set => bool => bool ((3∀ -∈-./ -) [0, 0, 10] 10)
-Bex        :: pttrn => 'a set => bool => bool ((3∃ -∈-./ -) [0, 0, 10] 10)
```

syntax (*HTML* **output**)
```
  -setle      :: 'a set => 'a set => bool          (op ⊆)
  -setle      :: 'a set => 'a set => bool          ((-/ ⊆ -) [50, 51] 50)
  -setless    :: 'a set => 'a set => bool          (op ⊂)
  -setless    :: 'a set => 'a set => bool          ((-/ ⊂ -) [50, 51] 50)
  op Int      :: 'a set => 'a set => 'a set        (infixl ∩ 70)
  op Un       :: 'a set => 'a set => 'a set        (infixl ∪ 65)
  op :        :: 'a => 'a set => bool              (op ∈)
  op :        :: 'a => 'a set => bool              ((-/ ∈ -) [50, 51] 50)
  op ~:       :: 'a => 'a set => bool              (op ∉)
  op ~:       :: 'a => 'a set => bool              ((-/ ∉ -) [50, 51] 50)
  -Ball       :: pttrn => 'a set => bool => bool   ((3∀ -∈-./ -) [0, 0, 10] 10)
  -Bex        :: 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⋃-./ -) 10)
  @INTER1     :: pttrns => 'b set => 'b set             ((3⋂-./ -) 10)
  @UNION      :: pttrn => 'a set => 'b set => 'b set    ((3⋃-∈-./ -) 10)
  @INTER      :: pttrn => 'a set => 'b set => 'b set    ((3⋂-∈-./ -) 10)
```

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

translations
```
  op ⊆ => op <= :: - set => - set => bool
  op ⊂ => op <  :: - set => - set => bool
```

⟨*ML*⟩

5.2.1 Bounded quantifiers

syntax
```
  -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)
```

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*)

syntax (*HOL*)
-*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*)

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*)

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*

⟨*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*⟩

5.3 Rules and definitions

Isomorphisms between predicates and sets.

axioms
mem-Collect-eq: $(a : \{x.\ P(x)\}) = P(a)$
Collect-mem-eq: $\{x.\ x{:}A\} = A$
finalconsts
Collect
op :

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

defs (**overloaded**)
subset-def: *A* <= *B* == *ALL x*:*A*. *x*:*B*
psubset-def: *A* < *B* == (*A*::'*a set*) <= *B* & ~ *A*=*B*
Compl-def: − *A* == {*x*. ~*x*:*A*}
set-diff-def: *A* − *B* == {*x*. *x*:*A* & ~*x*:*B*}

defs

Un-def: *A Un B* == {*x*. *x*:*A* | *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*)}

5.4 Lemmas and proof tool setup

5.4.1 Relating predicates and sets

declare *mem-Collect-eq* [*iff*] *Collect-mem-eq* [*simp*]

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*]

5.4.2 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*: $(ALL\ x{:}A.\ {\sim}P\ x$ ==> $P\ a)$ ==> $a{:}A$ ==> $EX\ x{:}A.\ P\ x$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

⟨*ML*⟩

5.4.3 Congruence rules

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

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

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

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

5.4.4 Subsets

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

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

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

declare *subsetD* [*intro?*] — FIXME

lemma *rev-subsetD*: $c \in A ==> A \subseteq B ==> c \in B$
— 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*]: $A \subseteq\ B ==> (c \notin A ==> P) ==> (c \in B ==> P) ==> P$
— Classical elimination rule.
⟨*proof*⟩

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

⟨*ML*⟩

lemma *contra-subsetD*: $A \subseteq B ==> c \notin B ==> c \notin A$

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

Lemma for creating induction formulae – for "pattern matching" on p. To make the induction hypotheses usable, apply *spec* or *bspec* to put universal quantifiers over the free variables in p.

lemma *setup-induction*: $p{:}A ==> (!!z.\ z{:}A ==> p = z --> R) ==> R$
⟨*proof*⟩

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

⟨*proof*⟩

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

5.4.6 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*: $A \subseteq UNIV$
⟨*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*⟩

5.4.7 The empty set

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

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

5.4.8 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*⟩

5.4.9 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~:A
⟨*proof*⟩

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

5.4.10 Binary union – Un

lemma *Un-iff* [*simp*]: (c : A *Un* B) = (c:A | c:B)
⟨*proof*⟩

lemma *UnI1* [*elim?*]: c:A ==> c : A *Un* B
⟨*proof*⟩

lemma *UnI2* [*elim?*]: c:B ==> c : A *Un* B
⟨*proof*⟩

Classical introduction rule: no commitment to A vs B.

lemma *UnCI* [*intro*!]: (c~:B ==> c:A) ==> c : A *Un* B
⟨*proof*⟩

lemma *UnE* [*elim*!]: c : A *Un* B ==> (c:A ==> P) ==> (c:B ==> P) ==> P
⟨*proof*⟩

5.4.11 Binary intersection – Int

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

lemma *IntI* [*intro*!]: c:A ==> c:B ==> c : A *Int* B
⟨*proof*⟩

lemma *IntD1*: c : A *Int* B ==> c:A
⟨*proof*⟩

lemma *IntD2*: c : A *Int* B ==> c:B
⟨*proof*⟩

lemma *IntE* [*elim*!]: c : A *Int* B ==> (c:A ==> c:B ==> P) ==> P
⟨*proof*⟩

5.4.12 Set difference

lemma *Diff-iff* [*simp*]: (c : A − B) = (c:A & c~:B)
⟨*proof*⟩

lemma *DiffI* [*intro*!]: c : A ==> c ~: B ==> c : A − B
⟨*proof*⟩

lemma *DiffD1*: c : A − B ==> c : A
⟨*proof*⟩

lemma *DiffD2*: c : A − B ==> c : B ==> P
⟨*proof*⟩

lemma *DiffE* [*elim*!]: c : A − B ==> (c:A ==> c~:B ==> P) ==> P
⟨*proof*⟩

5.4.13 Augmenting a set – insert

lemma *insert-iff* [*simp*]: (a : *insert* b A) = (a = b | a:A)
⟨*proof*⟩

lemma *insertI1*: a : *insert* a B
⟨*proof*⟩

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

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

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

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

5.4.14 Singletons, using insert

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

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

lemma *singleton-insert-inj-eq'* [*iff*]: $(insert\ a\ A = \{b\}) = (a = b\ \&\ A \subseteq \{b\})$
⟨*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*: $A - \{x\} \subseteq B \Longrightarrow x \in A \Longrightarrow A \subseteq insert\ x\ B$

⟨*proof*⟩

5.4.15 Unions of families

UN x:A. B x is $\bigcup B$ ' A.

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

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

lemma *UN-E* [*elim*!]: *b* : (*UN x*:*A*. *B x*) ==> (!!*x*. *x*:*A* ==> *b*: *B x* ==> *R*) ==> *R*
⟨*proof*⟩

lemma *UN-cong* [*cong*]:
A = *B* ==> (!!*x*. *x*:*B* ==> *C x* = *D x*) ==> (*UN x*:*A*. *C x*) = (*UN x*:*B*. *D x*)
⟨*proof*⟩

5.4.16 Intersections of families

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

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

lemma *INT-I* [*intro*!]: (!!*x*. *x*:*A* ==> *b*: *B x*) ==> *b* : (*INT x*:*A*. *B x*)
⟨*proof*⟩

lemma *INT-D* [*elim*]: *b* : (*INT x*:*A*. *B x*) ==> *a*:*A* ==> *b*: *B a*
⟨*proof*⟩

lemma *INT-E* [*elim*]: *b* : (*INT x*:*A*. *B x*) ==> (*b*: *B a* ==> *R*) ==> (a~:*A* ==> *R*) ==> *R*
— "Classical" elimination – by the Excluded Middle on $a \in A$.
⟨*proof*⟩

lemma *INT-cong* [*cong*]:
A = *B* ==> (!!*x*. *x*:*B* ==> *C x* = *D x*) ==> (*INT x*:*A*. *C x*) = (*INT x*:*B*. *D x*)
⟨*proof*⟩

5.4.17 Union

lemma *Union-iff* [*simp*]: (*A* : *Union C*) = (*EX X*:*C*. *A*:*X*)
⟨*proof*⟩

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

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

5.4.18 Inter

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

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

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

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

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

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

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

lemma *imageI*: x : A ==> $f\ x$: f ‘ A
⟨*proof*⟩

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

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

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

lemma *image-iff*: (z : f‘A) = (*EX* $x{:}A$. $z = f\ x$)
⟨*proof*⟩

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

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

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

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

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

5.4.19 Set reasoning tools

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

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

declare *subset-UNIV* [*simp*] *subset-refl* [*simp*]

5.4.20 The "proper subset" relation

lemma *psubsetI* [*intro*!]: $A \subseteq B ==> A \neq B ==> A \subset B$
⟨*proof*⟩

lemma *psubsetE* [*elim*!]:
$[|A \subset B;\ [|A \subseteq B;\ ^\sim\ (B{\subseteq}A)|] ==> R|] ==> R$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

declare *atomize-ball* [*symmetric*, *rulify*]

5.5 Further set-theory lemmas

5.5.1 Derived rules involving subsets.

insert.

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

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

lemma *subset-insert*: $x \notin A$ ==> $(A \subseteq$ *insert* x $B) = (A \subseteq B)$
⟨*proof*⟩

Big Union – least upper bound of a set.

lemma *Union-upper*: $B \in A$ ==> $B \subseteq$ *Union* A
⟨*proof*⟩

lemma *Union-least*: (!!X. $X \in A$ ==> $X \subseteq C$) ==> *Union* $A \subseteq C$
⟨*proof*⟩

General union.

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

lemma *UN-least*: (!!x. $x \in A$ ==> $B\ x \subseteq C$) ==> $(\bigcup x{\in}A.\ B\ x) \subseteq C$
⟨*proof*⟩

Big Intersection – greatest lower bound of a set.

lemma *Inter-lower*: $B \in A$ ==> *Inter* $A \subseteq B$
⟨*proof*⟩

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

lemma *Inter-greatest*: (!!X. $X \in A$ ==> $C \subseteq X$) ==> $C \subseteq$ *Inter* A
⟨*proof*⟩

lemma *INT-lower*: $a \in A$ ==> $(\bigcap x{\in}A.\ B\ x) \subseteq B\ a$
⟨*proof*⟩

lemma *INT-greatest*: (!!x. $x \in A$ ==> $C \subseteq B\ x$) ==> $C \subseteq (\bigcap x{\in}A.\ B\ x)$
⟨*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*: $A \subseteq C$ ==> $B \subseteq C$ ==> $A \cup B \subseteq C$
⟨*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*: $C \subseteq A$ ==> $C \subseteq B$ ==> $C \subseteq A \cap B$
⟨*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*⟩

Monotonicity.

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

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

5.5.2 Equalities involving union, intersection, inclusion, etc.

{}.

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

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

lemma *Collect-imp-eq*: $\{x.\ P\ x\ {-}{-}{>}\ Q\ x\} = -\{x.\ P\ x\} \cup \{x.\ Q\ x\}$
⟨*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*: $\{x.\ \exists\, y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
⟨*proof*⟩

lemma *Collect-bex-eq*: $\{x.\ \exists\, y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
⟨*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*]: $insert\ a\ A \neq \{\}$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

image.

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

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

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

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

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

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

lemma *image-Collect*: $f\ `\ \{x.\ P\ x\} = \{f\ x \mid x.\ P\ x\}$
— 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*: $M = N$ ==> (!!$x.\ x \in N$ ==> $f\ x = g\ x$) ==> f‘$M = g$‘N
⟨*proof*⟩

range.

lemma *full-SetCompr-eq*: $\{u.\ \exists\, x.\ u = f\ x\} = \mathit{range}\ f$
⟨*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*: $B \subseteq A$ ==> $A \cap B = B$
⟨*proof*⟩

lemma *Int-absorb2*: $A \subseteq B$ ==> $A \cap B = A$
⟨*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*]: $(A \cap B = UNIV) = (A = UNIV \;\&\; B = UNIV)$
⟨*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*: $A \subseteq B ==> A \cup B = B$
⟨*proof*⟩

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

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

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

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

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

lemma *Un-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*⟩

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap -A = \{\}$
⟨*proof*⟩

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
⟨*proof*⟩

lemma *Compl-partition*: $A \cup -A = \mathit{UNIV}$
⟨*proof*⟩

lemma *Compl-partition2*: $-A \cup A = \mathit{UNIV}$
⟨*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*]: $-\mathit{UNIV} = \{\}$
⟨*proof*⟩

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

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

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

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

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

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

Inter.

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

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

lemma *Inter-insert* [*simp*]: $\bigcap(insert\ a\ B) = a \cap \bigcap B$
⟨*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* [*iff*]:

$(\bigcap A = UNIV) = (\forall x{\in}A.\ x = UNIV)$
$(UNIV = \bigcap A) = (\forall x{\in}A.\ x = UNIV)$
⟨*proof*⟩

UN and *INT*.

Basic identities:

lemma *UN-empty* [*simp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
⟨*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*: $(\bigcup i \in A \cup B.\ M\ i) = (\bigcup i{\in}A.\ M\ i) \cup (\bigcup i{\in}B.\ M\ i)$
⟨*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*]: $(\bigcup y\in A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{\}\ \mathit{else}\ c)$
⟨*proof*⟩

lemma *INT-constant* [*simp*]: $(\bigcap y\in A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ c)$
⟨*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*[*iff*]:
$(\{\} = (\mathit{UN}\ x{:}A.\ B\ x)) = (\forall x\in A.\ B\ x = \{\})$
$((\mathit{UN}\ x{:}A.\ B\ x) = \{\}) = (\forall x\in A.\ B\ x = \{\})$
⟨*proof*⟩

lemma *INTER-UNIV-conv*[*iff*]:
$(\mathit{UNIV} = (\mathit{INT}\ x{:}A.\ B\ x)) = (\forall x\in A.\ B\ x = \mathit{UNIV})$
$((\mathit{INT}\ x{:}A.\ B\ x) = \mathit{UNIV}) = (\forall x\in A.\ B\ x = \mathit{UNIV})$
⟨*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*: $(\forall z \in UNION\ A\ B.\ P\ z) = (\forall x{\in}A.\ \forall z \in B\ x.\ P\ z)$
⟨*proof*⟩

lemma *bex-UN*: $(\exists z \in UNION\ A\ B.\ P\ z) = (\exists x{\in}A.\ \exists z{\in}B\ x.\ P\ z)$
⟨*proof*⟩

Set difference.

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

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

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
⟨*proof*⟩

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - (B::'a\ set)$
⟨*proof*⟩

lemma *Diff-triv*: $A \cap B = \{\} ==> A - B = A$
⟨*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*]: $x \notin A ==> A - insert\ x\ B = A - B$
⟨*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*]: $x \in B ==> insert\ x\ A - B = A - B$
⟨*proof*⟩

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

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

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

lemma *Diff-disjoint* [*simp*]: $A \cap (B - A) = \{\}$
⟨*proof*⟩

lemma *Diff-partition*: $A \subseteq B ==> A \cup (B - A) = B$
⟨*proof*⟩

lemma *double-diff*: $A \subseteq B ==> B \subseteq C ==> B - (C - A) = A$
⟨*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 *all-bool-eq*: $(\forall b::bool.\ P\ b) = (P\ True\ \&\ P\ False)$
⟨*proof*⟩

lemma *bool-induct*: $P\ True \Longrightarrow P\ False \Longrightarrow P\ x$
⟨*proof*⟩

lemma *ex-bool-eq*: $(\exists b::bool.\ P\ b) = (P\ True \mid P\ False)$
⟨*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*: $(A \subseteq B) = (\forall t.\ t \in A\ {-}{-}{>}\ t \in B)$
⟨*proof*⟩

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

lemma *all-not-in-conv* [*iff*]: $(\forall x.\ x \notin A) = (A = \{\})$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *distinct-lemma*: $f\ x \neq f\ y$ ==> $x \neq y$
⟨*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*]:
$!!A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ |\ Q) = ((ALL\ x{:}A.\ P\ x)\ |\ Q)$
$!!A\ P\ Q.\ (ALL\ x{:}A.\ P\ |\ Q\ x) = (P\ |\ (ALL\ x{:}A.\ Q\ x))$
$!!A\ P\ Q.\ (ALL\ x{:}A.\ P\ {-}{-}{>}\ Q\ x) = (P\ {-}{-}{>}\ (ALL\ x{:}A.\ Q\ x))$
$!!A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ {-}{-}{>}\ Q) = ((EX\ x{:}A.\ P\ x)\ {-}{-}{>}\ Q)$
$!!P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
$!!P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
$!!a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
$!!A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
$!!A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$

!!*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*. (~(*ALL x*:*A*. *P x*)) = (*EX x*:*A*. ~*P x*)
⟨*proof*⟩

lemma *bex-simps* [*simp*]:
!!*A P Q*. (*EX x*:*A*. *P x* & *Q*) = ((*EX x*:*A*. *P x*) & *Q*)
!!*A P Q*. (*EX x*:*A*. *P* & *Q x*) = (*P* & (*EX x*:*A*. *Q x*))
!!*P*. (*EX x*:{}. *P x*) = *False*
!!*P*. (*EX x*:*UNIV*. *P x*) = (*EX x*. *P x*)
!!*a B P*. (*EX x*:*insert a B*. *P x*) = (*P*(*a*) | (*EX x*:*B*. *P x*))
!!*A P*. (*EX x*:*Union A*. *P x*) = (*EX y*:*A*. *EX x*:*y*. *P x*)
!!*A B P*. (*EX x*: *UNION A B*. *P x*) = (*EX a*:*A*. *EX x*:*B a*. *P x*)
!!*P Q*. (*EX x*:*Collect Q*. *P x*) = (*EX x*. *Q x* & *P x*)
!!*A P f*. (*EX x*:*f*'*A*. *P x*) = (*EX x*:*A*. *P* (*f x*))
!!*A P*. (~(*EX x*:*A*. *P x*)) = (*ALL x*:*A*. ~*P x*)
⟨*proof*⟩

lemma *ball-conj-distrib*:
(*ALL x*:*A*. *P x* & *Q x*) = ((*ALL x*:*A*. *P x*) & (*ALL x*:*A*. *Q x*))
⟨*proof*⟩

lemma *bex-disj-distrib*:
(*EX x*:*A*. *P x* | *Q x*) = ((*EX x*:*A*. *P x*) | (*EX x*:*A*. *Q x*))
⟨*proof*⟩

Maxiscoping: pulling out big Unions and Intersections.

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

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

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

5.5.3 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B ==> f`A \subseteq f`B$
⟨*proof*⟩

lemma *Pow-mono*: $A \subseteq B ==> Pow\ A \subseteq Pow\ B$
⟨*proof*⟩

lemma *Union-mono*: $A \subseteq B ==> \bigcup A \subseteq \bigcup B$
⟨*proof*⟩

lemma *Inter-anti-mono*: $B \subseteq A ==> \bigcap A \subseteq \bigcap B$
⟨*proof*⟩

lemma *UN-mono*:
$A \subseteq B ==> (!!x.\ x \in A ==> f\ x \subseteq g\ x) ==>$
$(\bigcup x{\in}A.\ f\ x) \subseteq (\bigcup x{\in}B.\ g\ x)$
⟨*proof*⟩

lemma *INT-anti-mono*:
$B \subseteq A ==> (!!x.\ x \in A ==> f\ x \subseteq g\ x) ==>$
$(\bigcap x{\in}A.\ f\ x) \subseteq (\bigcap x{\in}A.\ g\ x)$
— The last inclusion is POSITIVE!
⟨*proof*⟩

lemma *insert-mono*: $C \subseteq D ==> insert\ a\ C \subseteq insert\ a\ D$
⟨*proof*⟩

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

lemma *Int-mono*: $A \subseteq C ==> B \subseteq D ==> A \cap B \subseteq C \cap D$
⟨*proof*⟩

lemma *Diff-mono*: $A \subseteq C ==> D \subseteq B ==> A - B \subseteq C - D$
⟨*proof*⟩

lemma *Compl-anti-mono*: $A \subseteq B ==> -B \subseteq -A$
⟨*proof*⟩

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B ==> x \in A --> x \in B$
⟨*proof*⟩

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

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

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

lemma *imp-refl*: *P --> P* ⟨*proof*⟩

lemma *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* $\subseteq$ *Collect Q*
⟨*proof*⟩

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

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*⟩

5.6 Inverse image of a function

constdefs
vimage :: *('a => 'b) => 'b set => 'a set* (**infixr** *-` 90*)

$f -` B == \{x.\ f\ x : B\}$

5.6.1 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*⟩

5.6.2 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*: $A \subseteq B ==> f\ -`\ A \subseteq f\ -`\ B$
— monotonicity
⟨*proof*⟩

5.7 Getting the Contents of a Singleton Set

constdefs
contents :: $'a\ set => 'a$
contents $X == THE\ x.\ X = \{x\}$

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

5.8 Transitivity rules for calculational reasoning

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

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

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

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

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

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

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 *basic-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
set-rev-mp
set-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

end

6 Typedef: HOL type definitions

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

locale *type-definition* =
  fixes *Rep* **and** *Abs* **and** *A*
  assumes *Rep*: *Rep* $x \in A$
    and *Rep-inverse*: *Abs* (*Rep* x) = x
    and *Abs-inverse*: $y \in A$ ==> *Rep* (*Abs* y) = y
  — This will be axiomatized for each typedef!

lemma (**in** *type-definition*) *Rep-inject*:
  (*Rep* x = *Rep* y) = (x = y)
⟨*proof*⟩

lemma (**in** *type-definition*) *Abs-inject*:
  assumes x: $x \in A$ **and** y: $y \in A$
  shows (*Abs* x = *Abs* y) = (x = y)
⟨*proof*⟩

lemma (**in** *type-definition*) *Rep-cases* [*cases set*]:
  assumes y: $y \in A$
    and *hyp*: !!x. y = *Rep* x ==> P
  shows P
⟨*proof*⟩

lemma (**in** *type-definition*) *Abs-cases* [*cases type*]:
  assumes r: !!y. x = *Abs* y ==> $y \in A$ ==> P
  shows P
⟨*proof*⟩

lemma (**in** *type-definition*) *Rep-induct* [*induct set*]:
  assumes y: $y \in A$
    and *hyp*: !!x. P (*Rep* x)
  shows P y
⟨*proof*⟩

lemma (**in** *type-definition*) *Abs-induct* [*induct type*]:
  assumes r: !!y. $y \in A$ ==> P (*Abs* y)
  shows P x
⟨*proof*⟩

⟨*ML*⟩

end

theory *Fun*
imports *Typedef*
begin

instance *set* :: (*type*) *order*
  ⟨*proof*⟩

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
```

```
constdefs
 override-on :: ('a => 'b) => ('a => 'b) => 'a set => ('a => 'b)
override-on f g A == %a. if a : A then g a else f a

 id :: 'a => 'a
id == %x. x

 comp :: ['b => 'c, 'a => 'b, 'a] => 'c   (infixl o 55)
f o g == %x. f(g(x))
```

compatibility

```
lemmas o-def = comp-def

syntax (xsymbols)
  comp :: ['b => 'c, 'a => 'b, 'a] => 'c        (infixl ∘ 55)
syntax (HTML output)
  comp :: ['b => 'c, 'a => 'b, 'a] => 'c        (infixl ∘ 55)
```

```
constdefs
  inj-on :: ['a => 'b, 'a set] => bool
    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.

```
syntax inj  :: ('a => 'b) => bool
translations
  inj f == inj-on f UNIV

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

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

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

lemma *expand-fun-eq*: $(f = g) = (!\ x.\ f(x){=}g(x))$
⟨*proof*⟩

lemma *apply-inverse*:
$[|\ f(x){=}u;\ \ !!x.\ P(x) ==> g(f(x)) = x;\ \ P(x)\ |] ==> x{=}g(u)$
⟨*proof*⟩

The Identity Function: *id*

lemma *id-apply* [*simp*]: $id\ x = x$
⟨*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*⟩

6.1 The Composition Operator: $f \circ g$

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

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

6.2 The Injectivity Predicate, *inj*

NB: *inj* now just translates to *inj-on*

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*: $inj\ f \Longrightarrow b : range\ f = (EX!\ x.\ b = f\ x)$
⟨*proof*⟩

lemma *injD*: $[|\ inj(f);\ f(x) = f(y)\ |] ==> x{=}y$
⟨*proof*⟩

lemma *inj-eq*: $inj(f) ==> (f(x) = f(y)) = (x{=}y)$
⟨*proof*⟩

6.3 The Predicate *inj-on*: Injectivity On A Restricted Domain

lemma *inj-onI*:
$(!!\ x\ y.\ [|\ x{:}A;\ y{:}A;\ f(x) = f(y)\ |] ==> x{=}y) ==> inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *inj-on-inverseI*: $(!!x.\ x{:}A ==> g(f(x)) = x) ==> inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *inj-onD*: $[|\ inj\text{-}on\ f\ A;\ f(x){=}f(y);\ x{:}A;\ y{:}A\ |] ==> x{=}y$
⟨*proof*⟩

lemma *inj-on-iff*: $[|\ inj\text{-}on\ f\ A;\ x{:}A;\ y{:}A\ |] ==> (f(x){=}f(y)) = (x{=}y)$
⟨*proof*⟩

lemma *comp-inj-on*:
$[|\ inj\text{-}on\ f\ A;\ inj\text{-}on\ g\ (f`A)\ |] ==> inj\text{-}on\ (g\ o\ f)\ A$
⟨*proof*⟩

lemma *inj-on-imageI*: $inj\text{-}on\ (g\ o\ f)\ A \Longrightarrow inj\text{-}on\ g\ (f\ `\ A)$
⟨*proof*⟩

lemma *inj-on-image-iff*: $[\![\ ALL\ x{:}A.\ ALL\ y{:}A.\ (g(f\ x) = g(f\ y)) = (g\ x = g\ y);$
$inj\text{-}on\ f\ A\ ]\!] \Longrightarrow inj\text{-}on\ g\ (f\ `\ A) = inj\text{-}on\ g\ A$
⟨*proof*⟩

lemma *inj-on-contraD*: $[|\ inj\text{-}on\ f\ A;\ \sim x{=}y;\ x{:}A;\ y{:}A\ |] ==> \sim f(x){=}f(y)$
⟨*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*⟩

6.4 The Predicate *surj*: Surjectivity

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*⟩

6.5 The Predicate *bij*: Bijectivity

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*⟩

6.6 Facts About the Identity Function

We seem to need both the *id* forms and the $\lambda x.\ x$ forms. The latter can arise by rewriting, while *id* may be used explicitly.

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*⟩

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

6.7 Function Updating

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*]: $(f(x{:=}y))z = (\textit{if } z{=}x \textit{ then } y \textit{ else } f\ z)$
⟨*proof*⟩

lemma *fun-upd-same*: $(f(x{:=}y))\ x = y$
⟨*proof*⟩

lemma *fun-upd-other*: $z\tilde{}{=}x \;{=}{=}{>}\; (f(x{:=}y))\ z = f\ z$
⟨*proof*⟩

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

lemma *fun-upd-twist*: $a\ \tilde{}{=}\ c \;{=}{=}{>}\; (m(a{:=}b))(c{:=}d) = (m(c{:=}d))(a{:=}b)$
⟨*proof*⟩

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

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

6.8 *override-on*

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

lemma *override-on-apply-notin*[*simp*]: $a\ \tilde{}{:}\ A \;{=}{=}{>}\; (\textit{override-on}\ f\ g\ A)\ a = f\ a$
⟨*proof*⟩

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

6.9 swap

constdefs
$\textit{swap} :: [{'a},\ {'a},\ {'a} \;{=}{>}\; {'b}] \;{=}{>}\; ({'a} \;{=}{>}\; {'b})$
$\textit{swap}\ a\ b\ f \;{=}{=}\; f(a := f\ b,\ b{:=}\ f\ a)$

lemma *swap-self*: *swap a a f* = *f*
⟨*proof*⟩

lemma *swap-commute*: *swap a b f* = *swap b a f*
⟨*proof*⟩

lemma *swap-nilpotent* [*simp*]: *swap a b* (*swap a b f*) = *f*
⟨*proof*⟩

lemma *inj-on-imp-inj-on-swap*:
[|*inj-on f A*; *a* ∈ *A*; *b* ∈ *A*|] ==> *inj-on* (*swap a b f*) *A*
⟨*proof*⟩

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

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*⟩

The ML section includes some compatibility bindings and a simproc for function updates, in addition to the usual ML-bindings of theorems.

⟨*ML*⟩

end

7 Product-Type: Cartesian products

theory *Product-Type*
imports *Fun*
uses (*Tools/split-rule.ML*)
begin

7.1 Unit

typedef *unit* = {*True*}
⟨*proof*⟩

constdefs
Unity :: *unit* ('(')')
() == *Abs-unit True*

lemma *unit-eq*: *u* = ()
⟨*proof*⟩

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

⟨*ML*⟩

lemma *unit-all-eq1*: (!!*x*::*unit*. *PROP P x*) == *PROP P* ()
  ⟨*proof*⟩

lemma *unit-all-eq2*: (!!*x*::*unit*. *PROP P*) == *PROP P*
  ⟨*proof*⟩

lemma *unit-induct* [*induct type*: *unit*]: *P* () ==> *P x*
  ⟨*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*]: (%*u*::*unit*. *f* ()) = *f*
  ⟨*proof*⟩

7.2 Pairs

7.2.1 Type definition

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

global

typedef (*Prod*)
  ('*a*, '*b*) * (**infixr** *20*)
    = {*f*. *EX 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*)

local

7.2.2 Abstract constants and syntax

global

consts
  fst :: '*a* * '*b* => '*a*
  snd :: '*a* * '*b* => '*b*
  split :: [['*a*, '*b*] => '*c*, '*a* * '*b*] => '*c*
  curry :: ['*a* * '*b* => '*c*, '*a*, '*b*] => '*c*
  prod-fun :: ['*a* => '*b*, '*c* => '*d*, '*a* * '*c*] => '*b* * '*d*
  Pair :: ['*a*, '*b*] => '*a* * '*b*
  Sigma :: ['*a set*, '*a* => '*b set*] => ('*a* * '*b*) *set*

local

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        (-,/ -)
  @Sigma ::[pttrn, 'a set, 'b set] => ('a * 'b) set ((3SIGMA -:-./ -) 10)
  @Times ::['a set,  'a => 'b set] => ('a * 'b) set (infixr <*> 80)
```

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


  SIGMA x:A. B => Sigma A (%x. B)
  A <*> B      => Sigma A (-K B)
```

⟨*ML*⟩

Deleted x-symbol and html support using Σ (Sigma) because of the danger of confusion with Sum.

syntax (*xsymbols*)

```
  @Times :: ['a set,  'a => 'b set] => ('a * 'b) set  (- × - [81, 80] 80)
```

syntax (*HTML* **output**)

```
  @Times :: ['a set,  'a => 'b set] => ('a * 'b) set  (- × - [81, 80] 80)
```

⟨*ML*⟩

7.2.3 Definitions

defs

```
  Pair-def:    Pair a b == Abs-Prod(Pair-Rep a b)
  fst-def:     fst p == THE a. EX b. p = (a, b)
  snd-def:     snd p == THE b. EX a. p = (a, b)
  split-def:   split == (%c p. c (fst p) (snd p))
```

curry-def: *curry* == (%*c x y*. *c* (*x*,*y*))
prod-fun-def: *prod-fun f g* == *split*(%*x y*.(*f*(*x*), *g*(*y*)))
Sigma-def: *Sigma A B* == *UN x*:*A*. *UN y*:*B*(*x*). {(*x*, *y*)}

7.2.4 Lemmas and proof tool setup

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*:
(*a*, *b*) = (*a′*, *b′*) ==> (*a* = *a′* ==> *b* = *b′* ==> *R*) ==> *R*
⟨*proof*⟩

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

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

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

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

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

lemma *PairE-lemma*: *EX x y*. *p* = (*x*, *y*)
⟨*proof*⟩

lemma *PairE* [*cases type*: *]: (!!*x y*. *p* = (*x*, *y*) ==> *Q*) ==> *Q*
⟨*proof*⟩

⟨*ML*⟩

lemma *surjective-pairing*: *p* = (*fst p*, *snd p*)
— Do not add as rewrite rule: invalidates some proofs in IMP
⟨*proof*⟩

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

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

lemma *split-paired-all*: $(!!x.\ PROP\ P\ x) == (!!a\ b.\ PROP\ P\ (a,\ b))$
⟨*proof*⟩

lemmas *split-tupled-all* = *split-paired-all unit-all-eq2*

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.

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

lemma *split-curry* [*simp*]: *split* (*curry f*) = *f*
⟨*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-conv* [*simp*]: *curry f a b* $= f\ (a,b)$
⟨*proof*⟩

lemma *prod-induct* [*induct type*: ∗]: $!!x.\ (!!a\ b.\ P\ (a,\ b)) ==> P\ x$
⟨*proof*⟩

lemma *split-paired-Ex* [*simp*]: $(EX\ x.\ P\ x) = (EX\ a\ b.\ P\ (a,\ b))$
⟨*proof*⟩

lemma *split-conv* [*simp*]: *split* $c\ (a,\ b) = c\ a\ b$
⟨*proof*⟩

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

lemmas *splitI* = *split-conv* [*THEN iffD2*, *standard*]
lemmas *splitD* = *split-conv* [*THEN iffD1*, *standard*]

lemma *split-Pair-apply*: *split* (%x y. f (x, y)) = f
— Subsumes the old *split-Pair* when f is the identity function.
⟨*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 *Pair-fst-snd-eq*: !!s t. (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*⟩

lemma *split-weak-cong*: p = q ==> *split* c p = *split* c q
— Prevents simplification of c: much faster
⟨*proof*⟩

lemma *split-eta*: (%(x, y). f (x, y)) = f
⟨*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*: (%(x, y). P x y) z = P (*fst* z) (*snd* z)
⟨*proof*⟩

lemma *split-split*: 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*: R (*split* c p) = (~(*EX* x y. p = (x, y) & (~R (c x y))))
⟨*proof*⟩

split used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

lemma *splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *split c p*
⟨*proof*⟩

lemma *splitI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *split c p x*
⟨*proof*⟩

lemma *splitE*: *split c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
⟨*proof*⟩

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
⟨*proof*⟩

lemma *splitE2*:
[| *Q* (*split P z*); !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R* |] ==> *R*
⟨*proof*⟩

lemma *splitD'*: *split R* (*a*,*b*) *c* ==> *R a b c*
⟨*proof*⟩

lemma *mem-splitI*: *z*: *c a b* ==> *z*: *split c* (*a*, *b*)
⟨*proof*⟩

lemma *mem-splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *z*: *c a b* |] ==> *z*: *split c p*
⟨*proof*⟩

lemma *mem-splitE*: [| *z*: *split c p*; !!*x y*. [| *p* = (*x*,*y*); *z*: *c x y* |] ==> *Q* |] ==> *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*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
⟨*proof*⟩

lemma *split-eta-SetCompr2* [*simp*]: (%*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*:
(%*u*. *f* (*g* (*fst u*)) (*snd u*)) = (*split* (%*x*. *f* (*g x*)))
⟨*proof*⟩

lemma *The-split-eq* [*simp*]: (*THE* (x',y'). $x = x'$ & $y = y'$) = (x, y)
⟨*proof*⟩

lemma *injective-fst-snd*: !!x y. [|*fst* x = *fst* y; *snd* x = *snd* y|] ==> $x = y$
⟨*proof*⟩

prod-fun — action of the product functor upon functions.

lemma *prod-fun* [*simp*]: *prod-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* ==> (*f a*, *g b*) : *prod-fun f g* ‘ *r*
⟨*proof*⟩

lemma *prod-fun-imageE* [*elim*!]:
[| *c*: (*prod-fun f g*)‘*r*; !!*x y*. [| *c*=(*f*(*x*),*g*(*y*)); (*x*,*y*):*r* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

constdefs
upd-fst :: ('*a* => '*c*) => '*a* * '*b* => '*c* * '*b*
upd-fst f == *prod-fun f id*

upd-snd :: ('*b* => '*c*) => '*a* * '*b* => '*a* * '*c*
upd-snd f == *prod-fun id f*

lemma *upd-fst-conv* [*simp*]: *upd-fst f* (*x*,*y*) = (*f x*,*y*)
⟨*proof*⟩

lemma *upd-snd-conv* [*simp*]: *upd-snd f* (*x*,*y*) = (*x*,*f y*)
⟨*proof*⟩

Disjoint union of a family of sets – Sigma.

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 $(a, b) \in A \times B$ – 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*:
$[\![A = B; !!x.\ x \in B \Longrightarrow C\ x = D\ x]\!]$
$\Longrightarrow (SIGMA\ x{:}\ A.\ C\ x) = (SIGMA\ x{:}\ B.\ D\ x)$
⟨*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*⟩

Complex rules for Sigma.

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

lemma *split-paired-Bex-Sigma* [*simp*]:
(*EX z*: *Sigma A B*. *P z*) = (*EX x*:*A*. *EX y*: *B x*. *P*(*x*,*y*))
⟨*proof*⟩

lemma *Sigma-Un-distrib1*: (*SIGMA i*:*I Un J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Un* (*SIGMA j*:*J*. *C*(*j*))
⟨*proof*⟩

lemma *Sigma-Un-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Un B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Un* (*SIGMA i*:*I*. *B*(*i*))
⟨*proof*⟩

lemma *Sigma-Int-distrib1*: (*SIGMA i*:*I Int J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Int* (*SIGMA j*:*J*. *C*(*j*))
⟨*proof*⟩

lemma *Sigma-Int-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Int B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Int* (*SIGMA i*:*I*. *B*(*i*))
⟨*proof*⟩

lemma *Sigma-Diff-distrib1*: (*SIGMA i*:*I* − *J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) − (*SIGMA j*:*J*. *C*(*j*))
⟨*proof*⟩

lemma *Sigma-Diff-distrib2*: (*SIGMA i*:*I*. *A*(*i*) − *B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) − (*SIGMA i*:*I*. *B*(*i*))
⟨*proof*⟩

lemma *Sigma-Union*: *Sigma* (*Union X*) *B* = (*UN A*:*X*. *Sigma A B*)
⟨*proof*⟩

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: (*A Un B*) <*> *C* = (*A* <*> *C*) *Un* (*B* <*> *C*)
⟨*proof*⟩

lemma *Times-Int-distrib1*: (*A Int B*) <*> *C* = (*A* <*> *C*) *Int* (*B* <*> *C*)
⟨*proof*⟩

lemma *Times-Diff-distrib1*: (*A* − *B*) <*> *C* = (*A* <*> *C*) − (*B* <*> *C*)
⟨*proof*⟩

lemma *pair-imageI* [*intro*]: (*a*, *b*) : *A* ==> *f a b* : (%(*a*, *b*). *f a b*) ‘ *A*
⟨*proof*⟩

Setup of internal *split-rule*.

constdefs
internal-split :: ('*a* => '*b* => '*c*) => '*a* ∗ '*b* => '*c*
internal-split == *split*

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

hide *const internal-split*

⟨*ML*⟩

7.3 Code generator setup

types-code
∗ ((- ∗/ -))
attach (*term-of*) ⟪
fun term-of-id-42 f T g U (*x*, *y*) = *HOLogic.pair-const T U* $ *f x* $ *g y*;
⟫
attach (*test*) ⟪
fun gen-id-42 aG bG i = (*aG i*, *bG i*);
⟫

consts-code
Pair ((-,/ -))
fst (*fst*)
snd (*snd*)

⟨*ML*⟩

end

8 FixedPoint: Fixed Points and the Knaster-Tarski Theorem

theory *FixedPoint*
imports *Product-Type*
begin

constdefs
  lfp :: $['a\ set \Rightarrow 'a\ set] \Rightarrow 'a\ set$
    $lfp(f) == Inter(\{u.\ f(u) \subseteq u\})$ — least fixed point

  gfp :: $['a\ set{=}{>}'a\ set] => 'a\ set$
    $gfp(f) == Union(\{u.\ u \subseteq f(u)\})$

8.1 Proof of Knaster-Tarski Theorem using *lfp*

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

lemma *lfp-lowerbound*: $f(A) \subseteq A ==> lfp(f) \subseteq A$
⟨*proof*⟩

lemma *lfp-greatest*: [| !!*u*. $f(u) \subseteq u ==> A{\subseteq}u$ |] $==> A \subseteq lfp(f)$
⟨*proof*⟩

lemma *lfp-lemma2*: $mono(f) ==> f(lfp(f)) \subseteq lfp(f)$
⟨*proof*⟩

lemma *lfp-lemma3*: $mono(f) ==> lfp(f) \subseteq f(lfp(f))$
⟨*proof*⟩

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

8.2 General induction rules for greatest fixed points

lemma *lfp-induct*:
  assumes *lfp*: *a*: $lfp(f)$
    and *mono*: $mono(f)$
    and *indhyp*: !!*x*. [| *x*: $f(lfp(f)\ Int\ \{x.\ P(x)\})$ |] $==> P(x)$
  shows $P(a)$
⟨*proof*⟩

Version of induction for binary relations

lemmas *lfp-induct2* = *lfp-induct* [*of* (*a*,*b*), *split-format* (*complete*)]

lemma *lfp-ordinal-induct*:

assumes *mono*: *mono f*
shows [| !!S. P S ==> P(f S); !!M. !S:M. P S ==> P(*Union* M) |]
==> 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); a:A;
!!x. [| x: f(A *Int* {x. P(x)}) |] ==> P(x)
|] ==> P(a)
⟨*proof*⟩

lemma *lfp-mono*: [| !!Z. f(Z)⊆g(Z) |] ==> *lfp*(f) ⊆ *lfp*(g)
⟨*proof*⟩

8.3 Proof of Knaster-Tarski Theorem using *gfp*

gfp f is the greatest lower bound of the set {u. u ⊆ f u}

lemma *gfp-upperbound*: [| X ⊆ f(X) |] ==> X ⊆ *gfp*(f)
⟨*proof*⟩

lemma *gfp-least*: [| !!u. u ⊆ f(u) ==> u⊆X |] ==> *gfp*(f) ⊆ X
⟨*proof*⟩

lemma *gfp-lemma2*: *mono*(f) ==> *gfp*(f) ⊆ f(*gfp*(f))
⟨*proof*⟩

lemma *gfp-lemma3*: *mono*(f) ==> f(*gfp*(f)) ⊆ *gfp*(f)
⟨*proof*⟩

lemma *gfp-unfold*: *mono*(f) ==> *gfp*(f) = f(*gfp*(f))
⟨*proof*⟩

8.4 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: [| a: X; X ⊆ f(X) |] ==> a : *gfp*(f)
⟨*proof*⟩

lemma *weak-coinduct-image*: !!X. [| a : X; g'X ⊆ f (g'X) |] ==> g a : *gfp* f
⟨*proof*⟩

lemma *coinduct-lemma*:
[| X ⊆ f(X *Un* *gfp*(f)); *mono*(f) |] ==> X *Un* *gfp*(f) ⊆ f(X *Un* *gfp*(f))

⟨*proof*⟩

strong version, thanks to Coen and Frost

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

lemma *gfp-fun-UnI2*: [| $mono(f)$; a: $gfp(f)$ |] ==> a: $f(X\ Un\ gfp(f))$
⟨*proof*⟩

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

Monotonicity of *gfp*!

lemma *gfp-mono*: [| !!Z. $f(Z) \subseteq g(Z)$ |] ==> $gfp(f) \subseteq gfp(g)$
⟨*proof*⟩

⟨ML⟩

end

9 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*
imports *Product-Type*
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*⟩

9.1 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*⟩

9.2 The Disjoint Sum of Sets

lemma *InlI* [*intro*!]: $a : A \Longrightarrow Inl(a) : A <+> B$
⟨*proof*⟩

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

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

Exhaustion rule for sums, a degenerate form of induction

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

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

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

9.3 The *Part* Primitive

lemma *Part-eqI* [*intro*]: [| $a : A$; a=$h(b)$ |] ==> a : *Part* A h
⟨*proof*⟩

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

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

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

lemma *Part-mono*: A<=B ==> *Part* A h <= *Part* B h
⟨*proof*⟩

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

lemma *PartD1*: a : *Part* A h ==> $a : A$

⟨*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

10 Relation: Relations

theory *Relation*
imports *Product-Type*
begin

10.1 Definitions

constdefs
  converse :: ('*a* ∗ '*b*) *set* => ('*b* ∗ '*a*) *set*   ((-ˆ−*1*) [*1000*] *999*)
  *r*ˆ−*1* == {(*y*, *x*). (*x*, *y*) : *r*}
syntax (*xsymbols*)
  converse :: ('*a* ∗ '*b*) *set* => ('*b* ∗ '*a*) *set*   ((-$^{-1}$) [*1000*] *999*)

constdefs
  rel-comp :: [('*b* ∗ '*c*) *set*, ('*a* ∗ '*b*) *set*] => ('*a* ∗ '*c*) *set* (**infixr** *O 60*)
  r O s == {(*x*,*z*). *EX y*. (*x*, *y*) : *s* & (*y*, *z*) : *r*}

  Image :: [('*a* ∗ '*b*) *set*, '*a set*] => '*b set*   (**infixl** ‘‘ *90*)
  r ‘‘ *s* == {*y*. *EX x*:*s*. (*x*,*y*):*r*}

  Id :: ('*a* ∗ '*a*) *set* — the identity relation
  Id == {*p*. *EX x*. *p* = (*x*,*x*)}

  diag :: '*a set* => ('*a* ∗ '*a*) *set* — diagonal: identity over a set
  diag A == ⋃*x*∈*A*. {(*x*,*x*)}

  Domain :: ('*a* ∗ '*b*) *set* => '*a set*
  Domain r == {*x*. *EX y*. (*x*,*y*):*r*}

  Range :: ('*a* ∗ '*b*) *set* => '*b set*
  Range r == *Domain*(*r*ˆ−*1*)

Field :: ('*a* ∗ '*a*) *set* => '*a set*
Field r == *Domain r* ∪ *Range r*

refl :: ['*a set*, ('*a* ∗ '*a*) *set*] => *bool* — reflexivity over a set
refl A r == *r* ⊆ *A* × *A* & (*ALL x*: *A*. (*x*,*x*) : *r*)

sym :: ('*a* ∗ '*a*) *set* => *bool* — symmetry predicate
sym r == *ALL x y*. (*x*,*y*): *r* --> (*y*,*x*): *r*

antisym:: ('*a* ∗ '*a*) *set* => *bool* — antisymmetry predicate
antisym r == *ALL x y*. (*x*,*y*):*r* --> (*y*,*x*):*r* --> *x*=*y*

trans :: ('*a* ∗ '*a*) *set* => *bool* — transitivity predicate
trans r == (*ALL x y z*. (*x*,*y*):*r* --> (*y*,*z*):*r* --> (*x*,*z*):*r*)

single-valued :: ('*a* ∗ '*b*) *set* => *bool*
single-valued r == *ALL x y*. (*x*,*y*):*r* --> (*ALL z*. (*x*,*z*):*r* --> *y*=*z*)

inv-image :: ('*b* ∗ '*b*) *set* => ('*a* => '*b*) => ('*a* ∗ '*a*) *set*
inv-image r f == {(*x*, *y*). (*f x*, *f y*) : *r*}

syntax
reflexive :: ('*a* ∗ '*a*) *set* => *bool* — reflexivity over a type
translations
reflexive == *refl UNIV*

10.2 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 *trans-Id*: *trans Id*
⟨*proof*⟩

10.3 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*!]: $a : A ==> (a,\ a) : diag\ A$
⟨*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*⟩

10.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a,\ b) : s ==> (b,\ c) : r ==> (a,\ c) : r\ O\ s$
⟨*proof*⟩

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s ==>$
$(!!x\ y\ z.\ xz = (x,\ z) ==> (x,\ y) : s ==> (y,\ z) : r\ ==> P) ==> P$
⟨*proof*⟩

lemma *rel-compEpair*:
$(a,\ c) : r\ O\ s ==> (!!y.\ (a,\ y) : s ==> (y,\ c) : r ==> P) ==> P$
⟨*proof*⟩

lemma *R-O-Id* [*simp*]: $R\ O\ Id = R$
⟨*proof*⟩

lemma *Id-O-R* [*simp*]: $Id\ O\ R = R$
⟨*proof*⟩

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
⟨*proof*⟩

lemma *trans-O-subset*: $trans\ r ==> r\ O\ r \subseteq r$
⟨*proof*⟩

lemma *rel-comp-mono*: $r' \subseteq r ==> s' \subseteq s ==> (r'\ O\ s') \subseteq (r\ O\ s)$
⟨*proof*⟩

lemma *rel-comp-subset-Sigma*:
$s \subseteq A \times B$ ==> $r \subseteq B \times C$ ==> $(r\ O\ s) \subseteq A \times C$
⟨*proof*⟩

10.5 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*⟩

10.6 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*⟩

10.7 Symmetry and Transitivity

lemma *symD*: *sym* r ==> (a, b) : r ==> (b, a) : r
⟨*proof*⟩

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*⟩

10.8 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^−*1* ==> (b, a) : r
⟨*proof*⟩

lemma *converseE* [*elim*!]:
yx : r^−*1* ==> (!!x y. yx = (y, x) ==> (x, y) : r ==> P) ==> P
— More general than *converseD*, as it "splits" the member of the relation.
⟨*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-Id* [*simp*]: $Id\hat{\ }{-1} = Id$
⟨*proof*⟩

lemma *converse-diag* [*simp*]: $(diag\ A)\hat{\ }{-1} = diag\ A$
⟨*proof*⟩

lemma *refl-converse*: *refl A r ==> refl A (converse r)*
⟨*proof*⟩

lemma *antisym-converse*: *antisym (converse r) = antisym r*
⟨*proof*⟩

lemma *trans-converse*: *trans (converse r) = trans r*
⟨*proof*⟩

10.9 Domain

lemma *Domain-iff*: *(a : Domain r) = (EX y. (a, y) : r)*
⟨*proof*⟩

lemma *DomainI* [*intro*]: *(a, b) : r ==> a : Domain r*
⟨*proof*⟩

lemma *DomainE* [*elim*!]:
a : Domain r ==> (!!y. (a, y) : r ==> P) ==> 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-mono*: $r \subseteq s$ ==> $Domain\ r \subseteq Domain\ s$
⟨*proof*⟩

10.10 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*⟩

10.11 Image of a set under a relation

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*]: $(a,\ b) : r ==> a : A ==> b : r``A$
⟨*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*]: $diag\ A `` B = A \cap B$
⟨*proof*⟩

lemma *Image-Int-subset*: $R `` (A \cap B) \subseteq R `` A \cap R `` B$
⟨*proof*⟩

lemma *Image-Int-eq*:
$single\text{-}valued\ (converse\ R) ==> R `` (A \cap B) = R `` A \cap R `` B$
⟨*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*:
$[|single\text{-}valued\ (r^{-1});\ A{\neq}\{\}|] ==> r\ ``\ INTER\ A\ B = (\bigcap x{\in}A.\ r\ ``\ B\ x)$
⟨*proof*⟩

lemma *Image-subset-eq*: $(r``A \subseteq B) = (A \subseteq -\ ((r\hat{}-1)\ ``\ (-B)))$
⟨*proof*⟩

10.12 Single valued relations

lemma *single-valuedI*:
$ALL\ x\ y.\ (x,y){:}r\ -->\ (ALL\ z.\ (x,z){:}r\ -->\ y{=}z) ==> single\text{-}valued\ r$
⟨*proof*⟩

lemma *single-valuedD*:
$single\text{-}valued\ r ==> (x,\ y) : r ==> (x,\ z) : r ==> y = z$
⟨*proof*⟩

10.13 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*⟩

10.14 Inverse image

lemma *trans-inv-image*: $trans\ r ==> trans\ (inv\text{-}image\ r\ f)$
⟨*proof*⟩

end

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

⟨ML⟩

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⟩

10.15 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

11 Inductive: Support for inductive sets and types

theory *Inductive*
imports *FixedPoint Sum-Type Relation Record*
uses
(*Tools/inductive-package.ML*)
(*Tools/inductive-realizer.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-realizer.ML*)
(*Tools/datatype-package.ML*)
(*Tools/datatype-codegen.ML*)
(*Tools/recfun-codegen.ML*)
(*Tools/primrec-package.ML*)
begin

11.1 Inductive sets

Inversion of injective functions.

constdefs
$myinv :: ('a => 'b) => ('b => 'a)$
$myinv\ (f :: 'a => 'b) == \lambda y.\ THE\ x.\ f\ x = y$

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

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

hide *const myinv*

Package setup.

⟨*ML*⟩

theorems *basic-monos* [*mono*] =
subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-def2
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-rulify2

11.2 Inductive datatypes and primitive recursion

Package setup.

⟨*ML*⟩

end

12 Transitive-Closure: Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Inductive*
uses (*../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.

consts
  rtrancl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-^*) [*1000*] *999*)

inductive r^*
  intros
    rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: $(a, a) : r$^*
    rtrancl-into-rtrancl [*Pure.intro*]: $(a, b) : r$^* ==> $(b, c) : r$ ==> $(a, c) : r$^*

consts
  trancl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-^+) [*1000*] *999*)

inductive r^+
  intros
    r-into-trancl [*intro*, *Pure.intro*]: $(a, b) : r$ ==> $(a, b) : r$^+
    trancl-into-trancl [*Pure.intro*]: $(a, b) : r$^+ ==> $(b, c) : r$ ==> $(a,c) : r$^+

syntax
  -reflcl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-^=) [*1000*] *999*)
translations
  r^= == $r \cup Id$

syntax (*xsymbols*)
  rtrancl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-*) [*1000*] *999*)
  trancl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-$^+$) [*1000*] *999*)
  -reflcl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-$^=$) [*1000*] *999*)

syntax (*HTML* **output**)
  rtrancl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-*) [*1000*] *999*)
  trancl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-$^+$) [*1000*] *999*)
  -reflcl :: $('a \times 'a)$ *set* => $('a \times 'a)$ *set* ((-$^=$) [*1000*] *999*)

12.1 Reflexive-transitive closure

lemma *r-into-rtrancl* [*intro*]: !!*p*. $p \in r$ ==> $p \in r$^∗
— *rtrancl* of *r* contains *r*
⟨*proof*⟩

lemma *rtrancl-mono*: $r \subseteq s$ ==> *r*^∗ ⊆ *s*^∗
— monotonicity of *rtrancl*
⟨*proof*⟩

theorem *rtrancl-induct* [*consumes 1*, *induct set*: *rtrancl*]:
assumes *a*: (*a*, *b*) : *r*^∗
and *cases*: *P a* !!*y z*. [| (*a*, *y*) : *r*^∗; (*y*, *z*) : *r*; *P y* |] ==> *P z*
shows *P b*
⟨*proof*⟩

lemmas *rtrancl-induct2* =
rtrancl-induct[*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *trans-rtrancl*: *trans*(*r*^∗)
— transitivity of transitive closure!! – by induction
⟨*proof*⟩

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD*, *standard*]

lemma *rtranclE*:
[| (*a*::'*a*,*b*) : *r*^∗; (*a* = *b*) ==> *P*;
!!*y*.[| (*a*,*y*) : *r*^∗; (*y*,*b*) : *r* |] ==> *P*
|] ==> *P*
— elimination of *rtrancl* – by induction on a special formula
⟨*proof*⟩

lemma *converse-rtrancl-into-rtrancl*:
$(a,\ b) \in r \Longrightarrow (b,\ c) \in r^* \Longrightarrow (a,\ c) \in r^*$
⟨*proof*⟩

More r^* equations and inclusions.

lemma *rtrancl-idemp* [*simp*]: (*r*^∗)^∗ = *r*^∗
⟨*proof*⟩

lemma *rtrancl-idemp-self-comp* [*simp*]: *R*^∗ *O R*^∗ = *R*^∗
⟨*proof*⟩

lemma *rtrancl-subset-rtrancl*: *r* ⊆ *s*^∗ ==> *r*^∗ ⊆ *s*^∗
⟨*proof*⟩

lemma *rtrancl-subset*: *R* ⊆ *S* ==> *S* ⊆ *R*^∗ ==> *S*^∗ = *R*^∗
⟨*proof*⟩

lemma *rtrancl-Un-rtrancl*: $(R\hat{}* \cup S\hat{}*)\hat{}* = (R \cup S)\hat{}*$
⟨*proof*⟩

lemma *rtrancl-reflcl* [*simp*]: $(R\hat{}=)\hat{}* = R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-r-diff-Id*: $(r - Id)\hat{}* = r\hat{}*$
⟨*proof*⟩

theorem *rtrancl-converseD*:
assumes *r*: $(x, y) \in (r\hat{}-1)\hat{}*$
shows $(y, x) \in r\hat{}*$
⟨*proof*⟩

theorem *rtrancl-converseI*:
assumes *r*: $(y, x) \in r\hat{}*$
shows $(x, y) \in (r\hat{}-1)\hat{}*$
⟨*proof*⟩

lemma *rtrancl-converse*: $(r\hat{}-1)\hat{}* = (r\hat{}*)\hat{}-1$
⟨*proof*⟩

theorem *converse-rtrancl-induct*[*consumes 1*]:
assumes *major*: $(a, b) : r\hat{}*$
and *cases*: $P\ b$!!$y\ z$. [| $(y, z) : r$; $(z, b) : r\hat{}*$; $P\ z$ |] ==> $P\ y$
shows $P\ a$
⟨*proof*⟩

lemmas *converse-rtrancl-induct2* =
converse-rtrancl-induct[*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *converse-rtranclE*:
[| (x,z):$r\hat{}*$;
$x=z$ ==> P;
!!y. [| (x,y):r; (y,z):$r\hat{}*$ |] ==> P
|] ==> P
⟨*proof*⟩

⟨*ML*⟩

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*⟩

12.2 Transitive closure

lemma *trancl-mono*: $!!p.\ p \in r\hat{}+ ==> r \subseteq s ==> p \in s\hat{}+$
⟨*proof*⟩

lemma *r-into-trancl′*: $!!p.\ p : r ==> p : r\hat{}+$
⟨*proof*⟩

Conversions between *trancl* and *rtrancl*.

lemma *trancl-into-rtrancl*: $(a,\ b) \in r\hat{}+ ==> (a,\ b) \in r\hat{}*$
⟨*proof*⟩

lemma *rtrancl-into-trancl1*: **assumes** r: $(a,\ b) \in r\hat{}*$
shows $!!c.\ (b,\ c) \in r ==> (a,\ c) \in r\hat{}+$ ⟨*proof*⟩

lemma *rtrancl-into-trancl2*: $[|\ (a,b) : r;\ \ (b,c) : r\hat{}*\ |]\ \ ==>\ \ (a,c) : r\hat{}+$
— intro rule from r and *rtrancl*
⟨*proof*⟩

lemma *trancl-induct* [*consumes 1*, *induct set*: *trancl*]:
assumes a: $(a,b) : r\hat{}+$
and *cases*: $!!y.\ (a,\ y) : r ==> P\ y$
$!!y\ z.\ (a,y) : r\hat{}+ ==> (y,\ z) : r ==> P\ y ==> P\ z$
shows $P\ b$
— Nice induction rule for *trancl*
⟨*proof*⟩

lemma *trancl-trans-induct*:
$[|\ (x,y) : r\hat{}+;$
$!!x\ y.\ (x,y) : r ==> P\ x\ y;$
$!!x\ y\ z.\ [|\ (x,y) : r\hat{}+;\ P\ x\ y;\ (y,z) : r\hat{}+;\ P\ y\ z\ |] ==> P\ x\ z$
$|] ==> P\ x\ y$
— Another induction rule for trancl, incorporating transitivity
⟨*proof*⟩

inductive-cases *tranclE*: $(a,\ b) : r\hat{}+$

lemma *trancl-unfold*: $r\hat{}+ = r\ Un\ (r\ O\ r\hat{}+)$
⟨*proof*⟩

lemma *trans-trancl*: $trans(r\hat{}+)$
— Transitivity of r^+
⟨*proof*⟩

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*, *standard*]

lemma *rtrancl-trancl-trancl*: **assumes** r: $(x,\ y) \in r\hat{}*$
shows $!!z.\ (y,\ z) \in r\hat{}+ ==> (x,\ z) \in r\hat{}+$ ⟨*proof*⟩

lemma *trancl-into-trancl2*: $(a,\ b) \in r \ ==> (b,\ c) \in r\hat{}+ \ ==> (a,\ c) \in r\hat{}+$
⟨*proof*⟩

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 *trancl-converseI*: $(x,\ y) \in (r\hat{}+)\hat{}-1 \ ==> (x,\ y) \in (r\hat{}-1)\hat{}+$
⟨*proof*⟩

lemma *trancl-converseD*: $(x,\ y) \in (r\hat{}-1)\hat{}+ \ ==> (x,\ y) \in (r\hat{}+)\hat{}-1$
⟨*proof*⟩

lemma *trancl-converse*: $(r\hat{}-1)\hat{}+ = (r\hat{}+)\hat{}-1$
⟨*proof*⟩

lemma *converse-trancl-induct*:
$[|\ (a,b) : r\hat{}+;\ !!y.\ (y,b) : r \ ==> P(y);$
$!!y\ z.[|\ (y,z) : r;\ \ (z,b) : r\hat{}+;\ \ P(z)\ |] \ ==> P(y)\ |]$
$==> P(a)$
⟨*proof*⟩

lemma *tranclD*: $(x,\ y) \in R\hat{}+ \ ==> EX\ z.\ (x,\ z) \in R \wedge (z,\ y) \in R\hat{}*$
⟨*proof*⟩

lemma *irrefl-tranclI*: $r\hat{}-1 \cap r\hat{}* = \{\} \ ==> (x,\ x) \notin r\hat{}+$
⟨*proof*⟩

lemma *irrefl-trancl-rD*: $!!X.\ ALL\ x.\ (x,\ x) \notin r\hat{}+ \ ==> (x,\ y) \in r \ ==> x \neq y$
⟨*proof*⟩

lemma *trancl-subset-Sigma-aux*:
$(a,\ b) \in r\hat{}* \ ==> r \subseteq A \times A \ ==> a = b \vee a \in A$
⟨*proof*⟩

lemma *trancl-subset-Sigma*: $r \subseteq A \times A \ ==> r\hat{}+ \subseteq A \times A$
⟨*proof*⟩

lemma *reflcl-trancl* [*simp*]: $(r\hat{}+)\hat{}= \ = r\hat{}*$
⟨*proof*⟩

lemma *trancl-reflcl* [*simp*]: $(r\hat{}=)\hat{}+ = r\hat{}*$
⟨*proof*⟩

lemma *trancl-empty* [*simp*]: $\{\}\hat{}+ = \{\}$
⟨*proof*⟩

lemma *rtrancl-empty* [*simp*]: $\{\}\hat{}* = Id$

⟨*proof*⟩

lemma *rtranclD*: $(a,\ b) \in R\hat{}* ==> a = b \lor a \neq b \land (a,\ b) \in R\hat{}+$
⟨*proof*⟩

lemma *rtrancl-eq-or-trancl*:
$(x,y) \in R^* = (x=y \lor x \neq y \land (x,y) \in R^+)$
⟨*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*: $x \in R\hat{}* \lor x \in S\hat{}* ==> x \in (R \cup S)\hat{}*$
⟨*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*:
$[\![\ single\text{-}valued\ r;\ (x,y) \in r\hat{}*;\ (x,z) \in r\hat{}*\ ]\!]$
$\Longrightarrow (y,z) \in r\hat{}* \lor (z,y) \in r\hat{}*$
⟨*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*]:
$(a,\ b) \in r^+ ==> (b,\ c) \in r \dashrightarrow (a,c) \in r^+$
⟨*proof*⟩

lemma *trancl-rtrancl-trancl*:
$(a,\ b) \in r^+ ==> (b,\ c) \in r^* ==> (a,\ c) \in r^+$
⟨*proof*⟩

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl-into-trancl trancl-into-trancl2
rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

declare *trancl-into-rtrancl* [*elim*]

declare *rtranclE* [*cases set*: *rtrancl*]
declare *tranclE* [*cases set*: *trancl*]

12.3 Setup of transitivity reasoner

⟨*ML*⟩

end

13 Wellfounded-Recursion: Well-founded Recursion

theory *Wellfounded-Recursion*
imports *Transitive-Closure*
begin

consts
```
  wfrec-rel :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => ('a * 'b) set
```

inductive *wfrec-rel R F*
intros
```
  wfrecI: ALL z. (z, x) : R --> (z, g z) : wfrec-rel R F ==>
            (x, F g x) : wfrec-rel R F
```

constdefs
```
  wf       :: ('a * 'a)set => bool
  wf(r) == (!P. (!x. (!y. (y,x):r --> P(y)) --> P(x)) --> (!x. P(x)))

  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
wfrec R F == %x. *THE y.* (x, y) : *wfrec-rel R* (%f x. F (*cut f R x*) x)

axclass *wellorder* ⊆ *linorder*
wf: *wf* {(x,y::'a::ord). x<y}

lemma *wfUNIVI*:
(!!P x. (ALL x. (ALL y. (y,x) : r --> P(y)) --> P(x)) ==> P(x)) ==>
wf(r)
⟨*proof*⟩

Restriction to domain *A*. If *r* is well-founded over *A* then *wf r*

lemma *wfI*:
[| r <= A <*> A;
!!x P. [| ALL x. (ALL y. (y,x) : r --> P y) --> P x; x:A |] ==> P x |]
==> wf r
⟨*proof*⟩

lemma *wf-induct*:
[| wf(r);
!!x.[| ALL y. (y,x): r --> P(y) |] ==> P(x)
|] ==> P(a)
⟨*proof*⟩

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *case-names less*, *induct set*: *wf*]

lemma *wf-not-sym* [*rule-format*]: wf(r) ==> ALL x. (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*]

transitive closure of a well-founded relation is well-founded!

lemma *wf-trancl*: wf(r) ==> wf(r^+)
⟨*proof*⟩

lemma *wf-converse-trancl*: wf (r^-1) ==> wf ((r^+)^-1)
⟨*proof*⟩

13.0.1 Minimal-element characterization of well-foundedness

lemma lemma1: wf r ==> x:Q --> (EX z:Q. ALL y. (y,z):r --> y~:Q)
⟨proof⟩

lemma lemma2: (ALL Q x. x:Q --> (EX z:Q. ALL y. (y,z):r --> y~:Q))
==> wf r
⟨proof⟩

lemma wf-eq-minimal: wf r = (ALL Q x. x:Q --> (EX z:Q. ALL y. (y,z):r
--> y~:Q))
⟨proof⟩

13.0.2 Other simple well-foundedness results

Well-foundedness of subsets

lemma wf-subset: [| wf(r); p<=r |] ==> wf(p)
⟨proof⟩

Well-foundedness of the empty relation

lemma wf-empty [iff]: wf({})
⟨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⟩

13.0.3 Well-Foundedness Results for Unions

Well-foundedness of indexed union with disjoint domains and ranges

lemma wf-UN: [| ALL i:I. wf(r i);
    ALL i:I. ALL j:I. r i ~= r j --> Domain(r i) Int Range(r j) = {}
  |] ==> wf(UN i:I. r i)
⟨proof⟩

lemma 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⟩

13.0.4 acyclic

lemma *acyclicI*: ALL x. (x, x) ~: r^+ ==> acyclic r
⟨proof⟩

lemma *wf-acyclic*: wf r ==> acyclic r
⟨proof⟩

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⟩

lemma *acyclic-impl-antisym-rtrancl*: acyclic r ==> antisym(r^*)
⟨proof⟩

lemma *acyclic-subset*: [| acyclic s; r <= s |] ==> acyclic r
⟨proof⟩

13.1 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. (x, y) : wfrec-rel R F
⟨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⟩

* 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⟩

13.2 Code generator setup

consts-code
  wfrec (⟨**module**⟩*wfrec?*)
attach ⟪
fun wfrec f x = f (wfrec f) x;
⟫

13.3 Variants for TFL: the Recdef Package

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*⟩

13.4 LEAST and wellorderings

See also *wf-linord-ex-has-least* and its consequences in *Wellfounded-Relations.ML*

lemma *wellorder-Least-lemma* [*rule-format*]:
  P (k::'a::wellorder) --> P (LEAST x. P(x)) & (LEAST x. P(x)) <= k
⟨*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 ==> P (Least P)*
⟨*proof*⟩

lemma *LeastI2*:
  [| *P (a::'a::wellorder)*; !!*x. P x ==> Q x* |] ==> *Q (Least P)*
⟨*proof*⟩

lemma *LeastI2-ex*:
  [| *EX a::'a::wellorder. P a*; !!*x. P x ==> Q x* |] ==> *Q (Least P)*
⟨*proof*⟩

lemma *not-less-Least*: [| *k < (LEAST x. P x)* |] ==> ~*P (k::'a::wellorder)*
⟨*proof*⟩

⟨*ML*⟩

end

14 OrderedGroup: Ordered Groups

theory *OrderedGroup*
imports *Inductive LOrder*
uses *../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.

14.1 Semigroups, Groups

axclass *semigroup-add* $\subseteq$ *plus*
add-assoc: $(a + b) + c = a + (b + c)$

axclass *ab-semigroup-add* $\subseteq$ *semigroup-add*
add-commute: $a + b = b + a$

lemma *add-left-commute*: $a + (b + c) = b + (a + (c::'a::ab\text{-}semigroup\text{-}add))$
$\langle proof \rangle$

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

axclass *semigroup-mult* $\subseteq$ *times*
mult-assoc: $(a * b) * c = a * (b * c)$

axclass *ab-semigroup-mult* $\subseteq$ *semigroup-mult*
mult-commute: $a * b = b * a$

lemma *mult-left-commute*: $a * (b * c) = b * (a * (c::'a::ab\text{-}semigroup\text{-}mult))$
$\langle proof \rangle$

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

axclass *comm-monoid-add* $\subseteq$ *zero*, *ab-semigroup-add*

add-0[*simp*]: $0 + a = a$

axclass *monoid-mult* ⊆ *one*, *semigroup-mult*
mult-1-left[*simp*]: $1 * a = a$
mult-1-right[*simp*]: $a * 1 = a$

axclass *comm-monoid-mult* ⊆ *one*, *ab-semigroup-mult*
mult-1: $1 * a = a$

instance *comm-monoid-mult* ⊆ *monoid-mult*
⟨*proof*⟩

axclass *cancel-semigroup-add* ⊆ *semigroup-add*
add-left-imp-eq: $a + b = a + c \Longrightarrow b = c$
add-right-imp-eq: $b + a = c + a \Longrightarrow b = c$

axclass *cancel-ab-semigroup-add* ⊆ *ab-semigroup-add*
add-imp-eq: $a + b = a + c \Longrightarrow b = c$

instance *cancel-ab-semigroup-add* ⊆ *cancel-semigroup-add*
⟨*proof*⟩

axclass *ab-group-add* ⊆ *minus*, *comm-monoid-add*
left-minus[*simp*]: $- a + a = 0$
diff-minus: $a - b = a + (-b)$

instance *ab-group-add* ⊆ *cancel-ab-semigroup-add*
⟨*proof*⟩

lemma *add-0-right* [*simp*]: $a + 0 = (a::'a::\mathit{comm\text{-}monoid\text{-}add})$
⟨*proof*⟩

lemma *add-left-cancel* [*simp*]:
$(a + b = a + c) = (b = (c::'a::\mathit{cancel\text{-}semigroup\text{-}add}))$
⟨*proof*⟩

lemma *add-right-cancel* [*simp*]:
$(b + a = c + a) = (b = (c::'a::\mathit{cancel\text{-}semigroup\text{-}add}))$
⟨*proof*⟩

lemma *right-minus* [*simp*]: $a + -(a::'a::\mathit{ab\text{-}group\text{-}add}) = 0$
⟨*proof*⟩

lemma *right-minus-eq*: $(a - b = 0) = (a = (b::'a::\mathit{ab\text{-}group\text{-}add}))$
⟨*proof*⟩

lemma *minus-minus* [*simp*]: $- (- (a::'a::\mathit{ab\text{-}group\text{-}add})) = a$
⟨*proof*⟩

lemma *equals-zero-I*: $a+b = 0 ==> -a = (b::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *minus-zero* [*simp*]: $- 0 = (0::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *diff-self* [*simp*]: $a - (a::'a::ab\text{-}group\text{-}add) = 0$
⟨*proof*⟩

lemma *diff-0* [*simp*]: $(0::'a::ab\text{-}group\text{-}add) - a = -a$
⟨*proof*⟩

lemma *diff-0-right* [*simp*]: $a - (0::'a::ab\text{-}group\text{-}add) = a$
⟨*proof*⟩

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + (b::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *neg-equal-iff-equal* [*simp*]: $(-a = -b) = (a = (b::'a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *neg-equal-0-iff-equal* [*simp*]: $(-a = 0) = (a = (0::'a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *neg-0-equal-iff-equal* [*simp*]: $(0 = -a) = (0 = (a::'a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*: $(a = - b) = (b = - (a::'a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *minus-equation-iff*: $(- a = b) = (- (b::'a::ab\text{-}group\text{-}add) = a)$
⟨*proof*⟩

lemma *minus-add-distrib* [*simp*]: $- (a + b) = -a + -(b::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *minus-diff-eq* [*simp*]: $- (a - b) = b - (a::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

14.2 (Partially) Ordered Groups

axclass *pordered-ab-semigroup-add* ⊆ *order*, *ab-semigroup-add*
add-left-mono: $a \le b \Longrightarrow c + a \le c + b$

axclass *pordered-cancel-ab-semigroup-add* ⊆ *pordered-ab-semigroup-add*, *cancel-ab-semigroup-add*

instance *pordered-cancel-ab-semigroup-add* ⊆ *pordered-ab-semigroup-add* ⟨*proof*⟩

axclass *pordered-ab-semigroup-add-imp-le* $\subseteq$ *pordered-cancel-ab-semigroup-add*
add-le-imp-le-left: $c + a \leq c + b \Longrightarrow a \leq b$

axclass *pordered-ab-group-add* $\subseteq$ *ab-group-add*, *pordered-ab-semigroup-add*

instance *pordered-ab-group-add* $\subseteq$ *pordered-ab-semigroup-add-imp-le*
⟨*proof*⟩

axclass *ordered-cancel-ab-semigroup-add* $\subseteq$ *pordered-cancel-ab-semigroup-add*, *linorder*

instance *ordered-cancel-ab-semigroup-add* $\subseteq$ *pordered-ab-semigroup-add-imp-le*
⟨*proof*⟩

lemma *add-right-mono*: $a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add) ==> a + c \leq b + c$
⟨*proof*⟩

non-strict, in both arguments

lemma *add-mono*:
$[|a \leq b; \ c \leq d|] ==> a + c \leq b + (d::'a::pordered\text{-}ab\text{-}semigroup\text{-}add)$
⟨*proof*⟩

lemma *add-strict-left-mono*:
$a < b ==> c + a < c + (b::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
⟨*proof*⟩

lemma *add-strict-right-mono*:
$a < b ==> a + c < b + (c::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
⟨*proof*⟩

Strict monotonicity in both arguments

lemma *add-strict-mono*: $[|a<b; c<d|] ==> a + c < b + (d::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
⟨*proof*⟩

lemma *add-less-le-mono*:
$[| \ a<b; c\leq d \ |] ==> a + c < b + (d::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
⟨*proof*⟩

lemma *add-le-less-mono*:
$[| \ a\leq b; c<d \ |] ==> a + c < b + (d::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
⟨*proof*⟩

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$ **shows** $a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le)$
⟨*proof*⟩

lemma *add-less-imp-less-right*:
$a + c < b + c ==> a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le)$
⟨*proof*⟩

lemma *add-less-cancel-left* [*simp*]:
$(c+a < c+b) = (a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
⟨*proof*⟩

lemma *add-less-cancel-right* [*simp*]:
$(a+c < b+c) = (a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
⟨*proof*⟩

lemma *add-le-cancel-left* [*simp*]:
$(c+a \leq c+b) = (a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
⟨*proof*⟩

lemma *add-le-cancel-right* [*simp*]:
$(a+c \leq b+c) = (a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
⟨*proof*⟩

lemma *add-le-imp-le-right*:
$a + c \leq b + c ==> a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le)$
⟨*proof*⟩

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*⟩

14.3 Ordering Rules for Unary Minus

lemma *le-imp-neg-le*:
assumes $a \leq (b::'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le,\ ab\text{-}group\text{-}add\})$ **shows** $-b \leq -a$
⟨*proof*⟩

lemma *neg-le-iff-le* [*simp*]: $(-b \leq -a) = (a \leq (b::'a::pordered\text{-}ab\text{-}group\text{-}add))$

⟨*proof*⟩

lemma *neg-le-0-iff-le* [*simp*]: $(-a \leq 0) = (0 \leq (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *neg-0-le-iff-le* [*simp*]: $(0 \leq -a) = (a \leq (0::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *neg-less-iff-less* [*simp*]: $(-b < -a) = (a < (b::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *neg-less-0-iff-less* [*simp*]: $(-a < 0) = (0 < (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *neg-0-less-iff-less* [*simp*]: $(0 < -a) = (a < (0::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $(a < -\ b) = (b < -\ (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *minus-less-iff*: $(-\ a < b) = (-\ b < (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *le-minus-iff*: $(a \leq -\ b) = (b \leq -\ (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *minus-le-iff*: $(-\ a \leq b) = (-\ b \leq (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *add-diff-eq*: $a + (b - c) = (a + b) - (c::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *diff-add-eq*: $(a - b) + c = (a + c) - (b::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *diff-eq-eq*: $(a-b = c) = (a = c + (b::'a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *eq-diff-eq*: $(a = c-b) = (a + (b::'a::ab\text{-}group\text{-}add) = c)$
⟨*proof*⟩

lemma *diff-diff-eq*: $(a - b) - c = a - (b + (c::'a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *diff-diff-eq2*: $a - (b - c) = (a + c) - (b::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *diff-add-cancel*: $a - b + b = (a::'a::ab\text{-}group\text{-}add)$

⟨*proof*⟩

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

Further subtraction laws

lemma *less-iff-diff-less-0*: $(a < b) = (a - b < (0::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *diff-less-eq*: $(a-b < c) = (a < c + (b::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *less-diff-eq*: $(a < c-b) = (a + (b::'a::pordered\text{-}ab\text{-}group\text{-}add) < c)$
⟨*proof*⟩

lemma *diff-le-eq*: $(a-b \leq c) = (a \leq c + (b::'a::pordered\text{-}ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *le-diff-eq*: $(a \leq c-b) = (a + (b::'a::pordered\text{-}ab\text{-}group\text{-}add) \leq c)$
⟨*proof*⟩

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 *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

14.4 Support for reasoning about signs

lemma *add-pos-pos*: $0 <$
 $(x::'a::\{comm\text{-}monoid\text{-}add,pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add\})$
  $\Longrightarrow 0 < y \Longrightarrow 0 < x + y$
⟨*proof*⟩

lemma *add-pos-nonneg*: $0 <$
 $(x::'a::\{comm\text{-}monoid\text{-}add,pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add\})$
  $\Longrightarrow 0 <= y \Longrightarrow 0 < x + y$
⟨*proof*⟩

lemma *add-nonneg-pos*: $0 <=$
 $(x::'a::\{comm\text{-}monoid\text{-}add,pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add\})$
  $\Longrightarrow 0 < y \Longrightarrow 0 < x + y$
⟨*proof*⟩

lemma *add-nonneg-nonneg*: $0 <=$
 $(x::'a::\{comm\text{-}monoid\text{-}add,pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add\})$
  $\Longrightarrow 0 <= y \Longrightarrow 0 <= x + y$

⟨*proof*⟩

lemma *add-neg-neg*: (*x*::'*a*::{*comm-monoid-add*,*pordered-cancel-ab-semigroup-add*})
  < *0* ==> *y* < *0* ==> *x* + *y* < *0*
⟨*proof*⟩

lemma *add-neg-nonpos*:
  (*x*::'*a*::{*comm-monoid-add*,*pordered-cancel-ab-semigroup-add*}) < *0*
   ==> *y* <= *0* ==> *x* + *y* < *0*
⟨*proof*⟩

lemma *add-nonpos-neg*:
  (*x*::'*a*::{*comm-monoid-add*,*pordered-cancel-ab-semigroup-add*}) <= *0*
   ==> *y* < *0* ==> *x* + *y* < *0*
⟨*proof*⟩

lemma *add-nonpos-nonpos*:
  (*x*::'*a*::{*comm-monoid-add*,*pordered-cancel-ab-semigroup-add*}) <= *0*
   ==> *y* <= *0* ==> *x* + *y* <= *0*
⟨*proof*⟩

14.5 Lemmas for the *cancel-numerals* simproc

lemma *eq-iff-diff-eq-0*: (*a* = *b*) = (*a*−*b* = (*0*::'*a*::*ab-group-add*))
⟨*proof*⟩

lemma *le-iff-diff-le-0*: (*a* ≤ *b*) = (*a*−*b* ≤ (*0*::'*a*::*pordered-ab-group-add*))
⟨*proof*⟩

14.6 Lattice Ordered (Abelian) Groups

axclass *lordered-ab-group-meet* < *pordered-ab-group-add*, *meet-semilorder*

axclass *lordered-ab-group-join* < *pordered-ab-group-add*, *join-semilorder*

lemma *add-meet-distrib-left*: *a* + (*meet b c*) = *meet* (*a* + *b*) (*a* + (*c*::'*a*::{*pordered-ab-group-add*, *meet-semilorder*}))
⟨*proof*⟩

lemma *add-join-distrib-left*: *a* + (*join b c*) = *join* (*a* + *b*) (*a*+ (*c*::'*a*::{*pordered-ab-group-add*, *join-semilorder*}))
⟨*proof*⟩

lemma *is-join-neg-meet*: *is-join* (% (*a*::'*a*::{*pordered-ab-group-add*, *meet-semilorder*}) *b*. − (*meet* (−*a*) (−*b*)))
⟨*proof*⟩

lemma *is-meet-neg-join*: *is-meet* (% (*a*::'*a*::{*pordered-ab-group-add*, *join-semilorder*}) *b*. − (*join* (−*a*) (−*b*)))
⟨*proof*⟩

axclass *lordered-ab-group* ⊆ *pordered-ab-group-add*, *lorder*

instance *lordered-ab-group-meet* ⊆ *lordered-ab-group*
⟨*proof*⟩

instance *lordered-ab-group-join* ⊆ *lordered-ab-group*
⟨*proof*⟩

lemma *add-join-distrib-right*: $(join\ a\ b) + (c::'a::lordered\text{-}ab\text{-}group) = join\ (a+c)\ (b+c)$
⟨*proof*⟩

lemma *add-meet-distrib-right*: $(meet\ a\ b) + (c::'a::lordered\text{-}ab\text{-}group) = meet\ (a+c)\ (b+c)$
⟨*proof*⟩

lemmas *add-meet-join-distribs* = *add-meet-distrib-right add-meet-distrib-left add-join-distrib-right add-join-distrib-left*

lemma *join-eq-neg-meet*: $join\ a\ (b::'a::lordered\text{-}ab\text{-}group) = -\ meet\ (-a)\ (-b)$
⟨*proof*⟩

lemma *meet-eq-neg-join*: $meet\ a\ (b::'a::lordered\text{-}ab\text{-}group) = -\ join\ (-a)\ (-b)$
⟨*proof*⟩

lemma *add-eq-meet-join*: $a + b = (join\ a\ b) + (meet\ a\ (b::'a::lordered\text{-}ab\text{-}group))$
⟨*proof*⟩

14.7 Positive Part, Negative Part, Absolute Value

constdefs
  $pprt :: 'a \Rightarrow ('a::lordered\text{-}ab\text{-}group)$
  $pprt\ x == join\ x\ 0$
  $nprt :: 'a \Rightarrow ('a::lordered\text{-}ab\text{-}group)$
  $nprt\ x == meet\ x\ 0$

lemma *prts*: $a = pprt\ a + nprt\ a$
⟨*proof*⟩

lemma *zero-le-pprt*[*simp*]: $0 \le pprt\ a$
⟨*proof*⟩

lemma *nprt-le-zero*[*simp*]: $nprt\ a \le 0$
⟨*proof*⟩

lemma *le-eq-neg*: $(a \le -b) = (a + b \le (0::\text{-}::lordered\text{-}ab\text{-}group))$ (**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*]: $0 <= x \Longrightarrow pprt\ x = x$
⟨*proof*⟩

lemma *nprt-eq-id*[*simp*]: $x <= 0 \Longrightarrow nprt\ x = x$
⟨*proof*⟩

lemma *pprt-eq-0*[*simp*]: $x <= 0 \Longrightarrow pprt\ x = 0$
⟨*proof*⟩

lemma *nprt-eq-0*[*simp*]: $0 <= x \Longrightarrow nprt\ x = 0$
⟨*proof*⟩

lemma *join-0-imp-0*: $join\ a\ (-a) = 0 \Longrightarrow a = (0::'a::lordered\text{-}ab\text{-}group)$
⟨*proof*⟩

lemma *meet-0-imp-0*: $meet\ a\ (-a) = 0 \Longrightarrow a = (0::'a::lordered\text{-}ab\text{-}group)$
⟨*proof*⟩

lemma *join-0-eq-0*[*simp*]: $(join\ a\ (-a) = 0) = (a = (0::'a::lordered\text{-}ab\text{-}group))$
⟨*proof*⟩

lemma *meet-0-eq-0*[*simp*]: $(meet\ a\ (-a) = 0) = (a = (0::'a::lordered\text{-}ab\text{-}group))$
⟨*proof*⟩

lemma *zero-le-double-add-iff-zero-le-single-add*[*simp*]: $(0 \leq a + a) = (0 \leq (a::'a::lordered\text{-}ab\text{-}group))$
⟨*proof*⟩

lemma *double-add-le-zero-iff-single-add-le-zero*[*simp*]: $(a + a <= 0) = ((a::'a::lordered\text{-}ab\text{-}group)$
$<= 0)$
⟨*proof*⟩

lemma *double-add-less-zero-iff-single-less-zero*[*simp*]: $(a+a<0) = ((a::'a::\{pordered\text{-}ab\text{-}group\text{-}add,linorder\})$
$< 0)$ (**is** *?s*)
⟨*proof*⟩

axclass *lordered-ab-group-abs* $\subseteq$ *lordered-ab-group*
abs-lattice: $abs\ x = join\ x\ (-x)$

lemma *abs-zero*[*simp*]: $abs\ 0 = (0::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
⟨*proof*⟩

lemma *abs-eq-0*[*simp*]: $(abs\ a = 0) = (a = (0::'a::lordered\text{-}ab\text{-}group\text{-}abs))$
⟨*proof*⟩

lemma *abs-0-eq*[*simp*]: $(0 = abs\ a) = (a = (0::'a::lordered\text{-}ab\text{-}group\text{-}abs))$
⟨*proof*⟩

lemma *neg-meet-eq-join*[*simp*]: $-$ *meet* a (b::-::*lordered-ab-group*) = *join* ($-a$) ($-b$)
⟨*proof*⟩

lemma *neg-join-eq-meet*[*simp*]: $-$ *join* a (b::-::*lordered-ab-group*) = *meet* ($-a$) ($-b$)
⟨*proof*⟩

lemma *join-eq-if*: *join* a ($-a$) = (*if* $a < 0$ *then* $-a$ *else* (a::'a::{*lordered-ab-group*, *linorder*}))
⟨*proof*⟩

lemma *abs-if-lattice*: $|a|$ = (*if* $a < 0$ *then* $-a$ *else* (a::'a::{*lordered-ab-group-abs*, *linorder*}))
⟨*proof*⟩

lemma *abs-ge-zero*[*simp*]: $0 \leq$ *abs* (a::'a::*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *abs-le-zero-iff* [*simp*]: (*abs* $a \leq$ (0::'a::*lordered-ab-group-abs*)) = ($a = 0$)
⟨*proof*⟩

lemma *zero-less-abs-iff* [*simp*]: ($0 <$ *abs* a) = ($a \neq$ (0::'a::*lordered-ab-group-abs*))
⟨*proof*⟩

lemma *abs-not-less-zero* [*simp*]: ~ *abs* $a <$ (0::'a::*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *abs-ge-self*: $a \leq$ *abs* (a::'a::*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *abs-ge-minus-self*: $-a \leq$ *abs* (a::'a::*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *le-imp-join-eq*: $a \leq b \Longrightarrow$ *join* a $b = b$
⟨*proof*⟩

lemma *ge-imp-join-eq*: $b \leq a \Longrightarrow$ *join* a $b = a$
⟨*proof*⟩

lemma *le-imp-meet-eq*: $a \leq b \Longrightarrow$ *meet* a $b = a$
⟨*proof*⟩

lemma *ge-imp-meet-eq*: $b \leq a \Longrightarrow$ *meet* a $b = b$
⟨*proof*⟩

lemma *abs-prts*: *abs* (a::-::*lordered-ab-group-abs*) = *pprt* a $-$ *nprt* a
⟨*proof*⟩

lemma *abs-minus-cancel* [*simp*]: *abs* $(-a)$ = *abs*$(a::'a::$*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *abs-idempotent* [*simp*]: *abs* (*abs* a) = *abs* $(a::'a::$*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *abs-minus-commute*:
fixes a :: $'a::$*lordered-ab-group-abs*
shows *abs* $(a-b)$ = *abs*$(b-a)$
⟨*proof*⟩

lemma *zero-le-iff-zero-nprt*: $(0 \leq a) = ($*nprt* $a = 0)$
⟨*proof*⟩

lemma *le-zero-iff-zero-pprt*: $(a \leq 0) = ($*pprt* $a = 0)$
⟨*proof*⟩

lemma *le-zero-iff-pprt-id*: $(0 \leq a) = ($*pprt* $a = a)$
⟨*proof*⟩

lemma *zero-le-iff-nprt-id*: $(a \leq 0) = ($*nprt* $a = a)$
⟨*proof*⟩

lemma *pprt-mono*[*simp*]: (a::-::*lordered-ab-group*) <= $b \Longrightarrow$ *pprt* a <= *pprt* b
⟨*proof*⟩

lemma *nprt-mono*[*simp*]: (a::-::*lordered-ab-group*) <= $b \Longrightarrow$ *nprt* a <= *nprt* b
⟨*proof*⟩

lemma *iff2imp*: $(A=B) \Longrightarrow (A \Longrightarrow B)$
⟨*proof*⟩

lemma *abs-of-nonneg* [*simp*]: $0 \leq a \Longrightarrow$ *abs* $a = (a::'a::$*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *abs-of-pos*: $0 < (x::'a::$*lordered-ab-group-abs*) ==> *abs* $x = x$
⟨*proof*⟩

lemma *abs-of-nonpos* [*simp*]: $a \leq 0 \Longrightarrow$ *abs* $a = -(a::'a::$*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *abs-of-neg*: $(x::'a::$*lordered-ab-group-abs*) < 0 ==>
abs $x = -x$
⟨*proof*⟩

lemma *abs-leI*: $[|a \leq b; -a \leq b|]$ ==> *abs* $a \leq (b::'a::$*lordered-ab-group-abs*)
⟨*proof*⟩

lemma *le-minus-self-iff*: $(a \leq -a) = (a \leq (0::'a::lordered\text{-}ab\text{-}group))$
⟨*proof*⟩

lemma *minus-le-self-iff*: $(-a \leq a) = (0 \leq (a::'a::lordered\text{-}ab\text{-}group))$
⟨*proof*⟩

lemma *abs-le-D1*: $abs\ a \leq b \Longrightarrow a \leq (b::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
⟨*proof*⟩

lemma *abs-le-D2*: $abs\ a \leq b \Longrightarrow -a \leq (b::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
⟨*proof*⟩

lemma *abs-le-iff*: $(abs\ a \leq b) = (a \leq b\ \&\ -a \leq (b::'a::lordered\text{-}ab\text{-}group\text{-}abs))$
⟨*proof*⟩

lemma *abs-triangle-ineq*: $abs(a+b) \leq abs\ a + abs(b::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
⟨*proof*⟩

lemma *abs-triangle-ineq2*: $abs\ (a::'a::lordered\text{-}ab\text{-}group\text{-}abs) -$
$abs\ b <= abs\ (a - b)$
⟨*proof*⟩

lemma *abs-triangle-ineq3*:
$abs(abs\ (a::'a::lordered\text{-}ab\text{-}group\text{-}abs) - abs\ b) <= abs\ (a - b)$
⟨*proof*⟩

lemma *abs-triangle-ineq4*: $abs\ ((a::'a::lordered\text{-}ab\text{-}group\text{-}abs) - b) <=$
$abs\ a + abs\ b$
⟨*proof*⟩

lemma *abs-diff-triangle-ineq*:
$|(a::'a::lordered\text{-}ab\text{-}group\text{-}abs) + b - (c+d)| \leq |a-c| + |b-d|$
⟨*proof*⟩

lemma *abs-add-abs*[*simp*]:
fixes $a::\ 'a::\{lordered\text{-}ab\text{-}group\text{-}abs\}$
shows $abs(abs\ a + abs\ b) = abs\ a + abs\ b$ (**is** $?L = ?R$)
⟨*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 *minus-add-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*⟩

lemmas *group-eq-simps* =
mult-ac
add-ac
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq

lemma *estimate-by-abs*:
$a + b <= (c{::}'a{::}lordered\text{-}ab\text{-}group\text{-}abs) \Longrightarrow a <= c + abs\ b$
⟨*proof*⟩

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

lemmas *diff-less-0-iff-less* = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-eq-0-iff-eq* = *eq-iff-diff-eq-0* [*symmetric*]
lemmas *diff-le-0-iff-le* = *le-iff-diff-le-0* [*symmetric*]
declare *diff-less-0-iff-less* [*simp*]
declare *diff-eq-0-iff-eq* [*simp*]
declare *diff-le-0-iff-le* [*simp*]

⟨*ML*⟩

end

15 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.

axclass *semiring* $\subseteq$ *ab-semigroup-add*, *semigroup-mult*
left-distrib: $(a + b) * c = a * c + b * c$
right-distrib: $a * (b + c) = a * b + a * c$

axclass *semiring-0* $\subseteq$ *semiring*, *comm-monoid-add*

axclass *semiring-0-cancel* $\subseteq$ *semiring-0*, *cancel-ab-semigroup-add*

axclass *comm-semiring* $\subseteq$ *ab-semigroup-add*, *ab-semigroup-mult*
distrib: $(a + b) * c = a * c + b * c$

instance *comm-semiring* $\subseteq$ *semiring*
⟨*proof*⟩

axclass *comm-semiring-0* $\subseteq$ *comm-semiring*, *comm-monoid-add*

instance *comm-semiring-0* $\subseteq$ *semiring-0* ⟨*proof*⟩

axclass *comm-semiring-0-cancel* $\subseteq$ *comm-semiring-0*, *cancel-ab-semigroup-add*

instance *comm-semiring-0-cancel* $\subseteq$ *semiring-0-cancel* ⟨*proof*⟩

axclass *axclass-0-neq-1* $\subseteq$ *zero*, *one*
zero-neq-one [*simp*]: $0 \neq 1$

axclass *semiring-1* $\subseteq$ *axclass-0-neq-1*, *semiring-0*, *monoid-mult*

axclass *comm-semiring-1* $\subseteq$ *axclass-0-neq-1*, *comm-semiring-0*, *comm-monoid-mult*

instance *comm-semiring-1* $\subseteq$ *semiring-1* ⟨*proof*⟩

axclass *axclass-no-zero-divisors* $\subseteq$ *zero*, *times*
no-zero-divisors: $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$

axclass *semiring-1-cancel* $\subseteq$ *semiring-1*, *cancel-ab-semigroup-add*

instance *semiring-1-cancel* $\subseteq$ *semiring-0-cancel* $\langle proof \rangle$

axclass *comm-semiring-1-cancel* $\subseteq$ *comm-semiring-1*, *cancel-ab-semigroup-add*

instance *comm-semiring-1-cancel* $\subseteq$ *semiring-1-cancel* $\langle proof \rangle$

instance *comm-semiring-1-cancel* $\subseteq$ *comm-semiring-0-cancel* $\langle proof \rangle$

axclass *ring* $\subseteq$ *semiring*, *ab-group-add*

instance *ring* $\subseteq$ *semiring-0-cancel* $\langle proof \rangle$

axclass *comm-ring* $\subseteq$ *comm-semiring-0*, *ab-group-add*

instance *comm-ring* $\subseteq$ *ring* $\langle proof \rangle$

instance *comm-ring* $\subseteq$ *comm-semiring-0-cancel* $\langle proof \rangle$

axclass *ring-1* $\subseteq$ *ring*, *semiring-1*

instance *ring-1* $\subseteq$ *semiring-1-cancel* $\langle proof \rangle$

axclass *comm-ring-1* $\subseteq$ *comm-ring*, *comm-semiring-1*

instance *comm-ring-1* $\subseteq$ *ring-1* $\langle proof \rangle$

instance *comm-ring-1* $\subseteq$ *comm-semiring-1-cancel* $\langle proof \rangle$

axclass *idom* $\subseteq$ *comm-ring-1*, *axclass-no-zero-divisors*

axclass *field* $\subseteq$ *comm-ring-1*, *inverse*
left-inverse [*simp*]: $a \neq 0 ==> inverse\ a * a = 1$
divide-inverse: $a\ /\ b = a * inverse\ b$

lemma *mult-zero-left* [*simp*]: $0 * a = (0::'a::semiring\text{-}0\text{-}cancel)$
$\langle proof \rangle$

lemma *mult-zero-right* [*simp*]: $a * 0 = (0::'a::semiring\text{-}0\text{-}cancel)$
$\langle proof \rangle$

lemma *field-mult-eq-0-iff* [*simp*]: $(a*b = (0::'a::field)) = (a = 0 \mid b = 0)$
$\langle proof \rangle$

instance *field* $\subseteq$ *idom*
⟨*proof*⟩

axclass *division-by-zero* $\subseteq$ *zero*, *inverse*
inverse-zero [*simp*]: *inverse 0* = *0*

15.1 Distribution rules

theorems *ring-distrib* = *right-distrib left-distrib*

For the *combine-numerals* simproc

lemma *combine-common-factor*:
$a*e + (b*e + c) = (a+b)*e + (c::'a::semiring)$
⟨*proof*⟩

lemma *minus-mult-left*: $-(a * b) = (-a) * (b::'a::ring)$
⟨*proof*⟩

lemma *minus-mult-right*: $-(a * b) = a * -(b::'a::ring)$
⟨*proof*⟩

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

lemma *minus-mult-commute*: $(- a) * b = a * (- b::'a::ring)$
⟨*proof*⟩

lemma *right-diff-distrib*: $a * (b - c) = a * b - a * (c::'a::ring)$
⟨*proof*⟩

lemma *left-diff-distrib*: $(a - b) * c = a * c - b * (c::'a::ring)$
⟨*proof*⟩

axclass *pordered-semiring* $\subseteq$ *semiring-0*, *pordered-ab-semigroup-add*
mult-left-mono: $a <= b \Longrightarrow 0 <= c \Longrightarrow c * a <= c * b$
mult-right-mono: $a <= b \Longrightarrow 0 <= c \Longrightarrow a * c <= b * c$

axclass *pordered-cancel-semiring* $\subseteq$ *pordered-semiring*, *cancel-ab-semigroup-add*

instance *pordered-cancel-semiring* $\subseteq$ *semiring-0-cancel* ⟨*proof*⟩

axclass *ordered-semiring-strict* $\subseteq$ *semiring-0*, *ordered-cancel-ab-semigroup-add*
mult-strict-left-mono: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
mult-strict-right-mono: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$

instance *ordered-semiring-strict* $\subseteq$ *semiring-0-cancel* ⟨*proof*⟩

instance *ordered-semiring-strict* $\subseteq$ *pordered-cancel-semiring*
⟨*proof*⟩

axclass *pordered-comm-semiring* $\subseteq$ *comm-semiring-0*, *pordered-ab-semigroup-add*
  mult-mono: $a <= b \Longrightarrow 0 <= c \Longrightarrow c * a <= c * b$

axclass *pordered-cancel-comm-semiring* $\subseteq$ *pordered-comm-semiring*, *cancel-ab-semigroup-add*

instance *pordered-cancel-comm-semiring* $\subseteq$ *pordered-comm-semiring* ⟨*proof*⟩

axclass *ordered-comm-semiring-strict* $\subseteq$ *comm-semiring-0*, *ordered-cancel-ab-semigroup-add*
  mult-strict-mono: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$

instance *pordered-comm-semiring* $\subseteq$ *pordered-semiring*
⟨*proof*⟩

instance *pordered-cancel-comm-semiring* $\subseteq$ *pordered-cancel-semiring* ⟨*proof*⟩

instance *ordered-comm-semiring-strict* $\subseteq$ *ordered-semiring-strict*
⟨*proof*⟩

instance *ordered-comm-semiring-strict* $\subseteq$ *pordered-cancel-comm-semiring*
⟨*proof*⟩

axclass *pordered-ring* $\subseteq$ *ring*, *pordered-semiring*

instance *pordered-ring* $\subseteq$ *pordered-ab-group-add* ⟨*proof*⟩

instance *pordered-ring* $\subseteq$ *pordered-cancel-semiring* ⟨*proof*⟩

axclass *lordered-ring* $\subseteq$ *pordered-ring*, *lordered-ab-group-abs*

instance *lordered-ring* $\subseteq$ *lordered-ab-group-meet* ⟨*proof*⟩

instance *lordered-ring* $\subseteq$ *lordered-ab-group-join* ⟨*proof*⟩

axclass *axclass-abs-if* $\subseteq$ *minus*, *ord*, *zero*
  abs-if: $abs\ a = (if\ (a < 0)\ then\ (-a)\ else\ a)$

axclass *ordered-ring-strict* $\subseteq$ *ring*, *ordered-semiring-strict*, *axclass-abs-if*

instance *ordered-ring-strict* $\subseteq$ *lordered-ab-group* ⟨*proof*⟩

instance *ordered-ring-strict* $\subseteq$ *lordered-ring*
⟨*proof*⟩

axclass *pordered-comm-ring* $\subseteq$ *comm-ring*, *pordered-comm-semiring*

axclass *ordered-semidom* $\subseteq$ *comm-semiring-1-cancel*, *ordered-comm-semiring-strict*

  zero-less-one [*simp*]: $0 < 1$

axclass *ordered-idom* ⊆ *comm-ring-1*, *ordered-comm-semiring-strict*, *axclass-abs-if*

instance *ordered-idom* ⊆ *ordered-ring-strict* ⟨*proof*⟩

axclass *ordered-field* ⊆ *field*, *ordered-idom*

lemmas *linorder-neqE-ordered-idom* =
linorder-neqE[**where** *'a* = *?'b::ordered-idom*]

lemma *eq-add-iff1*:
$(a*e + c = b*e + d) = ((a-b)*e + c = (d::'a::ring))$
⟨*proof*⟩

lemma *eq-add-iff2*:
$(a*e + c = b*e + d) = (c = (b-a)*e + (d::'a::ring))$
⟨*proof*⟩

lemma *less-add-iff1*:
$(a*e + c < b*e + d) = ((a-b)*e + c < (d::'a::pordered\text{-}ring))$
⟨*proof*⟩

lemma *less-add-iff2*:
$(a*e + c < b*e + d) = (c < (b-a)*e + (d::'a::pordered\text{-}ring))$
⟨*proof*⟩

lemma *le-add-iff1*:
$(a*e + c \le b*e + d) = ((a-b)*e + c \le (d::'a::pordered\text{-}ring))$
⟨*proof*⟩

lemma *le-add-iff2*:
$(a*e + c \le b*e + d) = (c \le (b-a)*e + (d::'a::pordered\text{-}ring))$
⟨*proof*⟩

15.2 Ordering Rules for Multiplication

lemma *mult-left-le-imp-le*:
$[|c*a \le c*b;\ 0 < c|] ==> a \le (b::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *mult-right-le-imp-le*:
$[|a*c \le b*c;\ 0 < c|] ==> a \le (b::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *mult-left-less-imp-less*:
$[|c*a < c*b;\ 0 \le c|] ==> a < (b::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *mult-right-less-imp-less*:
$[|a*c < b*c;\ 0 \le c|] \Longrightarrow a < (b::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *mult-strict-left-mono-neg*:
$[|b < a;\ c < 0|] \Longrightarrow c * a < c * (b::'a::ordered\text{-}ring\text{-}strict)$
⟨*proof*⟩

lemma *mult-left-mono-neg*:
$[|b \le a;\ c \le 0|] \Longrightarrow c * a \le c * (b::'a::pordered\text{-}ring)$
⟨*proof*⟩

lemma *mult-strict-right-mono-neg*:
$[|b < a;\ c < 0|] \Longrightarrow a * c < b * (c::'a::ordered\text{-}ring\text{-}strict)$
⟨*proof*⟩

lemma *mult-right-mono-neg*:
$[|b \le a;\ c \le 0|] \Longrightarrow a * c \le (b::'a::pordered\text{-}ring) * c$
⟨*proof*⟩

15.3 Products of Signs

lemma *mult-pos-pos*: $[|\ (0::'a::ordered\text{-}semiring\text{-}strict) < a;\ 0 < b\ |] \Longrightarrow 0 < a*b$
⟨*proof*⟩

lemma *mult-nonneg-nonneg*: $[|\ (0::'a::pordered\text{-}cancel\text{-}semiring) \le a;\ 0 \le b\ |] \Longrightarrow 0 \le a*b$
⟨*proof*⟩

lemma *mult-pos-neg*: $[|\ (0::'a::ordered\text{-}semiring\text{-}strict) < a;\ b < 0\ |] \Longrightarrow a*b < 0$
⟨*proof*⟩

lemma *mult-nonneg-nonpos*: $[|\ (0::'a::pordered\text{-}cancel\text{-}semiring) \le a;\ b \le 0\ |] \Longrightarrow a*b \le 0$
⟨*proof*⟩

lemma *mult-pos-neg2*: $[|\ (0::'a::ordered\text{-}semiring\text{-}strict) < a;\ b < 0\ |] \Longrightarrow b*a < 0$
⟨*proof*⟩

lemma *mult-nonneg-nonpos2*: $[|\ (0::'a::pordered\text{-}cancel\text{-}semiring) \le a;\ b \le 0\ |] \Longrightarrow b*a \le 0$
⟨*proof*⟩

lemma *mult-neg-neg*: $[|\ a < (0::'a::ordered\text{-}ring\text{-}strict);\ b < 0\ |] \Longrightarrow 0 < a*b$
⟨*proof*⟩

lemma *mult-nonpos-nonpos*: $[| \ a \le (0::'a::pordered\text{-}ring); \ b \le 0 \ |] ==> 0 \le a*b$
⟨*proof*⟩

lemma *zero-less-mult-pos*:
$[| \ 0 < a*b; \ 0 < a|] ==> 0 < (b::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *zero-less-mult-pos2*:
$[| \ 0 < b*a; \ 0 < a|] ==> 0 < (b::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *zero-less-mult-iff*:
$((0::'a::ordered\text{-}ring\text{-}strict) < a*b) = (0 < a \ \& \ 0 < b \ | \ a < 0 \ \& \ b < 0)$
⟨*proof*⟩

A field has no "zero divisors", and this theorem holds without the assumption of an ordering. See *field-mult-eq-0-iff* below.

lemma *mult-eq-0-iff* [*simp*]: $(a*b = (0::'a::ordered\text{-}ring\text{-}strict)) = (a = 0 \ | \ b = 0)$
⟨*proof*⟩

lemma *zero-le-mult-iff*:
$((0::'a::ordered\text{-}ring\text{-}strict) \le a*b) = (0 \le a \ \& \ 0 \le b \ | \ a \le 0 \ \& \ b \le 0)$
⟨*proof*⟩

lemma *mult-less-0-iff*:
$(a*b < (0::'a::ordered\text{-}ring\text{-}strict)) = (0 < a \ \& \ b < 0 \ | \ a < 0 \ \& \ 0 < b)$
⟨*proof*⟩

lemma *mult-le-0-iff*:
$(a*b \le (0::'a::ordered\text{-}ring\text{-}strict)) = (0 \le a \ \& \ b \le 0 \ | \ a \le 0 \ \& \ 0 \le b)$
⟨*proof*⟩

lemma *split-mult-pos-le*: $(0 \le a \ \& \ 0 \le b) \ | \ (a \le 0 \ \& \ b \le 0) \Longrightarrow 0 \le a * (b::\text{-}::pordered\text{-}ring)$
⟨*proof*⟩

lemma *split-mult-neg-le*: $(0 \le a \ \& \ b \le 0) \ | \ (a \le 0 \ \& \ 0 \le b) \Longrightarrow a * b \le (0::\text{-}::pordered\text{-}cancel\text{-}semiring)$
⟨*proof*⟩

lemma *zero-le-square*: $(0::'a::ordered\text{-}ring\text{-}strict) \le a*a$
⟨*proof*⟩

Proving axiom *zero-less-one* makes all *ordered-semidom* theorems available to members of *ordered-idom*

instance *ordered-idom* ⊆ *ordered-semidom*
⟨*proof*⟩

instance *ordered-ring-strict* $\subseteq$ *axclass-no-zero-divisors*
⟨*proof*⟩

instance *ordered-idom* $\subseteq$ *idom* ⟨*proof*⟩

All three types of comparision involving 0 and 1 are covered.

lemmas *one-neq-zero* = *zero-neq-one* [*THEN not-sym*]
declare *one-neq-zero* [*simp*]

lemma *zero-le-one* [*simp*]: $(0::'a::ordered\text{-}semidom) \leq 1$
⟨*proof*⟩

lemma *not-one-le-zero* [*simp*]: $\sim (1::'a::ordered\text{-}semidom) \leq 0$
⟨*proof*⟩

lemma *not-one-less-zero* [*simp*]: $\sim (1::'a::ordered\text{-}semidom) < 0$
⟨*proof*⟩

15.4 More Monotonicity

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
$[|a<b;\ c<d;\ 0<b;\ 0\leq c|] ==> a * c < b * (d::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
$[|\ a<b;\ c<d;\ 0 \leq a;\ 0 \leq c|] ==> a * c < b * (d::'a::ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *mult-mono*:
$[|a \leq b;\ c \leq d;\ 0 \leq b;\ 0 \leq c|]$
$==> a * c \ \leq\ b * (d::'a::pordered\text{-}semiring)$
⟨*proof*⟩

lemma *less-1-mult*: $[|\ 1 < m;\ 1 < n\ |] ==> 1 < m*(n::'a::ordered\text{-}semidom)$
⟨*proof*⟩

lemma *mult-less-le-imp-less*: $(a::'a::ordered\text{-}semiring\text{-}strict) < b ==>$
$c <= d ==> 0 <= a ==> 0 < c ==> a * c < b * d$
⟨*proof*⟩

lemma *mult-le-less-imp-less*: $(a::'a::ordered\text{-}semiring\text{-}strict) <= b ==>$
$c < d ==> 0 < a ==> 0 <= c ==> a * c < b * d$
⟨*proof*⟩

15.5 Cancellation Laws for Relationships With a Common Factor

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations $\leq$ and equality.

These "disjunction" versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

lemma *mult-less-cancel-right-disj*:
  $(a{*}c < b{*}c) = ((0 < c \ \& \ a < b) \mid (c < 0 \ \& \ b < (a{::}'a{::}ordered\text{-}ring\text{-}strict)))$
⟨*proof*⟩

lemma *mult-less-cancel-left-disj*:
  $(c{*}a < c{*}b) = ((0 < c \ \& \ a < b) \mid (c < 0 \ \& \ b < (a{::}'a{::}ordered\text{-}ring\text{-}strict)))$
⟨*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*:
 fixes $c :: 'a :: ordered\text{-}ring\text{-}strict$
 shows $(a{*}c < b{*}c) = ((0 \leq c \dashrightarrow a < b) \ \& \ (c \leq 0 \dashrightarrow b < a))$
⟨*proof*⟩

lemma *mult-less-cancel-left*:
 fixes $c :: 'a :: ordered\text{-}ring\text{-}strict$
 shows $(c{*}a < c{*}b) = ((0 \leq c \dashrightarrow a < b) \ \& \ (c \leq 0 \dashrightarrow b < a))$
⟨*proof*⟩

lemma *mult-le-cancel-right*:
  $(a{*}c \leq b{*}c) = ((0{<}c \dashrightarrow a{\leq}b) \ \& \ (c{<}0 \dashrightarrow b \leq (a{::}'a{::}ordered\text{-}ring\text{-}strict)))$
⟨*proof*⟩

lemma *mult-le-cancel-left*:
  $(c{*}a \leq c{*}b) = ((0{<}c \dashrightarrow a{\leq}b) \ \& \ (c{<}0 \dashrightarrow b \leq (a{::}'a{::}ordered\text{-}ring\text{-}strict)))$
⟨*proof*⟩

lemma *mult-less-imp-less-left*:
  assumes *less*: $c{*}a < c{*}b$ **and** *nonneg*: $0 \leq c$
  shows $a < (b{::}'a{::}ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

lemma *mult-less-imp-less-right*:
 assumes *less*: $a{*}c < b{*}c$ **and** *nonneg*: $0 <= c$
 shows $a < (b{::}'a{::}ordered\text{-}semiring\text{-}strict)$
⟨*proof*⟩

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*]:

$(a*c = b*c) = (c = (0::'a::ordered\text{-}ring\text{-}strict) \mid a=b)$
⟨proof⟩

These cancellation theorems require an ordering. Versions are proved below that work for fields without an ordering.

lemma *mult-cancel-left* [*simp*]:
$(c*a = c*b) = (c = (0::'a::ordered\text{-}ring\text{-}strict) \mid a=b)$
⟨proof⟩

15.5.1 Special Cancellation Simprules for Multiplication

These also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c \le b*c) = ((0<c \dashrightarrow 1\le b)\ \&\ (c<0 \dashrightarrow b \le 1))$
⟨proof⟩

lemma *mult-le-cancel-right2*:
fixes c :: $'a$ :: *ordered-idom*
shows $(a*c \le c) = ((0<c \dashrightarrow a\le 1)\ \&\ (c<0 \dashrightarrow 1 \le a))$
⟨proof⟩

lemma *mult-le-cancel-left1*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c \le c*b) = ((0<c \dashrightarrow 1\le b)\ \&\ (c<0 \dashrightarrow b \le 1))$
⟨proof⟩

lemma *mult-le-cancel-left2*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c*a \le c) = ((0<c \dashrightarrow a\le 1)\ \&\ (c<0 \dashrightarrow 1 \le a))$
⟨proof⟩

lemma *mult-less-cancel-right1*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c < b*c) = ((0 \le c \dashrightarrow 1<b)\ \&\ (c \le 0 \dashrightarrow b < 1))$
⟨proof⟩

lemma *mult-less-cancel-right2*:
fixes c :: $'a$ :: *ordered-idom*
shows $(a*c < c) = ((0 \le c \dashrightarrow a<1)\ \&\ (c \le 0 \dashrightarrow 1 < a))$
⟨proof⟩

lemma *mult-less-cancel-left1*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c < c*b) = ((0 \le c \dashrightarrow 1<b)\ \&\ (c \le 0 \dashrightarrow b < 1))$
⟨proof⟩

lemma *mult-less-cancel-left2*:

fixes *c* :: *'a* :: *ordered-idom*
shows $(c{*}a < c) = ((0 \leq c \dashrightarrow a{<}1)$ & $(c \leq 0 \dashrightarrow 1 < a))$
⟨*proof*⟩

lemma *mult-cancel-right1* [*simp*]:
fixes *c* :: *'a* :: *ordered-idom*
shows $(c = b{*}c) = (c = 0 \mid b{=}1)$
⟨*proof*⟩

lemma *mult-cancel-right2* [*simp*]:
fixes *c* :: *'a* :: *ordered-idom*
shows $(a{*}c = c) = (c = 0 \mid a{=}1)$
⟨*proof*⟩

lemma *mult-cancel-left1* [*simp*]:
fixes *c* :: *'a* :: *ordered-idom*
shows $(c = c{*}b) = (c = 0 \mid b{=}1)$
⟨*proof*⟩

lemma *mult-cancel-left2* [*simp*]:
fixes *c* :: *'a* :: *ordered-idom*
shows $(c{*}a = c) = (c = 0 \mid a{=}1)$
⟨*proof*⟩

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

This list of rewrites decides ring equalities by ordered rewriting.

lemmas *ring-eq-simps* =

left-distrib right-distrib left-diff-distrib right-diff-distrib
group-eq-simps

15.6 Fields

lemma *right-inverse* [*simp*]:
assumes *not0*: $a \neq 0$ **shows** $a * inverse\ (a{::}'a{::}field) = 1$
⟨*proof*⟩

lemma *right-inverse-eq*: $b \neq 0 \implies (a\ /\ b = 1) = (a = (b{::}'a{::}field))$

⟨*proof*⟩

lemma *nonzero-inverse-eq-divide*: $a \neq 0 \implies inverse\ (a::'a::field) = 1/a$
⟨*proof*⟩

lemma *divide-self*: $a \neq 0 \implies a\ /\ (a::'a::field) = 1$
⟨*proof*⟩

lemma *divide-zero* [*simp*]: $a\ /\ 0 = (0::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *divide-self-if* [*simp*]:
$a\ /\ (a::'a::\{field,division\text{-}by\text{-}zero\}) = (if\ a{=}0\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *divide-zero-left* [*simp*]: $0/a = (0::'a::field)$
⟨*proof*⟩

lemma *inverse-eq-divide*: $inverse\ (a::'a::field) = 1/a$
⟨*proof*⟩

lemma *add-divide-distrib*: $(a+b)/(c::'a::field) = a/c + b/c$
⟨*proof*⟩

Compared with *mult-eq-0-iff*, this version removes the requirement of an ordering.

lemma *field-mult-eq-0-iff* [*simp*]: $(a*b = (0::'a::field)) = (a = 0 \mid b = 0)$
⟨*proof*⟩

Cancellation of equalities with a common factor

lemma *field-mult-cancel-right-lemma*:
assumes *cnz*: $c \neq (0::'a::field)$
and *eq*: $a*c = b*c$
shows $a{=}b$
⟨*proof*⟩

lemma *field-mult-cancel-right* [*simp*]:
$(a*c = b*c) = (c = (0::'a::field) \mid a{=}b)$
⟨*proof*⟩

lemma *field-mult-cancel-left* [*simp*]:
$(c*a = c*b) = (c = (0::'a::field) \mid a{=}b)$
⟨*proof*⟩

lemma *nonzero-imp-inverse-nonzero*: $a \neq 0 \implies inverse\ a \neq (0::'a::field)$
⟨*proof*⟩

15.7 Basic Properties of *inverse*

lemma *inverse-zero-imp-zero*: *inverse* $a = 0 ==> a = (0::'a::field)$
⟨*proof*⟩

lemma *inverse-nonzero-imp-nonzero*:
inverse $a = 0 ==> a = (0::'a::field)$
⟨*proof*⟩

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
(*inverse* $a = 0) = (a = (0::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *nonzero-inverse-minus-eq*:
assumes [*simp*]: $a \neq 0$ **shows** *inverse*$(-a) = -$*inverse*$(a::'a::field)$
⟨*proof*⟩

lemma *inverse-minus-eq* [*simp*]:
inverse$(-a) = -$*inverse*$(a::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *nonzero-inverse-eq-imp-eq*:
assumes *inveq*: *inverse* $a =$ *inverse* b
and *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows $a = (b::'a::field)$
⟨*proof*⟩

lemma *inverse-eq-imp-eq*:
inverse $a =$ *inverse* $b ==> a = (b::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *inverse-eq-iff-eq* [*simp*]:
(*inverse* $a =$ *inverse* $b) = (a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *nonzero-inverse-inverse-eq*:
assumes [*simp*]: $a \neq 0$ **shows** *inverse*(*inverse* $(a::'a::field)) = a$
⟨*proof*⟩

lemma *inverse-inverse-eq* [*simp*]:
inverse(*inverse* $(a::'a::\{field,division\text{-}by\text{-}zero\})) = a$
⟨*proof*⟩

lemma *inverse-1* [*simp*]: *inverse* $1 = (1::'a::field)$
⟨*proof*⟩

lemma *inverse-unique*:
assumes *ab*: $a*b = 1$
shows *inverse* $a = (b::'a::field)$

⟨*proof*⟩

lemma *nonzero-inverse-mult-distrib*:
assumes *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows $inverse(a*b) = inverse(b) * inverse(a::'a::field)$
⟨*proof*⟩

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
$inverse(a*b) = inverse(a) * inverse(b::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

There is no slick version using division by zero.

lemma *inverse-add*:
$[|a \neq 0;\ \ b \neq 0|]$
$==> inverse\ a + inverse\ b = (a+b) * inverse\ a * inverse\ (b::'a::field)$
⟨*proof*⟩

lemma *inverse-divide* [*simp*]:
$inverse\ (a/b) = b\ /\ (a::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

15.8 Calculations with fractions

lemma *nonzero-mult-divide-cancel-left*:
assumes [*simp*]: $b \neq 0$ **and** [*simp*]: $c \neq 0$
shows $(c*a)/(c*b) = a/(b::'a::field)$
⟨*proof*⟩

lemma *mult-divide-cancel-left*:
$c \neq 0 ==> (c*a)\ /\ (c*b) = a\ /\ (b::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *nonzero-mult-divide-cancel-right*:
$[|b \neq 0;\ c \neq 0|] ==> (a*c)\ /\ (b*c) = a/(b::'a::field)$
⟨*proof*⟩

lemma *mult-divide-cancel-right*:
$c \neq 0 ==> (a*c)\ /\ (b*c) = a\ /\ (b::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *mult-divide-cancel-eq-if*:
$(c*a)\ /\ (c*b) =$
$(if\ c=0\ then\ 0\ else\ a\ /\ (b::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*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*⟩

lemma *divide-divide-eq-right* [*simp*]:
$a / (b/c) = (a*c) / (b::'a::\{field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *divide-divide-eq-left* [*simp*]:
$(a / b) / (c::'a::\{field,division\text{-}by\text{-}zero\}) = a / (b*c)$
⟨*proof*⟩

lemma *add-frac-eq*: $(y::'a::field)$ ~= 0 ==> z ~= 0 ==>
$x / y + w / z = (x * z + w * y) / (y * z)$
⟨*proof*⟩

15.8.1 Special Cancellation Simprules for Division

lemma *mult-divide-cancel-left-if* [*simp*]:
fixes $c :: 'a :: \{field,division\text{-}by\text{-}zero\}$
shows $(c*a) / (c*b) = (if\ c=0\ then\ 0\ else\ a/b)$
⟨*proof*⟩

lemma *mult-divide-cancel-right-if* [*simp*]:
fixes $c :: 'a :: \{field,division\text{-}by\text{-}zero\}$
shows $(a*c) / (b*c) = (if\ c=0\ then\ 0\ else\ a/b)$
⟨*proof*⟩

lemma *mult-divide-cancel-left-if1* [*simp*]:
fixes $c :: 'a :: \{field,division\text{-}by\text{-}zero\}$
shows $c / (c*b) = (if\ c=0\ then\ 0\ else\ 1/b)$
⟨*proof*⟩

lemma *mult-divide-cancel-left-if2* [*simp*]:
fixes $c :: 'a :: \{field,division\text{-}by\text{-}zero\}$
shows $(c*a) / c = (if\ c=0\ then\ 0\ else\ a)$
⟨*proof*⟩

lemma *mult-divide-cancel-right-if1* [*simp*]:
fixes $c :: 'a :: \{field,division\text{-}by\text{-}zero\}$
shows $c / (b*c) = (if\ c=0\ then\ 0\ else\ 1/b)$
⟨*proof*⟩

lemma *mult-divide-cancel-right-if2* [*simp*]:

fixes c :: $'a$ :: {*field*,*division-by-zero*}
shows $(a*c) \ / \ c = (\textit{if } c{=}0 \textit{ then } 0 \textit{ else } a)$
⟨*proof*⟩

Two lemmas for cancelling the denominator

lemma *times-divide-self-right* [*simp*]:
fixes a :: $'a$ :: {*field*,*division-by-zero*}
shows $a * (b/a) = (\textit{if } a{=}0 \textit{ then } 0 \textit{ else } b)$
⟨*proof*⟩

lemma *times-divide-self-left* [*simp*]:
fixes a :: $'a$ :: {*field*,*division-by-zero*}
shows $(b/a) * a = (\textit{if } a{=}0 \textit{ then } 0 \textit{ else } b)$
⟨*proof*⟩

15.9 Division and Unary Minus

lemma *nonzero-minus-divide-left*: $b \neq 0 \Longrightarrow - (a/b) = (-a) \ / \ (b{::}'a{::}\textit{field})$
⟨*proof*⟩

lemma *nonzero-minus-divide-right*: $b \neq 0 \Longrightarrow - (a/b) = a \ / \ -(b{::}'a{::}\textit{field})$
⟨*proof*⟩

lemma *nonzero-minus-divide-divide*: $b \neq 0 \Longrightarrow (-a)/(-b) = a \ / \ (b{::}'a{::}\textit{field})$
⟨*proof*⟩

lemma *minus-divide-left*: $- (a/b) = (-a) \ / \ (b{::}'a{::}\textit{field})$
⟨*proof*⟩

lemma *minus-divide-right*: $- (a/b) = a \ / \ -(b{::}'a{::}\{\textit{field},\textit{division-by-zero}\})$
⟨*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*]:
$(-a)/(-b) = a \ / \ (b{::}'a{::}\{\textit{field},\textit{division-by-zero}\})$
⟨*proof*⟩

lemma *diff-divide-distrib*: $(a-b)/(c{::}'a{::}\textit{field}) = a/c - b/c$
⟨*proof*⟩

lemma *diff-frac-eq*: $(y::'a::field)$ ~= 0 ==> z ~= 0 ==>
  x / y − w / z = (x ∗ z − w ∗ y) / (y ∗ z)
⟨*proof*⟩

15.10 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*:
  $a < 0$ ==> *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 [*simp*]: $a \neq 0$
  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 [*simp*]: $a \neq 0$
  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*⟩

15.11 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*]:
$[|0 < a;\ 0 < b|]$
$==> (inverse\ a < inverse\ b) = (b < (a{::}'a{::}ordered\text{-}field))$
⟨*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*]:
$[|0 < a;\ 0 < b|]$
$==> (inverse\ a \leq inverse\ b) = (b \leq (a{::}'a{::}ordered\text{-}field))$
⟨*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*:
$[|inverse\ a < inverse\ b;\ b < 0|] ==> b < (a{::}'a{::}ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-less-iff-less-neg* [*simp*]:
$[|a < 0;\ b < 0|]$
$==> (inverse\ a < inverse\ b) = (b < (a{::}'a{::}ordered\text{-}field))$
⟨*proof*⟩

lemma *le-imp-inverse-le-neg*:
$[|a \leq b;\ b < 0|] ==> inverse\ b \leq inverse\ (a{::}'a{::}ordered\text{-}field)$
⟨*proof*⟩

lemma *inverse-le-iff-le-neg* [*simp*]:
$[|a < 0;\ b < 0|]$
$\Longrightarrow (inverse\ a \le inverse\ b) = (b \le (a::'a::ordered\text{-}field))$
⟨*proof*⟩

15.12 Inverses and the Number One

lemma *one-less-inverse-iff*:
$(1 < inverse\ x) = (0 < x\ \&\ x < (1::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$⟨*proof*⟩

lemma *inverse-eq-1-iff* [*simp*]:
$(inverse\ x = 1) = (x = (1::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *one-le-inverse-iff*:
$(1 \le inverse\ x) = (0 < x\ \&\ x \le (1::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *inverse-less-1-iff*:
$(inverse\ x < 1) = (x \le 0\ |\ 1 < (x::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *inverse-le-1-iff*:
$(inverse\ x \le 1) = (x \le 0\ |\ 1 \le (x::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

15.13 Simplification of Inequalities Involving Literal Divisors

lemma *pos-le-divide-eq*: $0 < (c::'a::ordered\text{-}field) \Longrightarrow (a \le b/c) = (a*c \le b)$
⟨*proof*⟩

lemma *neg-le-divide-eq*: $c < (0::'a::ordered\text{-}field) \Longrightarrow (a \le b/c) = (b \le a*c)$
⟨*proof*⟩

lemma *le-divide-eq*:
$(a \le b/c) =$
$(if\ 0 < c\ then\ a*c \le b$
$\quad else\ if\ c < 0\ then\ b \le a*c$
$\quad else\ \ a \le (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *pos-divide-le-eq*: $0 < (c::'a::ordered\text{-}field) \Longrightarrow (b/c \le a) = (b \le a*c)$
⟨*proof*⟩

lemma *neg-divide-le-eq*: $c < (0::'a::ordered\text{-}field) \Longrightarrow (b/c \le a) = (a*c \le b)$
⟨*proof*⟩

lemma *divide-le-eq*:
$(b/c \le a) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ b \le a{*}c$
$\quad \mathit{else\ if}\ c < 0\ \mathit{then}\ a{*}c \le b$
$\quad \mathit{else}\ 0 \le (a{::}'a{::}\{\mathit{ordered\text{-}field},\mathit{division\text{-}by\text{-}zero}\}))$
⟨*proof*⟩

lemma *pos-less-divide-eq*:
$0 < (c{::}'a{::}\mathit{ordered\text{-}field}) \Longrightarrow (a < b/c) = (a{*}c < b)$
⟨*proof*⟩

lemma *neg-less-divide-eq*:
$c < (0{::}'a{::}\mathit{ordered\text{-}field}) \Longrightarrow (a < b/c) = (b < a{*}c)$
⟨*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*:
$0 < (c{::}'a{::}\mathit{ordered\text{-}field}) \Longrightarrow (b/c < a) = (b < a{*}c)$
⟨*proof*⟩

lemma *neg-divide-less-eq*:
$c < (0{::}'a{::}\mathit{ordered\text{-}field}) \Longrightarrow (b/c < a) = (a{*}c < b)$
⟨*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*⟩

lemma *nonzero-eq-divide-eq*: $c \neq 0 \Longrightarrow ((a{::}'a{::}\mathit{field}) = b/c) = (a{*}c = b)$
⟨*proof*⟩

lemma *eq-divide-eq*:
$((a{::}'a{::}\{\mathit{field},\mathit{division\text{-}by\text{-}zero}\}) = b/c) = (\mathit{if}\ c \neq 0\ \mathit{then}\ a{*}c = b\ \mathit{else}\ a{=}0)$
⟨*proof*⟩

lemma *nonzero-divide-eq-eq*: $c \neq 0 \Longrightarrow (b/c = (a{::}'a{::}\mathit{field})) = (b = a{*}c)$
⟨*proof*⟩

lemma *divide-eq-eq*:
$(b/c = (a{::}'a{::}\{\mathit{field},\mathit{division\text{-}by\text{-}zero}\})) = (\mathit{if}\ c \neq 0\ \mathit{then}\ b = a{*}c\ \mathit{else}\ a{=}0)$

⟨*proof*⟩

lemma *divide-eq-imp*: $(c::'a::\{division\text{-}by\text{-}zero,field\})$ ~= 0 ==>
  $b = a * c$ ==> $b / c = a$
⟨*proof*⟩

lemma *eq-divide-imp*: $(c::'a::\{division\text{-}by\text{-}zero,field\})$ ~= 0 ==>
  $a * c = b$ ==> $a = b / c$
⟨*proof*⟩

lemma *frac-eq-eq*: $(y::'a::field)$ ~= 0 ==> z ~= 0 ==>
  $(x / y = w / z) = (x * z = w * y)$
⟨*proof*⟩

15.14 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*:
  $((0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}) \le a/b) =$
  $(0 \le a$ & $0 \le b \mid a \le 0$ & $b \le 0)$
⟨*proof*⟩

lemma *divide-le-0-iff*:
  $(a/b \le (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) =$
  $(0 \le a$ & $b \le 0 \mid a \le 0$ & $0 \le b)$
⟨*proof*⟩

lemma *divide-eq-0-iff* [*simp*]:
  $(a/b = 0) = (a{=}0 \mid b{=}(0::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *divide-pos-pos*: $0 < (x::'a::ordered\text{-}field)$ ==>
  $0 < y$ ==> $0 < x / y$
⟨*proof*⟩

lemma *divide-nonneg-pos*: 0 <= $(x::'a::ordered\text{-}field)$ ==> $0 < y$ ==>
  0 <= x / y
⟨*proof*⟩

lemma *divide-neg-pos*: $(x::'a::ordered\text{-}field) < 0$ ==> $0 < y$ ==> $x / y < 0$

⟨*proof*⟩

lemma *divide-nonpos-pos*: $(x::'a::ordered\text{-}field) <= 0 ==>$
$0 < y ==> x \;/\; y <= 0$
⟨*proof*⟩

lemma *divide-pos-neg*: $0 < (x::'a::ordered\text{-}field) ==> y < 0 ==> x \;/\; y < 0$
⟨*proof*⟩

lemma *divide-nonneg-neg*: $0 <= (x::'a::ordered\text{-}field) ==>$
$y < 0 ==> x \;/\; y <= 0$
⟨*proof*⟩

lemma *divide-neg-neg*: $(x::'a::ordered\text{-}field) < 0 ==> y < 0 ==> 0 < x \;/\; y$
⟨*proof*⟩

lemma *divide-nonpos-neg*: $(x::'a::ordered\text{-}field) <= 0 ==> y < 0 ==>$
$0 <= x \;/\; y$
⟨*proof*⟩

15.15 Cancellation Laws for Division

lemma *divide-cancel-right* [*simp*]:
$(a/c = b/c) = (c = 0 \mid a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *divide-cancel-left* [*simp*]:
$(c/a = c/b) = (c = 0 \mid a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

15.16 Division and the Number One

Simplify expressions equated with 1

lemma *divide-eq-1-iff* [*simp*]:
$(a/b = 1) = (b \neq 0 \;\&\; a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *one-eq-divide-iff* [*simp*]:
$(1 = a/b) = (b \neq 0 \;\&\; a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
⟨*proof*⟩

lemma *zero-eq-1-divide-iff* [*simp*]:
$((0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}) = 1/a) = (a = 0)$
⟨*proof*⟩

lemma *one-divide-eq-0-iff* [*simp*]:
$(1/a = (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) = (a = 0)$
⟨*proof*⟩

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemmas *zero-less-divide-1-iff* = *zero-less-divide-iff* [*of 1*]
lemmas *divide-less-0-1-iff* = *divide-less-0-iff* [*of 1*]
lemmas *zero-le-divide-1-iff* = *zero-le-divide-iff* [*of 1*]
lemmas *divide-le-0-1-iff* = *divide-le-0-iff* [*of 1*]

declare *zero-less-divide-1-iff* [*simp*]
declare *divide-less-0-1-iff* [*simp*]
declare *zero-le-divide-1-iff* [*simp*]
declare *divide-le-0-1-iff* [*simp*]

15.17 Ordering Rules for Division

lemma *divide-strict-right-mono*:
$[|a < b;\ 0 < c|] ==> a\ /\ c < b\ /\ (c::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *divide-right-mono*:
$[|a \leq b;\ 0 \leq c|] ==> a/c \leq b/(c::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})$
⟨*proof*⟩

lemma *divide-right-mono-neg*: $(a::'a::\{division\text{-}by\text{-}zero,ordered\text{-}field\}) <= b$
$==> c <= 0 ==> b\ /\ c <= a\ /\ c$
⟨*proof*⟩

lemma *divide-strict-right-mono-neg*:
$[|b < a;\ c < 0|] ==> a\ /\ c < b\ /\ (c::'a::ordered\text{-}field)$
⟨*proof*⟩

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
$[|b < a;\ 0 < c;\ 0 < a*b|] ==> c\ /\ a < c\ /\ (b::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *divide-left-mono*:
$[|b \leq a;\ 0 \leq c;\ 0 < a*b|] ==> c\ /\ a \leq c\ /\ (b::'a::ordered\text{-}field)$
⟨*proof*⟩

lemma *divide-left-mono-neg*: $(a::'a::\{division\text{-}by\text{-}zero,ordered\text{-}field\}) <= b$
$==> c <= 0 ==> 0 < a * b ==> c\ /\ a <= c\ /\ b$
⟨*proof*⟩

lemma *divide-strict-left-mono-neg*:
$[|a < b;\ c < 0;\ 0 < a*b|] ==> c\ /\ a < c\ /\ (b::'a::ordered\text{-}field)$
⟨*proof*⟩

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1*:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$

shows $(1 \le b \ / \ a) = ((0 < a \ \& \ a \le b) \mid (a < 0 \ \& \ b \le a))$
⟨*proof*⟩

lemma *divide-le-eq-1*:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $(b \ / \ a \le 1) = ((0 < a \ \& \ b \le a) \mid (a < 0 \ \& \ a \le b) \mid a{=}0)$
⟨*proof*⟩

lemma *less-divide-eq-1*:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $(1 < b \ / \ a) = ((0 < a \ \& \ a < b) \mid (a < 0 \ \& \ b < a))$
⟨*proof*⟩

lemma *divide-less-eq-1*:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $(b \ / \ a < 1) = ((0 < a \ \& \ b < a) \mid (a < 0 \ \& \ a < b) \mid a{=}0)$
⟨*proof*⟩

15.18 Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*]:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $0 < a \Longrightarrow (1 \le b \ / \ a) = (a \le b)$
⟨*proof*⟩

lemma *le-divide-eq-1-neg* [*simp*]:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $a < 0 \Longrightarrow (1 \le b \ / \ a) = (b \le a)$
⟨*proof*⟩

lemma *divide-le-eq-1-pos* [*simp*]:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $0 < a \Longrightarrow (b \ / \ a \le 1) = (b \le a)$
⟨*proof*⟩

lemma *divide-le-eq-1-neg* [*simp*]:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $a < 0 \Longrightarrow (b \ / \ a \le 1) = (a \le b)$
⟨*proof*⟩

lemma *less-divide-eq-1-pos* [*simp*]:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $0 < a \Longrightarrow (1 < b \ / \ a) = (a < b)$
⟨*proof*⟩

lemma *less-divide-eq-1-neg* [*simp*]:
fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
shows $a < 0 \Longrightarrow (1 < b \ / \ a) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-pos* [*simp*]:
  fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
  shows $0 < a \Longrightarrow (b\ /\ a < 1) = (b < a)$
⟨*proof*⟩

lemma *eq-divide-eq-1* [*simp*]:
  fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
  shows $(1 = b\ /\ a) = ((a \neq 0\ \&\ a = b))$
⟨*proof*⟩

lemma *divide-eq-eq-1* [*simp*]:
  fixes a :: $'a$ :: {*ordered-field*,*division-by-zero*}
  shows $(b\ /\ a = 1) = ((a \neq 0\ \&\ a = b))$
⟨*proof*⟩

15.19 Reasoning about inequalities with division

lemma *mult-right-le-one-le*: $0 <= (x::'a::ordered\text{-}idom) ==> 0 <= y ==> y <=$
1
    $==> x * y <= x$
  ⟨*proof*⟩

lemma *mult-left-le-one-le*: $0 <= (x::'a::ordered\text{-}idom) ==> 0 <= y ==> y <=$
1
    $==> y * x <= x$
  ⟨*proof*⟩

lemma *mult-imp-div-pos-le*: $0 < (y::'a::ordered\text{-}field) ==> x <= z * y ==>$
    $x\ /\ y <= z$
  ⟨*proof*⟩

lemma *mult-imp-le-div-pos*: $0 < (y::'a::ordered\text{-}field) ==> z * y <= x ==>$
    $z <= x\ /\ y$
  ⟨*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*⟩

lemmas *times-divide-eq* = *times-divide-eq-right times-divide-eq-left*

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*]

15.20 Ordered Fields are Dense

lemma *less-add-one*: $a < (a+1::'a::ordered\text{-}semidom)$
⟨*proof*⟩

lemma *zero-less-two*: $0 < (1+1::'a::ordered\text{-}semidom)$
⟨*proof*⟩

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*⟩

lemma *dense*: $a < b ==> \exists\, r::'a::ordered\text{-}field.\ a < r \;\&\; r < b$
⟨*proof*⟩

15.21 Absolute Value

lemma *abs-one* [*simp*]: $abs\ 1 = (1::'a::ordered\text{-}idom)$
⟨*proof*⟩

lemma *abs-le-mult*: $abs\ (a * b) \le (abs\ a) * (abs\ (b::'a::lordered\text{-}ring))$
⟨*proof*⟩

lemma *abs-eq-mult*:
assumes $(0 \le a \lor a \le 0) \land (0 \le b \lor b \le 0)$
shows $abs\ (a*b) = abs\ a * abs\ (b::'a::lordered\text{-}ring)$
⟨*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*:
$a \neq 0 ==> abs\ (inverse\ (a::'a::ordered\text{-}field)) = inverse\ (abs\ a)$
⟨*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*:
$b \neq 0 ==> abs\ (a\ /\ (b::'a::ordered\text{-}field)) = abs\ a\ /\ abs\ b$
⟨*proof*⟩

lemma *abs-divide* [*simp*]:
$abs\ (a\ /\ (b::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) = abs\ a\ /\ abs\ b$
⟨*proof*⟩

lemma *abs-mult-less*:
$[|\ abs\ a < c;\ abs\ b < d\ |] ==> abs\ a * abs\ b < c*(d::'a::ordered\text{-}idom)$
⟨*proof*⟩

lemma *eq-minus-self-iff*: $(a = -a) = (a = (0::'a::ordered\text{-}idom))$
⟨*proof*⟩

lemma *less-minus-self-iff*: $(a < -a) = (a < (0::'a::ordered\text{-}idom))$
⟨*proof*⟩

lemma *abs-less-iff*: $(abs\ a < b) = (a < b\ \&\ -a < (b::'a::ordered\text{-}idom))$
⟨*proof*⟩

lemma *abs-mult-pos*: $(0::'a::ordered\text{-}idom) <= x ==>$
$(abs\ y) * x = abs\ (y * x)$
⟨*proof*⟩

lemma *abs-div-pos*: $(0::'a::\{division\text{-}by\text{-}zero,ordered\text{-}field\}) < y ==>$
$abs\ x\ /\ y = abs\ (x\ /\ y)$
⟨*proof*⟩

15.22 Miscellaneous

lemma *linprog-dual-estimate*:
assumes
$A * x \leq (b::'a::lordered\text{-}ring)$
$0 \leq y$
$abs\ (A - A') \leq \delta A$
$b \leq b'$
$abs\ (c - c') \leq \delta c$
$abs\ x \leq r$

shows
$c * x \le y * b' + (y * \delta A + abs\ (y * A' - c') + \delta c) * r$
⟨*proof*⟩

lemma *le-ge-imp-abs-diff-1*:
assumes
$A1 <= (A::'a::lordered\text{-}ring)$
$A <= A2$
shows $abs\ (A-A1) <= A2-A1$
⟨*proof*⟩

lemma *mult-le-prts*:
assumes
$a1 <= (a::'a::lordered\text{-}ring)$
$a <= a2$
$b1 <= b$
$b <= b2$
shows
$a * b <= pprt\ a2 * pprt\ b2 + pprt\ a1 * nprt\ b2 + nprt\ a2 * pprt\ b1 + nprt\ a1 * nprt\ b1$
⟨*proof*⟩

lemma *mult-le-dual-prts*:
assumes
$A * x \le (b::'a::lordered\text{-}ring)$
$0 \le y$
$A1 \le A$
$A \le A2$
$c1 \le c$
$c \le c2$
$r1 \le x$
$x \le r2$
shows
$c * x \le y * b + (let\ s1 = c1 - y * A2;\ s2 = c2 - y * A1\ in\ pprt\ s2 * pprt\ r2 + pprt\ s1 * nprt\ r2 + nprt\ s2 * pprt\ r1 + nprt\ s1 * nprt\ r1)$
(**is** - <= - + ?C)
⟨*proof*⟩

⟨*ML*⟩

end

16 Nat: Natural numbers

theory *Nat*
imports *Wellfounded-Recursion Ring-and-Field*
begin

16.1 Type *ind*

typedecl *ind*

consts
Zero-Rep :: *ind*
Suc-Rep :: *ind => ind*

axioms
— the axiom of infinity in 2 parts
inj-Suc-Rep: *inj Suc-Rep*
Suc-Rep-not-Zero-Rep: *Suc-Rep x* $\neq$ *Zero-Rep*
finalconsts
Zero-Rep
Suc-Rep

16.2 Type nat

Type definition

consts
Nat :: *ind set*

inductive *Nat*
intros
Zero-RepI: *Zero-Rep* : *Nat*
Suc-RepI: *i* : *Nat* ==> *Suc-Rep i* : *Nat*

global

typedef (**open** *Nat*)
nat = *Nat* ⟨*proof*⟩

instance *nat* :: {*ord*, *zero*, *one*} ⟨*proof*⟩

Abstract constants and syntax

consts
Suc :: *nat => nat*
pred-nat :: (*nat* $*$ *nat*) *set*

local

defs
Zero-nat-def: *0* == *Abs-Nat Zero-Rep*
Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))
One-nat-def [*simp*]: *1* == *Suc 0*

— nat operations
pred-nat-def: *pred-nat* == {(*m*, *n*). *n* = *Suc m*}

less-def: $m < n == (m, n) : trancl\ pred\text{-}nat$

le-def: $m \leq (n::nat) == \sim (n < m)$

Induction

theorem *nat-induct*: $P\ 0 ==> (!!n.\ P\ n ==> P\ (Suc\ n)) ==> P\ n$
⟨*proof*⟩

Distinctness of constructors

lemma *Suc-not-Zero* [*iff*]: $Suc\ m \neq 0$
⟨*proof*⟩

lemma *Zero-not-Suc* [*iff*]: $0 \neq Suc\ m$
⟨*proof*⟩

lemma *Suc-neq-Zero*: $Suc\ m = 0 ==> R$
⟨*proof*⟩

lemma *Zero-neq-Suc*: $0 = Suc\ m ==> R$
⟨*proof*⟩

Injectiveness of *Suc*

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
⟨*proof*⟩

lemma *Suc-inject*: $Suc\ x = Suc\ y ==> x = y$
⟨*proof*⟩

lemma *Suc-Suc-eq* [*iff*]: $(Suc\ m = Suc\ n) = (m = n)$
⟨*proof*⟩

lemma *nat-not-singleton*: $(\forall x.\ x = (0::nat)) = False$
⟨*proof*⟩

nat is a datatype

rep-datatype *nat*
distinct *Suc-not-Zero Zero-not-Suc*
inject *Suc-Suc-eq*
induction *nat-induct*

lemma *n-not-Suc-n*: $n \neq Suc\ n$
⟨*proof*⟩

lemma *Suc-n-not-n*: $Suc\ t \neq t$
⟨*proof*⟩

A special form of induction for reasoning about $m < n$ and $m - n$

theorem *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*⟩

16.3 Basic properties of "less than"

lemma *wf-pred-nat*: *wf pred-nat*
⟨*proof*⟩

lemma *wf-less*: *wf* {(x, y::*nat*). $x < y$}
⟨*proof*⟩

lemma *less-eq*: ((m, n) : *pred-nat*^+) = ($m < n$)
⟨*proof*⟩

16.3.1 Introduction properties

lemma *less-trans*: $i < j$ ==> $j < k$ ==> $i <$ (k::*nat*)
⟨*proof*⟩

lemma *lessI* [*iff*]: $n <$ *Suc* n
⟨*proof*⟩

lemma *less-SucI*: $i < j$ ==> $i <$ *Suc* j
⟨*proof*⟩

lemma *zero-less-Suc* [*iff*]: *0* < *Suc* n
⟨*proof*⟩

16.3.2 Elimination properties

lemma *less-not-sym*: $n < m$ ==> ~ $m <$ (n::*nat*)
⟨*proof*⟩

lemma *less-asym*:
assumes *h1*: (n::*nat*) $< m$ **and** *h2*: ~ P ==> $m < n$ **shows** P
⟨*proof*⟩

lemma *less-not-refl*: ~ $n <$ (n::*nat*)
⟨*proof*⟩

lemma *less-irrefl* [*elim!*]: (n::*nat*) $< n$ ==> R
⟨*proof*⟩

lemma *less-not-refl2*: $n < m$ ==> $m \neq$ (n::*nat*) ⟨*proof*⟩

lemma *less-not-refl3*: (s::*nat*) $< t$ ==> $s \neq t$
⟨*proof*⟩

lemma *lessE*:
assumes *major*: $i < k$

and *p1*: $k = Suc\ i ==> P$ **and** *p2*: $!!j.\ i < j ==> k = Suc\ j ==> P$
shows P
⟨*proof*⟩

lemma *not-less0* [*iff*]: $\sim\ n < (0::nat)$
⟨*proof*⟩

lemma *less-zeroE*: $(n::nat) < 0 ==> R$
⟨*proof*⟩

lemma *less-SucE*: **assumes** *major*: $m < Suc\ n$
and *less*: $m < n ==> P$ **and** *eq*: $m = n ==> P$ **shows** P
⟨*proof*⟩

lemma *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
⟨*proof*⟩

lemma *less-one* [*iff*]: $(n < (1::nat)) = (n = 0)$
⟨*proof*⟩

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
⟨*proof*⟩

lemma *Suc-mono*: $m < n ==> Suc\ m < Suc\ n$
⟨*proof*⟩

"Less than" is a linear ordering

lemma *less-linear*: $m < n \mid m = n \mid n < (m::nat)$
⟨*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*: $(m::nat) < n ==> P\ n\ m$
and *eqCase*: $m = n ==> P\ n\ m$ **and** *lessCase*: $n{<}m ==> P\ n\ m$
shows $P\ n\ m$
⟨*proof*⟩

16.3.3 Inductive (?) properties

lemma *Suc-lessI*: $m < n ==> Suc\ m \neq n ==> Suc\ m < n$
⟨*proof*⟩

lemma *Suc-lessD*: $Suc\ m < n ==> m < n$
⟨*proof*⟩

lemma *Suc-lessE*: **assumes** *major*: $Suc\ i < k$
and *minor*: $!!j.\ i < j ==> k = Suc\ j ==> P$ **shows** P
⟨*proof*⟩

lemma *Suc-less-SucD*: $Suc\ m < Suc\ n ==> m < n$
⟨*proof*⟩

lemma *Suc-less-eq* [*iff*, *code*]: $(Suc\ m < Suc\ n) = (m < n)$
⟨*proof*⟩

lemma *less-trans-Suc*:
assumes *le*: $i < j$ **shows** $j < k ==> Suc\ i < k$
⟨*proof*⟩

lemma [*code*]: $((n::nat) < 0) = False$ ⟨*proof*⟩
lemma [*code*]: $(0 < Suc\ n) = True$ ⟨*proof*⟩

Can be used with *less-Suc-eq* to get $n = m \lor n < m$

lemma *not-less-eq*: $(\sim\ m < n) = (n < Suc\ m)$
⟨*proof*⟩

Complete induction, aka course-of-values induction

lemma *nat-less-induct*:
assumes *prem*: $!!n.\ \forall m::nat.\ m < n\ {-}{-}{>}\ P\ m ==> P\ n$ **shows** $P\ n$
⟨*proof*⟩

lemmas *less-induct* = *nat-less-induct* [*rule-format*, *case-names less*]

16.4 Properties of "less than or equal"

Was *le-eq-less-Suc*, but this orientation is more useful

lemma *less-Suc-eq-le*: $(m < Suc\ n) = (m \leq n)$
⟨*proof*⟩

lemma *le-imp-less-Suc*: $m \leq n ==> m < Suc\ n$
⟨*proof*⟩

lemma *le0* [*iff*]: $(0::nat) \leq n$
⟨*proof*⟩

lemma *Suc-n-not-le-n*: $\sim\ Suc\ n \leq n$
⟨*proof*⟩

lemma *le-0-eq* [*iff*]: $((i::nat) \leq 0) = (i = 0)$
⟨*proof*⟩

lemma *le-Suc-eq*: $(m \leq Suc\ n) = (m \leq n \mid m = Suc\ n)$

$\langle proof \rangle$

lemma *le-SucE*: $m \leq Suc\ n ==> (m \leq n ==> R) ==> (m = Suc\ n ==> R) ==> R$
$\langle proof \rangle$

lemma *Suc-leI*: $m < n ==> Suc(m) \leq n$
$\langle proof \rangle$

lemma *Suc-leD*: $Suc(m) \leq n ==> m \leq n$
$\langle proof \rangle$

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $Suc\ m \leq n ==> m < n$
$\langle proof \rangle$

lemma *Suc-le-eq*: $(Suc\ m \leq n) = (m < n)$
$\langle proof \rangle$

lemma *le-SucI*: $m \leq n ==> m \leq Suc\ n$
$\langle proof \rangle$

lemma *less-imp-le*: $m < n ==> m \leq (n::nat)$
$\langle proof \rangle$

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *le-imp-less-or-eq*: $m \leq n ==> m < n \mid m = (n::nat)$
$\langle proof \rangle$

lemma *less-or-eq-imp-le*: $m < n \mid m = n ==> m \leq (n::nat)$
$\langle proof \rangle$

lemma *le-eq-less-or-eq*: $(m \leq (n::nat)) = (m < n \mid m{=}n)$
$\langle proof \rangle$

Useful with *Blast*.

lemma *eq-imp-le*: $(m::nat) = n ==> m \leq n$
$\langle proof \rangle$

lemma *le-refl*: $n \leq (n::nat)$
$\langle proof \rangle$

lemma *le-less-trans*: $[\mid i \leq j;\ j < k \mid] ==> i < (k::nat)$
$\langle proof \rangle$

lemma *less-le-trans*: $[| \ i < j; j \le k \ |] ==> i < (k::nat)$
⟨*proof*⟩

lemma *le-trans*: $[| \ i \le j; j \le k \ |] ==> i \le (k::nat)$
⟨*proof*⟩

lemma *le-anti-sym*: $[| \ m \le n; n \le m \ |] ==> m = (n::nat)$
⟨*proof*⟩

lemma *Suc-le-mono* [*iff*]: $(Suc\ n \le Suc\ m) = (n \le m)$
⟨*proof*⟩

Axiom *order-less-le* of class *order*:

lemma *nat-less-le*: $((m::nat) < n) = (m \le n \ \& \ m \neq n)$
⟨*proof*⟩

lemma *le-neq-implies-less*: $(m::nat) \le n ==> m \neq n ==> m < n$
⟨*proof*⟩

Axiom *linorder-linear* of class *linorder*:

lemma *nat-le-linear*: $(m::nat) \le n \mid n \le m$
⟨*proof*⟩

Type @typ nat is a wellfounded linear order

instance *nat* :: {*order*, *linorder*, *wellorder*}
⟨*proof*⟩

lemmas *linorder-neqE-nat* = *linorder-neqE*[**where** *'a* = *nat*]

lemma *not-less-less-Suc-eq*: $\sim n < m ==> (n < Suc\ m) = (n = m)$
⟨*proof*⟩

Rewrite $n < Suc\ m$ to $n = m$ if $\neg\ n < m$ or $m \le n$ hold. Not suitable as default simprules because they often lead to looping

lemma *le-less-Suc-eq*: $m \le n ==> (n < Suc\ m) = (n = m)$
⟨*proof*⟩

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

Re-orientation of the equations $0 = x$ and $1 = x$. No longer added as simprules (they loop) but via *reorient-simproc* in Bin

Polymorphic, not just for *nat*

lemma *zero-reorient*: $(0 = x) = (x = 0)$
⟨*proof*⟩

lemma *one-reorient*: $(1 = x) = (x = 1)$
⟨*proof*⟩

16.5 Arithmetic operators

axclass *power* < *type*

consts
power :: (*'a::power*) => *nat* => *'a* (**infixr** ^ *80*)

arithmetic operators + − and ∗

instance *nat* :: {*plus*, *minus*, *times*, *power*} ⟨*proof*⟩

size of a datatype value; overloaded

consts *size* :: *'a* => *nat*

primrec
add-0: *0* + *n* = *n*
add-Suc: *Suc m* + *n* = *Suc* (*m* + *n*)

primrec
diff-0: *m* − *0* = *m*
diff-Suc: *m* − *Suc n* = (*case m* − *n of 0* => *0* | *Suc k* => *k*)

primrec
mult-0: *0* ∗ *n* = *0*
mult-Suc: *Suc m* ∗ *n* = *n* + (*m* ∗ *n*)

These two rules ease the use of primitive recursion. NOTE USE OF ==

lemma *def-nat-rec-0*: (!!*n*. *f n* == *nat-rec c h n*) ==> *f 0* = *c*
⟨*proof*⟩

lemma *def-nat-rec-Suc*: (!!*n*. *f n* == *nat-rec c h n*) ==> *f* (*Suc n*) = *h n* (*f n*)
⟨*proof*⟩

lemma *not0-implies-Suc*: *n* ≠ *0* ==> ∃ *m*. *n* = *Suc m*
⟨*proof*⟩

lemma *gr-implies-not0*: !!*n*::*nat*. *m*<*n* ==> *n* ≠ *0*
⟨*proof*⟩

lemma *neq0-conv* [*iff*]: !!*n*::*nat*. (*n* ≠ *0*) = (*0* < *n*)
⟨*proof*⟩

This theorem is useful with *blast*

lemma *gr0I*: ((*n*::*nat*) = *0* ==> *False*) ==> *0* < *n*
⟨*proof*⟩

lemma *gr0-conv-Suc*: (*0* < *n*) = (∃ *m*. *n* = *Suc m*)
⟨*proof*⟩

lemma *not-gr0* [*iff*]: !!*n*::*nat*. (~ (*0* < *n*)) = (*n* = *0*)

⟨*proof*⟩

lemma *Suc-le-D*: $(Suc\ n \leq m') ==> (?\ m.\ m' = Suc\ m)$
⟨*proof*⟩

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $(m < Suc\ n) = (m = 0 \mid (\exists j.\ m = Suc\ j\ \&\ j < n))$
⟨*proof*⟩

lemma *nat-induct2*: $[|P\ 0;\ P\ (Suc\ 0);\ !!k.\ P\ k ==> P\ (Suc\ (Suc\ k))|] ==> P\ n$
⟨*proof*⟩

16.6 *LEAST* theorems for type *nat*

lemma *Least-Suc*:
$[|\ P\ n;\ \sim\ P\ 0\ |] ==> (LEAST\ n.\ P\ n) = Suc\ (LEAST\ m.\ P(Suc\ m))$
⟨*proof*⟩

lemma *Least-Suc2*:
$[|P\ n;\ Q\ m;\ \sim P\ 0;\ !k.\ P\ (Suc\ k) = Q\ k|] ==> Least\ P = Suc\ (Least\ Q)$
⟨*proof*⟩

16.7 *min* and *max*

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 *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*⟩

16.8 Basic rewrite rules for the arithmetic operators

Difference

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*⟩

Could be (and is, below) generalized in various ways However, none of the generalizations are currently in the simpset, and I dread to think what happens if I put them in

lemma *Suc-pred* [*simp*]: $0 < n \Longrightarrow Suc\ (n - Suc\ 0) = n$
⟨*proof*⟩

declare *diff-Suc* [*simp del*, *code del*]

16.9 Addition

lemma *add-0-right* [*simp*]: $m + 0 = (m::nat)$
⟨*proof*⟩

lemma *add-Suc-right* [*simp*]: $m + Suc\ n = Suc\ (m + n)$
⟨*proof*⟩

lemma [*code*]: $Suc\ m + n = m + Suc\ n$ ⟨*proof*⟩

Associative law for addition

lemma *nat-add-assoc*: $(m + n) + k = m + ((n + k)::nat)$
⟨*proof*⟩

Commutative law for addition

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*⟩

lemma *nat-add-left-cancel-le* [*simp*]: $(k + m \leq k + n) = (m{\leq}(n::nat))$
⟨*proof*⟩

lemma *nat-add-left-cancel-less* [*simp*]: $(k + m < k + n) = (m{<}(n::nat))$
⟨*proof*⟩

Reasoning about $m + 0 = 0$, etc.

lemma *add-is-0* [*iff*]: $!!m::nat.\ (m + n = 0) = (m = 0\ \&\ n = 0)$
⟨*proof*⟩

lemma *add-is-1*: $(m{+}n{=}\ Suc\ 0) = (m{=}\ Suc\ 0\ \&\ n{=}0 \mid m{=}0\ \&\ n{=}\ Suc\ 0)$

⟨*proof*⟩

lemma *one-is-add*: $(Suc\ 0 = m + n) = (m = Suc\ 0\ \&\ n = 0\ |\ m = 0\ \&\ n = Suc\ 0)$
⟨*proof*⟩

lemma *add-gr-0* [*iff*]: $!!m::nat.\ (0 < m + n) = (0 < m\ |\ 0 < n)$
⟨*proof*⟩

lemma *add-eq-self-zero*: $!!m::nat.\ m + n = m ==> n = 0$
⟨*proof*⟩

lemma *inj-on-add-nat*[*simp*]: *inj-on* $(\%n::nat.\ n+k)\ N$
⟨*proof*⟩

16.10 Multiplication

right annihilation in product

lemma *mult-0-right* [*simp*]: $(m::nat) * 0 = 0$
⟨*proof*⟩

right successor law for multiplication

lemma *mult-Suc-right* [*simp*]: $m * Suc\ n = m + (m * n)$
⟨*proof*⟩

Commutative law for multiplication

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
⟨*proof*⟩

addition distributes over multiplication

lemma *add-mult-distrib*: $(m + n) * k = (m * k) + ((n * k)::nat)$
⟨*proof*⟩

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
⟨*proof*⟩

Associative law for multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
⟨*proof*⟩

The naturals form a *comm-semiring-1-cancel*

instance *nat* :: *comm-semiring-1-cancel*
⟨*proof*⟩

lemma *mult-is-0* [*simp*]: $((m::nat) * n = 0) = (m=0\ |\ n=0)$
⟨*proof*⟩

16.11 Monotonicity of Addition

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*: $m < n ==> (\exists\, k.\ n = Suc\ (m + k))$
⟨*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*⟩

16.12 Additional theorems about "less than"

A [clumsy] way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
assumes *lt-mono*: $!!i\ j{::}nat.\ i < j ==> f\,i < f\,j$
and *le*: $i \leq j$ **shows** $f\,i \leq ((f\,j){::}nat)$ ⟨*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*: $(i::nat) \le j$ ==> $i \le j + m$
⟨*proof*⟩

lemma *trans-le-add2*: $(i::nat) \le j$ ==> $i \le m + j$
⟨*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*]: ~ $(i + j < (i::nat))$
⟨*proof*⟩

lemma *not-add-less2* [*iff*]: ~ $(j + i < (i::nat))$
⟨*proof*⟩

lemma *add-leD1*: $m + k \le n$ ==> $m \le (n::nat)$
⟨*proof*⟩

lemma *add-leD2*: $m + k \le n$ ==> $k \le (n::nat)$
⟨*proof*⟩

lemma *add-leE*: $(m::nat) + k \le n$ ==> ($m \le n$ ==> $k \le n$ ==> R) ==> R
⟨*proof*⟩

needs !!*k* for *add-ac* to work

lemma *less-add-eq-less*: !!$k::nat.\ k < l$ ==> $m + l = k + n$ ==> $m < n$
⟨*proof*⟩

16.13 Difference

lemma *diff-self-eq-0* [*simp*]: $(m::nat) - m = 0$
⟨*proof*⟩

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*⟩

16.14 More results about difference

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 *less-imp-diff-less*: $(j::nat) < k \Longrightarrow j - n < k$
⟨*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-Suc-less* [*simp*]: $0<n \Longrightarrow n - Suc\ i < n$
⟨*proof*⟩

This and the next few suggested by Florian Kammueller

lemma *diff-commute*: $(i::nat) - j - k = i - k - j$
⟨*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 *diff-add-inverse*: $(n + m) - n = (m::nat)$
⟨*proof*⟩

lemma *diff-add-inverse2*: $(m + n) - n = (m::nat)$
⟨*proof*⟩

lemma *le-imp-diff-is-add*: $i \le (j::nat) ==> (j - i = k) = (j = k + i)$
⟨*proof*⟩

lemma *diff-is-0-eq* [*simp*]: $((m::nat) - n = 0) = (m \le n)$
⟨*proof*⟩

lemma *diff-is-0-eq'* [*simp*]: $m \le n ==> (m::nat) - n = 0$
⟨*proof*⟩

lemma *zero-less-diff* [*simp*]: $(0 < n - (m::nat)) = (m < n)$
⟨*proof*⟩

lemma *less-imp-add-positive*: $i < j \ ==> \exists k::nat.\ 0 < k\ \&\ i + k = j$
⟨*proof*⟩

lemma *zero-induct-lemma*: $P\ k ==> (!!n.\ P\ (Suc\ n) ==> P\ n) ==> P\ (k - i)$
⟨*proof*⟩

lemma *zero-induct*: $P\ k ==> (!!n.\ P\ (Suc\ n) ==> P\ n) ==> P\ 0$
⟨*proof*⟩

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*⟩

lemmas *nat-distrib* =
add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

16.15 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \le (j::nat) ==> i * k \le j * k$
⟨*proof*⟩

lemma *mult-le-mono2*: $i \le (j::nat) ==> k * i \le k * j$
⟨*proof*⟩

$\le$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq (j::nat) ==> k \leq l ==> i * k \leq j * l$
⟨*proof*⟩

lemma *mult-less-mono1*: $(i::nat) < j ==> 0 < k ==> i * k < j * k$
⟨*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*]: $(Suc\ 0 \leq m * n) = (1 \leq m \ \& \ 1 \leq n)$
⟨*proof*⟩

lemma *mult-eq-1-iff* [*simp*]: $(m * n = Suc\ 0) = (m = 1 \ \& \ n = 1)$
⟨*proof*⟩

lemma *one-eq-mult-iff* [*simp*]: $(Suc\ 0 = m * n) = (m = 1 \ \& \ n = 1)$
⟨*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*]: $(k * (m::nat) \leq k * n) = (0 < k --> m \leq n)$
⟨*proof*⟩

lemma *mult-le-cancel2* [*simp*]: $((m::nat) * k \leq n * k) = (0 < k --> m \leq n)$
⟨*proof*⟩

lemma *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
⟨*proof*⟩

lemma *mult-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
⟨*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 *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
⟨*proof*⟩

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n ==> n = 1 \mid m = 0$

⟨*proof*⟩

end

17 NatArith: Further Arithmetic Facts Concerning the Natural Numbers

theory *NatArith*
imports *Nat*
uses *arith-data.ML*
begin

⟨*ML*⟩

The following proofs may rely on the arithmetic proof procedures.

lemma *le-iff-add*: $(m::nat) \leq n = (\exists\, k.\ n = m + k)$
⟨*proof*⟩

lemma *pred-nat-trancl-eq-le*: $((m,\ n) : pred\text{-}nat\hat{\ }{*}) = (m \leq n)$
⟨*proof*⟩

lemma *nat-diff-split*:
$P(a - b::nat) = ((a{<}b \ {-}{-}{>}\ P\ 0)\ \&\ (ALL\ d.\ a = b + d\ {-}{-}{>}\ P\ d))$
— elimination of $-$ on *nat*
⟨*proof*⟩

lemma *nat-diff-split-asm*:
$P(a - b::nat) = ({\sim}\ (a < b\ \&\ {\sim}\ P\ 0\ |\ (EX\ d.\ a = b + d\ \&\ {\sim}\ P\ d)))$
— elimination of $-$ on *nat* in assumptions
⟨*proof*⟩

lemmas [*arith-split*] = *nat-diff-split split-min split-max*

lemma *le-square*: $m \leq m{*}(m::nat)$
⟨*proof*⟩

lemma *le-cube*: $(m::nat) \leq m{*}(m{*}m)$
⟨*proof*⟩

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: $[|\ a < (b::nat);\ c \leq a\ |] ==> a{-}c < b{-}c$
⟨*proof*⟩

lemma *less-diff-conv*: $(i < j{-}k) = (i{+}k < (j::nat))$

⟨*proof*⟩

lemma *le-diff-conv*: $(j-k \le (i::nat)) = (j \le i+k)$
⟨*proof*⟩

lemma *le-diff-conv2*: $k \le j ==> (i \le j-k) = (i+k \le (j::nat))$
⟨*proof*⟩

lemma *diff-diff-cancel* [*simp*]: $i \le (n::nat) ==> n - (n - i) = i$
⟨*proof*⟩

lemma *le-add-diff*: $k \le (n::nat) ==> m \le n + m - k$
⟨*proof*⟩

lemma *diff-less*[*simp*]: $!!m::nat.\ [|\ 0<n;\ 0<m\ |] ==> m - n < m$
⟨*proof*⟩

lemma *diff-diff-eq*: $[|\ k \le m;\ \ k \le (n::nat)\ |] ==> ((m-k) - (n-k)) = (m-n)$
⟨*proof*⟩

lemma *eq-diff-iff*: $[|\ k \le m;\ \ k \le (n::nat)\ |] ==> (m-k = n-k) = (m=n)$
⟨*proof*⟩

lemma *less-diff-iff*: $[|\ k \le m;\ \ k \le (n::nat)\ |] ==> (m-k < n-k) = (m<n)$
⟨*proof*⟩

lemma *le-diff-iff*: $[|\ k \le m;\ \ k \le (n::nat)\ |] ==> (m-k \le n-k) = (m \le n)$
⟨*proof*⟩

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \le (n::nat) ==> (m-l) \le (n-l)$
⟨*proof*⟩

lemma *diff-le-mono2*: $m \le (n::nat) ==> (l-n) \le (l-m)$
⟨*proof*⟩

lemma *diff-less-mono2*: $[|\ m < (n::nat);\ m<l\ |] ==> (l-n) < (l-m)$
⟨*proof*⟩

lemma *diffs0-imp-equal*: $!!m::nat.\ [|\ m-n = 0;\ n-m = 0\ |] ==>\ \ m=n$
⟨*proof*⟩

Lemmas for ex/Factorization

lemma *one-less-mult*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] ==> Suc\ 0 < m*n$
⟨*proof*⟩

lemma *n-less-m-mult-n*: $[\mid Suc\ 0 < n;\ Suc\ 0 < m \mid] ==> n{<}m{*}n$
⟨*proof*⟩

lemma *n-less-n-mult-m*: $[\mid Suc\ 0 < n;\ Suc\ 0 < m \mid] ==> n{<}n{*}m$
⟨*proof*⟩

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k{\le}j\ {-}{-}{>}\ i - (j - k) = i + (k{::}nat) - j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq1* [*simp*]: $k \le j ==> m - Suc\ (j - k) = m + k - Suc\ j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq2* [*simp*]: $k \le j ==> Suc\ (j - k) - m = Suc\ j - (k + m)$
⟨*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.

⟨*ML*⟩

17.1 Embedding of the Naturals into any *comm-semiring-1-cancel*: *of-nat*

consts *of-nat* :: *nat* => *'a*::*comm-semiring-1-cancel*

primrec
of-nat-0: *of-nat* $0 = 0$
of-nat-Suc: *of-nat* $(Suc\ m) =$ *of-nat* $m + 1$

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* [*simp*]: *of-nat* $(m{*}n) =$ *of-nat* $m *$ *of-nat* n
⟨*proof*⟩

lemma *zero-le-imp-of-nat*: $0 \le$ (*of-nat* *m*::*'a*::*ordered-semidom*)
⟨*proof*⟩

lemma *less-imp-of-nat-less*:
$m < n ==>$ *of-nat* $m <$ (*of-nat* *n*::*'a*::*ordered-semidom*)

⟨proof⟩

lemma *of-nat-less-imp-less*:
$of\text{-}nat\ m < (of\text{-}nat\ n::'a::ordered\text{-}semidom) ==> m < n$
⟨proof⟩

lemma *of-nat-less-iff* [*simp*]:
$(of\text{-}nat\ m < (of\text{-}nat\ n::'a::ordered\text{-}semidom)) = (m<n)$
⟨proof⟩

Special cases where either operand is zero

lemmas *of-nat-0-less-iff* = *of-nat-less-iff* [*of 0*, *simplified*]
lemmas *of-nat-less-0-iff* = *of-nat-less-iff* [*of - 0*, *simplified*]
declare *of-nat-0-less-iff* [*simp*]
declare *of-nat-less-0-iff* [*simp*]

lemma *of-nat-le-iff* [*simp*]:
$(of\text{-}nat\ m \leq (of\text{-}nat\ n::'a::ordered\text{-}semidom)) = (m \leq n)$
⟨proof⟩

Special cases where either operand is zero

lemmas *of-nat-0-le-iff* = *of-nat-le-iff* [*of 0*, *simplified*]
lemmas *of-nat-le-0-iff* = *of-nat-le-iff* [*of - 0*, *simplified*]
declare *of-nat-0-le-iff* [*simp*]
declare *of-nat-le-0-iff* [*simp*]

The ordering on the *comm-semiring-1-cancel* is necessary to exclude the possibility of a finite field, which indeed wraps back to zero.

lemma *of-nat-eq-iff* [*simp*]:
$(of\text{-}nat\ m = (of\text{-}nat\ n::'a::ordered\text{-}semidom)) = (m = n)$
⟨proof⟩

Special cases where either operand is zero

lemmas *of-nat-0-eq-iff* = *of-nat-eq-iff* [*of 0*, *simplified*]
lemmas *of-nat-eq-0-iff* = *of-nat-eq-iff* [*of - 0*, *simplified*]
declare *of-nat-0-eq-iff* [*simp*]
declare *of-nat-eq-0-iff* [*simp*]

lemma *of-nat-diff* [*simp*]:
$n \leq m ==> of\text{-}nat\ (m - n) = of\text{-}nat\ m - (of\text{-}nat\ n :: 'a::comm\text{-}ring\text{-}1)$
⟨proof⟩

end

18 Datatype-Universe: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

theory *Datatype-Universe*
imports *NatArith Sum-Type*
begin

typedef (*Node*)
$('a,'b)\ node = \{p.\ EX\ f\ x\ k.\ p = (f{::}nat{=}{>}'b{+}nat,\ x{::}'a{+}nat)\ \&\ f\ k = Inr\ 0\}$
— it is a subtype of $(nat{=}{>}'b{+}nat) * ('a{+}nat)$
$\langle proof \rangle$

Datatypes will be represented by sets of type *node*

types $'a\ item = ('a,\ unit)\ node\ set$
$('a,\ 'b)\ dtree = ('a,\ 'b)\ node\ set$

consts
$apfst :: ['a{=}{>}'c,\ 'a{*}'b] => 'c{*}'b$
$Push :: [('b + nat),\ nat => ('b + nat)] => (nat => ('b + nat))$

$Push\text{-}Node :: [('b + nat),\ ('a,\ 'b)\ node] => ('a,\ 'b)\ node$
$ndepth :: ('a,\ 'b)\ node => nat$

$Atom :: ('a + nat) => ('a,\ 'b)\ dtree$
$Leaf :: 'a => ('a,\ 'b)\ dtree$
$Numb :: nat => ('a,\ 'b)\ dtree$
$Scons :: [('a,\ 'b)\ dtree,\ ('a,\ 'b)\ dtree] => ('a,\ 'b)\ dtree$
$In0 :: ('a,\ 'b)\ dtree => ('a,\ 'b)\ dtree$
$In1 :: ('a,\ 'b)\ dtree => ('a,\ 'b)\ dtree$
$Lim :: ('b => ('a,\ 'b)\ dtree) => ('a,\ 'b)\ dtree$

$ntrunc :: [nat,\ ('a,\ 'b)\ dtree] => ('a,\ 'b)\ dtree$

$uprod :: [('a,\ 'b)\ dtree\ set,\ ('a,\ 'b)\ dtree\ set]{=}{>}\ ('a,\ 'b)\ dtree\ set$
$usum :: [('a,\ 'b)\ dtree\ set,\ ('a,\ 'b)\ dtree\ set]{=}{>}\ ('a,\ 'b)\ dtree\ set$

$Split :: [[('a,\ 'b)\ dtree,\ ('a,\ 'b)\ dtree]{=}{>}'c,\ ('a,\ 'b)\ dtree] => 'c$
$Case :: [[('a,\ 'b)\ dtree]{=}{>}'c,\ [('a,\ 'b)\ dtree]{=}{>}'c,\ ('a,\ 'b)\ dtree] => 'c$

$dprod :: [(('a,\ 'b)\ dtree * ('a,\ 'b)\ dtree)set,\ (('a,\ 'b)\ dtree * ('a,\ 'b)\ dtree)set]$
$=> (('a,\ 'b)\ dtree * ('a,\ 'b)\ dtree)set$
$dsum :: [(('a,\ 'b)\ dtree * ('a,\ 'b)\ dtree)set,\ (('a,\ 'b)\ dtree * ('a,\ 'b)\ dtree)set]$
$=> (('a,\ 'b)\ dtree * ('a,\ 'b)\ dtree)set$

defs

Push-Node-def: $Push\text{-}Node == (\%n\ x.\ Abs\text{-}Node\ (apfst\ (Push\ n)\ (Rep\text{-}Node$

x)))

apfst-def: *apfst* == (%*f* (*x*,*y*). (*f*(*x*),*y*))
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-conv* [*simp*]: *apfst f* $(a,b) = (f(a),b)$
⟨*proof*⟩

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*⟩

18.1 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(a)
⟨*proof*⟩

lemmas *Atom-not-Scons* = *Scons-not-Atom* [*THEN not-sym*, *standard*]
declare *Atom-not-Scons* [*iff*]

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 = inj-Leaf [THEN injD, standard]*
declare *Leaf-inject [dest!]*

lemma *inj-Numb: inj(Numb)*
⟨*proof*⟩

lemmas *Numb-inject = inj-Numb [THEN injD, standard]*
declare *Numb-inject [dest!]*

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* $\neq$ *Leaf*(*a*)
⟨*proof*⟩

lemmas *Leaf-not-Scons* = *Scons-not-Leaf* [*THEN not-sym*, *standard*]
declare *Leaf-not-Scons* [*iff*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* $\neq$ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Scons* = *Scons-not-Numb* [*THEN not-sym*, *standard*]
declare *Numb-not-Scons* [*iff*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) $\neq$ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Leaf* = *Leaf-not-Numb* [*THEN not-sym*, *standard*]
declare *Numb-not-Leaf* [*iff*]

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*⟩

18.2 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* = *In0-not-In1* [*THEN not-sym*, *standard*]
declare *In1-not-In0* [*iff*]

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*: $Scons\ (UN\ x.\ f\ x)\ M = (UN\ x.\ Scons\ (f\ x)\ M)$
$\langle proof \rangle$

lemma *Scons-UN1-y*: $Scons\ M\ (UN\ x.\ f\ x) = (UN\ x.\ Scons\ M\ (f\ x))$
$\langle proof \rangle$

lemma *In0-UN1*: $In0(UN\ x.\ f(x)) = (UN\ x.\ In0(f(x)))$
$\langle proof \rangle$

lemma *In1-UN1*: $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
$\langle proof \rangle$

lemma *dprodI* [*intro*!]:
  $[|\ (M,M'):r;\ \ (N,N'):s\ |] ==> (Scons\ M\ N,\ Scons\ M'\ N') : dprod\ r\ s$
$\langle proof \rangle$

lemma *dprodE* [*elim*!]:
  $[|\ c : dprod\ r\ s;$
    $!!x\ y\ x'\ y'.\ [|\ (x,x') : r;\ \ (y,y') : s;$
        $c = (Scons\ x\ y,\ Scons\ x'\ y')\ |] ==> P$
  $|] ==> P$
$\langle proof \rangle$

lemma *dsum-In0I* [*intro*]: $(M,M'):r ==> (In0(M),\ In0(M')) : dsum\ r\ s$
$\langle proof \rangle$

lemma *dsum-In1I* [*intro*]: $(N,N'):s ==> (In1(N),\ In1(N')) : dsum\ r\ s$
$\langle proof \rangle$

lemma *dsumE* [*elim*!]:
  $[|\ w : dsum\ r\ s;$
    $!!x\ x'.\ [|\ (x,x') : r;\ \ w = (In0(x),\ In0(x'))\ |] ==> P;$
    $!!y\ y'.\ [|\ (y,y') : s;\ \ w = (In1(y),\ In1(y'))\ |] ==> P$
  $|] ==> P$
$\langle proof \rangle$

lemma *dprod-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dprod\ r\ s <= dprod\ r'\ s'$
$\langle proof \rangle$

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*⟩

⟨*ML*⟩

end

19 Datatype: Datatypes

theory *Datatype*
imports *Datatype-Universe*
begin

19.1 Representing primitive types

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

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

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

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

rep-datatype *sum*
  distinct *Inl-not-Inr Inr-not-Inl*
  inject *Inl-eq Inr-eq*
  induction *sum-induct*

⟨*ML*⟩

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*⟩

19.2 Finishing the datatype package setup

Belongs to theory *Datatype-Universe*; hides popular names.

hide *const Push Node Atom Leaf Numb Lim Split Case Suml Sumr*
hide *type node item*

19.3 Further cases/induct rules for tuples

lemma *prod-cases3* [*case-names fields*, *cases type*]:
  $(!!a\ b\ c.\ y = (a,\ b,\ c) ==> P) ==> P$
  ⟨*proof*⟩

lemma *prod-induct3* [*case-names fields*, *induct type*]:
  $(!!a\ b\ c.\ P\ (a,\ b,\ c)) ==> P\ x$
  ⟨*proof*⟩

lemma *prod-cases4* [*case-names fields*, *cases type*]:
  $(!!a\ b\ c\ d.\ y = (a,\ b,\ c,\ d) ==> P) ==> P$
  ⟨*proof*⟩

lemma *prod-induct4* [*case-names fields*, *induct type*]:
  $(!!a\ b\ c\ d.\ P\ (a,\ b,\ c,\ d)) ==> P\ x$
  ⟨*proof*⟩

lemma *prod-cases5* [*case-names fields*, *cases type*]:
  $(!!a\ b\ c\ d\ e.\ y = (a,\ b,\ c,\ d,\ e) ==> P) ==> P$
  ⟨*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* [*case-names fields*, *cases type*]:
  $(!!a\ b\ c\ d\ e\ f.\ y = (a,\ b,\ c,\ d,\ e,\ f) ==> P) ==> P$
  ⟨*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* [*case-names fields*, *cases type*]:
  $(!!a\ b\ c\ d\ e\ f\ g.\ y = (a,\ b,\ c,\ d,\ e,\ f,\ g) ==> P) ==> P$
  ⟨*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*⟩

19.4 The option type

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*⟩

lemma *option-caseE*:
(*case x of None* => *P* | *Some y* => *Q y*) ==>
(*x* = *None* ==> *P* ==> *R*) ==>
(!!*y. x* = *Some y* ==> *Q y* ==> *R*) ==> *R*
⟨*proof*⟩

19.4.1 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*⟩

constdefs
option-map :: (*'a* => *'b*) => (*'a option* => *'b option*)
option-map == %*f y. case y of None* => *None* | *Some x* => *Some* (*f x*)

lemma *option-map-None* [*simp*]: *option-map f None* = *None*
⟨*proof*⟩

lemma *option-map-Some* [*simp*]: *option-map f* (*Some x*) = *Some* (*f x*)
⟨*proof*⟩

lemma *option-map-is-None*[*iff*]:
$(option\text{-}map\ f\ opt = None) = (opt = None)$
⟨*proof*⟩

lemma *option-map-eq-Some* [*iff*]:
$(option\text{-}map\ f\ xo = Some\ y) = (EX\ z.\ xo = Some\ z\ \&\ f\ z = y)$
⟨*proof*⟩

lemma *option-map-comp*:
$option\text{-}map\ f\ (option\text{-}map\ g\ opt) = option\text{-}map\ (f\ o\ g)\ opt$
⟨*proof*⟩

lemma *option-map-o-sum-case* [*simp*]:
$option\text{-}map\ f\ o\ sum\text{-}case\ g\ h = sum\text{-}case\ (option\text{-}map\ f\ o\ g)\ (option\text{-}map\ f\ o\ h)$
⟨*proof*⟩

lemmas [*code*] = *imp-conv-disj*

end

theory *Divides*
imports *Datatype*
begin

axclass
div < *type*

instance *nat* :: *div* ⟨*proof*⟩

consts
div :: $'a{::}div \Rightarrow 'a \Rightarrow 'a$ (**infixl** *70*)
mod :: $'a{::}div \Rightarrow 'a \Rightarrow 'a$ (**infixl** *70*)
dvd :: $'a{::}times \Rightarrow 'a \Rightarrow bool$ (**infixl** *50*)

defs

mod-def: $m\ mod\ n == wfrec\ (trancl\ pred\text{-}nat)$
$(\%f\ j.\ if\ j{<}n\ |\ n{=}0\ then\ j\ else\ f\ (j{-}n))\ m$

div-def: $m\ div\ n == wfrec\ (trancl\ pred\text{-}nat)$
$(\%f\ j.\ if\ j{<}n\ |\ n{=}0\ then\ 0\ else\ Suc\ (f\ (j{-}n)))\ m$

dvd-def: $m\ dvd\ n == \exists\, k.\ n = m{*}k$

constdefs
quorem :: (*nat*∗*nat*) ∗ (*nat*∗*nat*) => *bool*
quorem == %((*a*,*b*), (*q*,*r*)).
a = *b*∗*q* + *r* &
(*if 0*<*b then 0*≤*r* & *r*<*b else b*<*r* & *r* ≤*0*)

19.5 Initial Lemmas

lemmas *wf-less-trans* =
def-wfrec [*THEN trans*, *OF eq-reflection wf-pred-nat* [*THEN wf-trancl*],
standard]

lemma *mod-eq*: (%*m*. *m mod n*) =
wfrec (*trancl pred-nat*) (%*f j*. *if j*<*n* | *n*=*0 then j else f* (*j*−*n*))
⟨*proof*⟩

lemma *div-eq*: (%*m*. *m div n*) = *wfrec* (*trancl pred-nat*)
(%*f j*. *if j*<*n* | *n*=*0 then 0 else Suc* (*f* (*j*−*n*)))
⟨*proof*⟩

lemma *DIVISION-BY-ZERO-DIV* [*simp*]: *a div 0* = (*0*::*nat*)
⟨*proof*⟩

lemma *DIVISION-BY-ZERO-MOD* [*simp*]: *a mod 0* = (*a*::*nat*)
⟨*proof*⟩

19.6 Remainder

lemma *mod-less* [*simp*]: *m*<*n* ==> *m mod n* = (*m*::*nat*)
⟨*proof*⟩

lemma *mod-geq*: ~ *m* < (*n*::*nat*) ==> *m mod n* = (*m*−*n*) *mod n*
⟨*proof*⟩

lemma *le-mod-geq*: (*n*::*nat*) ≤ *m* ==> *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*⟩

lemma *mod-self* [*simp*]: *n mod n* = (*0*::*nat*)

⟨*proof*⟩

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*⟩

19.7 Quotient

lemma *div-less* [*simp*]: $m<n$ ==> m *div* $n = (0::nat)$
⟨*proof*⟩

lemma *div-geq*: [| $0<n$; $\sim m<n$ |] ==> m *div* $n = Suc((m-n)$ *div* $n)$
⟨*proof*⟩

lemma *le-div-geq*: [| $0<n$; $n \leq m$ |] ==> 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 *mod-div-equality*: $(m$ *div* $n)*n + m$ *mod* $n = (m::nat)$
⟨*proof*⟩

lemma *mod-div-equality2*: $n * (m$ *div* $n) + m$ *mod* $n = (m::nat)$
⟨*proof*⟩

19.8 Simproc for Cancelling Div and Mod

lemma *div-mod-equality*: $((m\ div\ n)*n + m\ mod\ n) + k = (m::nat) + k$
⟨*proof*⟩

lemma *div-mod-equality2*: $(n*(m\ div\ n) + m\ mod\ n) + k = (m::nat) + k$
⟨*proof*⟩

⟨*ML*⟩

lemma *mult-div-cancel*: $(n::nat) * (m\ div\ n) = m - (m\ mod\ n)$
⟨*proof*⟩

lemma *mod-less-divisor* [*simp*]: $0{<}n ==> m\ mod\ n < (n::nat)$
⟨*proof*⟩

lemma *mod-le-divisor*[*simp*]: $0 < n \Longrightarrow m\ mod\ n \leq (n::nat)$
⟨*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*⟩

19.9 Proving facts about Quotient and Remainder

lemma *unique-quotient-lemma*:
$[|\ b*q' + r' \ \leq b*q + r;\ \ x < b;\ \ r < b\ |]$
$==> q' \leq (q::nat)$
⟨*proof*⟩

lemma *unique-quotient*:
$[|\ quorem\ ((a,b),\ (q,r));\ \ quorem\ ((a,b),\ (q',r'));\ \ 0 < b\ |]$
$==> q = q'$
⟨*proof*⟩

lemma *unique-remainder*:
$[|\ quorem\ ((a,b),\ (q,r));\ \ quorem\ ((a,b),\ (q',r'));\ \ 0 < b\ |]$
$==> r = r'$
⟨*proof*⟩

lemma *quorem-div-mod*: $0 < b ==> quorem\ ((a,\ b),\ (a\ div\ b,\ a\ mod\ b))$
⟨*proof*⟩

lemma *quorem-div*: $[|\ quorem((a,b),(q,r));\ \ 0 < b\ |] ==> a\ div\ b = q$
⟨*proof*⟩

lemma *quorem-mod*: $[| \ quorem((a,b),(q,r)); \ \ 0 < b \ |] ==> a \ mod \ b = r$
⟨*proof*⟩

lemma *div-0* [*simp*]: $0 \ div \ m = (0::nat)$
⟨*proof*⟩

lemma *mod-0* [*simp*]: $0 \ mod \ m = (0::nat)$
⟨*proof*⟩

lemma *quorem-mult1-eq*:
$[| \ quorem((b,c),(q,r)); \ \ 0 < c \ |]$
$==> quorem \ ((a*b, \ c), \ (a*q + a*r \ div \ c, \ a*r \ mod \ c))$
⟨*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 *quorem-add1-eq*:
$[| \ quorem((a,c),(aq,ar)); \ \ quorem((b,c),(bq,br)); \ \ 0 < c \ |]$
$==> quorem \ ((a+b, \ c), \ (aq + bq + (ar+br) \ div \ c, \ (ar+br) \ mod \ c))$
⟨*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*⟩

19.10 Proving $a \ div \ (b * c) = a \ div \ b \ div \ c$

lemma *mod-lemma*: $[| \ (0::nat) < c; \ r < b \ |] ==> b * (q \ mod \ c) + r < b * c$

⟨*proof*⟩

lemma *quorem-mult2-eq*: [| *quorem* ((*a*,*b*), (*q*,*r*)); *0* < *b*; *0* < *c* |]
==> *quorem* ((*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*⟩

19.11 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*⟩

19.12 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m div Suc 0* = *m*
⟨*proof*⟩

lemma *div-self* [*simp*]: *0*<*n* ==> *n div n* = (*1*::*nat*)
⟨*proof*⟩

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*: $!!m::nat.\ [|\ 0<m;\ m\leq n\ |] ==> (k\ div\ n) \leq (k\ div\ m)$
⟨*proof*⟩

lemma *div-le-dividend* [*simp*]: $m\ div\ n \leq (m::nat)$
⟨*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*⟩

19.13 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*⟩

lemma *dvd-1-left* [*iff*]: $Suc\ 0\ dvd\ k$
⟨*proof*⟩

lemma *dvd-1-iff-1* [*simp*]: $(m\ dvd\ Suc\ 0) = (m = Suc\ 0)$
⟨*proof*⟩

lemma *dvd-refl* [*simp*]: *m dvd* (*m*::*nat*)
⟨*proof*⟩

lemma *dvd-trans* [*trans*]: [| *m dvd n*; *n dvd p* |] ==> *m dvd* (*p*::*nat*)
⟨*proof*⟩

lemma *dvd-anti-sym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
⟨*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*⟩

lemma *dvd-eq-mod-eq-0*: $!!k::nat.\ (k\ dvd\ n) = (n\ mod\ k = 0)$
⟨*proof*⟩

lemma *dvd-mult-div-cancel*: $n\ dvd\ m ==> n * (m\ div\ n) = (m::nat)$
⟨*proof*⟩

lemma *mod-eq-0-iff*: $(m\ mod\ d = 0) = (\exists\, q::nat.\ m = d*q)$
⟨*proof*⟩

lemmas *mod-eq-0D* = *mod-eq-0-iff* [*THEN iffD1*]
declare *mod-eq-0D* [*dest*!]

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** - = (- $\wedge$ (- $\longrightarrow$ *?R*)))
⟨*proof*⟩

lemma *split-div-lemma*:
$0 < n \Longrightarrow (n * q \leq m \wedge m < n * (Suc\ q)) = (q = ((m::nat)\ div\ n))$
⟨*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** - = (- $\wedge$ (- $\longrightarrow$ $?R$)))
⟨*proof*⟩

theorem *mod-div-equality'*: $(m::nat)\ mod\ n = m - (m\ div\ n) * n$
⟨*proof*⟩

19.14 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*⟩

⟨*ML*⟩

end

20 Power: Exponentiation

theory *Power*
imports *Divides*
begin

20.1 Powers for Arbitrary Semirings

axclass *recpower* $\subseteq$ *comm-semiring-1-cancel*, *power*
power-0 [*simp*]: $a\ \hat{}\ 0 = 1$
power-Suc: $a\ \hat{}\ (Suc\ n) = a * (a\ \hat{}\ n)$

lemma *power-0-Suc* [*simp*]: $(0::'a::recpower)\ \hat{}\ (Suc\ n) = 0$
⟨*proof*⟩

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: $0\hat{}n = (if\ n=0\ then\ 1\ else\ (0::'a::recpower))$
⟨*proof*⟩

lemma *power-one* [*simp*]: $1\hat{}n = (1::'a::recpower)$
⟨*proof*⟩

lemma *power-one-right* [*simp*]: $(a::'a::recpower)\ \hat{}\ 1 = a$
⟨*proof*⟩

lemma *power-add*: $(a::'a::recpower)\ \hat{}\ (m+n) = (a\hat{}m) * (a\hat{}n)$
⟨*proof*⟩

lemma *power-mult*: $(a::'a::recpower)\ \hat{}\ (m*n) = (a\hat{}m)\ \hat{}\ n$
⟨*proof*⟩

lemma *power-mult-distrib*: $((a::'a::recpower) * b)\ \hat{}\ n = (a\hat{}n) * (b\hat{}n)$
⟨*proof*⟩

lemma *zero-less-power*:
$0 < (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 0 < a\hat{}n$
⟨*proof*⟩

lemma *zero-le-power*:
$0 \le (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 0 \le a\hat{}n$
⟨*proof*⟩

lemma *one-le-power*:
$1 \le (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 1 \le a\hat{}n$
⟨*proof*⟩

lemma *gt1-imp-ge0*: $1 < a ==> 0 \le (a::'a::ordered\text{-}semidom)$
⟨*proof*⟩

lemma *power-gt1-lemma*:
assumes *gt1*: $1 < (a::'a::\{ordered\text{-}semidom,recpower\})$
shows $1 < a * a\hat{}n$
⟨*proof*⟩

lemma *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 $!!n.\ a\hat{}m \le a\hat{}n ==> m \le n$
⟨*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*:
$[|\ (1::'a::\{recpower,ordered\text{-}semidom\}) < a;\ a\hat{}m < a\hat{}n\ |] ==> m < n$
$\langle proof \rangle$

lemma *power-mono*:
$[|a \le b;\ (0::'a::\{recpower,ordered\text{-}semidom\}) \le a|] ==> a\hat{}n \le b\hat{}n$
$\langle proof \rangle$

lemma *power-strict-mono* [*rule-format*]:
$[|a < b;\ (0::'a::\{recpower,ordered\text{-}semidom\}) \le a|]$
$==> 0 < n --> a\hat{}n < b\hat{}n$
$\langle proof \rangle$

lemma *power-eq-0-iff* [*simp*]:
$(a\hat{}n = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\})\ \&\ 0<n)$
$\langle proof \rangle$

lemma *field-power-eq-0-iff* [*simp*]:
$(a\hat{}n = 0) = (a = (0::'a::\{field,recpower\})\ \&\ 0<n)$
$\langle proof \rangle$

lemma *field-power-not-zero*: $a \neq (0::'a::\{field,recpower\}) ==> a\hat{}n \neq 0$
$\langle proof \rangle$

lemma *nonzero-power-inverse*:
$a \neq 0 ==> inverse\ ((a::'a::\{field,recpower\})\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
$\langle proof \rangle$

Perhaps these should be simprules.

lemma *power-inverse*:
$inverse\ ((a::'a::\{field,division\text{-}by\text{-}zero,recpower\})\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
$\langle proof \rangle$

lemma *power-one-over*: $1\ /\ (a::'a::\{field,division\text{-}by\text{-}zero,recpower\})\hat{}n =$
$(1\ /\ a)\hat{}n$
$\langle proof \rangle$

lemma *nonzero-power-divide*:
$b \neq 0 ==> (a/b)\ \hat{}\ n = ((a::'a::\{field,recpower\})\ \hat{}\ n)\ /\ (b\ \hat{}\ n)$
$\langle proof \rangle$

lemma *power-divide*:
$(a/b)\ \hat{}\ n = ((a::'a::\{field,division\text{-}by\text{-}zero,recpower\})\ \hat{}\ n\ /\ b\ \hat{}\ n)$
$\langle proof \rangle$

lemma *power-abs*: $abs(a\ \hat{}\ n) = abs(a::'a::\{ordered\text{-}idom,recpower\})\ \hat{}\ n$
$\langle proof \rangle$

lemma *zero-less-power-abs-iff* [*simp*]:
$(0 < (abs\ a)\hat{}n) = (a \neq (0::'a::\{ordered\text{-}idom,recpower\}) \mid n=0)$
⟨*proof*⟩

lemma *zero-le-power-abs* [*simp*]:
$(0::'a::\{ordered\text{-}idom,recpower\}) \leq (abs\ a)\hat{}n$
⟨*proof*⟩

lemma *power-minus*: $(-a)\ \hat{}\ n = (-\ 1)\hat{}n * (a::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ n$
⟨*proof*⟩

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:
$[|(0::'a::\{ordered\text{-}semidom,recpower\}) < a;\ a < 1|]$
$==> a * a\hat{}n < a\hat{}n$
⟨*proof*⟩

lemma *power-strict-decreasing*:
$[|n < N;\ 0 < a;\ a < (1::'a::\{ordered\text{-}semidom,recpower\})|]$
$==> a\hat{}N < a\hat{}n$
⟨*proof*⟩

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing*:
$[|n \leq N;\ 0 \leq a;\ a \leq (1::'a::\{ordered\text{-}semidom,recpower\})|]$
$==> a\hat{}N \leq a\hat{}n$
⟨*proof*⟩

lemma *power-Suc-less-one*:
$[|\ 0 < a;\ a < (1::'a::\{ordered\text{-}semidom,recpower\})\ |] ==> a\ \hat{}\ Suc\ n < 1$
⟨*proof*⟩

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing*:
$[|n \leq N;\ (1::'a::\{ordered\text{-}semidom,recpower\}) \leq a|] ==> a\hat{}n \leq a\hat{}N$
⟨*proof*⟩

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
$(1::'a::\{ordered\text{-}semidom,recpower\}) < a ==> a\hat{}n < a * a\hat{}n$
⟨*proof*⟩

lemma *power-strict-increasing*:
$[|n < N;\ (1::'a::\{ordered\text{-}semidom,recpower\}) < a|] ==> a\hat{}n < a\hat{}N$
⟨*proof*⟩

lemma *power-increasing-iff* [*simp*]:

$1 < (b::'a::\{ordered\text{-}semidom,recpower\})$ ==> $(b \ \hat{} \ x \leq b \ \hat{} \ y) = (x \leq y)$
⟨*proof*⟩

lemma *power-strict-increasing-iff* [*simp*]:
$1 < (b::'a::\{ordered\text{-}semidom,recpower\})$ ==> $(b \ \hat{} \ x < b \ \hat{} \ y) = (x < y)$
⟨*proof*⟩

lemma *power-le-imp-le-base*:
assumes *le*: $a \ \hat{} \ Suc\ n \leq b \ \hat{} \ Suc\ n$
and *xnonneg*: $(0::'a::\{ordered\text{-}semidom,recpower\}) \leq a$
and *ynonneg*: $0 \leq b$
shows $a \leq b$
⟨*proof*⟩

lemma *power-inject-base*:
[| $a \ \hat{} \ Suc\ n = b \ \hat{} \ Suc\ n$; $0 \leq a$; $0 \leq b$ |]
==> $a = (b::'a::\{ordered\text{-}semidom,recpower\})$
⟨*proof*⟩

20.2 Exponentiation for the Natural Numbers

primrec (*power*)
$p \ \hat{} \ 0 = 1$
$p \ \hat{} \ (Suc\ n) = (p::nat) * (p \ \hat{} \ n)$

instance *nat* :: *recpower*
⟨*proof*⟩

lemma *nat-one-le-power* [*simp*]: $1 \leq i$ ==> $Suc\ 0 \leq i\hat{}n$
⟨*proof*⟩

lemma *le-imp-power-dvd*: !!$i::nat$. $m \leq n$ ==> $i\hat{}m\ dvd\ i\hat{}n$
⟨*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*: !!$i::nat$. [| $0 < i$; $i\hat{}m < i\hat{}n$ |] ==> $m < n$
⟨*proof*⟩

lemma *nat-zero-less-power-iff* [*simp*]: $(0 < x\hat{}n) = (x \neq (0::nat) \mid n=0)$
⟨*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 *power-diff*:

assumes *nz*: *a* ~= *0*
shows *n* <= *m* ==> (*a*::'*a*::{*recpower*, *field*}) ^ (*m*−*n*) = (*a*^*m*) / (*a*^*n*)
⟨*proof*⟩

ML bindings for the general exponentiation theorems

⟨*ML*⟩

ML bindings for the remaining theorems

⟨*ML*⟩

end

21 Finite-Set: Finite sets

theory *Finite-Set*
imports *Power Inductive Lattice-Locales*
begin

21.1 Definition and basic properties

consts *Finites* :: '*a set set*
syntax
finite :: '*a set* => *bool*
translations
finite A == *A* : *Finites*

inductive *Finites*
intros
emptyI [*simp*, *intro*!]: {} : *Finites*
insertI [*simp*, *intro*!]: *A* : *Finites* ==> *insert a A* : *Finites*

axclass *finite* ⊆ *type*
finite: *finite UNIV*

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*: *Finites*]:
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** $F \neq \{\} \Longrightarrow$
$[\![\bigwedge x.\ P\{x\};$
$\bigwedge x\ F.\ [\![$ *finite* $F;\ F \neq \{\};\ x \notin F;\ P\ F\ ]\!] \Longrightarrow P\ (insert\ x\ F)\ ]\!]$
$\Longrightarrow P\ F$
⟨*proof*⟩

lemma *finite-subset-induct* [*consumes 2*, *case-names empty insert*]:
finite F ==> $F \subseteq A$ ==>
P {} ==> (!!a F. *finite* F ==> $a \in A$ ==> $a \notin F$ ==> P F ==> P (*insert* a F)) ==>
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 $\exists$ (n::*nat*) f. $A = f$ ‘ {i. i<n} & *inj-on* f {i. i<n}
⟨*proof*⟩

lemma *nat-seg-image-imp-finite*:
!!f A. $A = f$ ‘ {i::*nat*. i<n} $\Longrightarrow$ *finite* A
⟨*proof*⟩

lemma *finite-conv-nat-seg-image*:
finite A = ($\exists$ (n::*nat*) f. $A = f$ ‘ {i::*nat*. i<n})
⟨*proof*⟩

21.1.1 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*: $A \subseteq B$ ==> *finite* B ==> *finite* A
— Every subset of a finite set is finite.
⟨*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. $M \in A \Longrightarrow$ *finite* M $]\!] \Longrightarrow$ *finite*($\bigcup A$)

⟨*proof*⟩

lemma *finite-empty-induct*:
 $finite\ A ==>$
 $P\ A ==> (!!a\ A.\ finite\ A ==> a{:}A ==> P\ A ==> P\ (A - \{a\})) ==> P\ \{\}$
⟨*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*⟩

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*: $finite\ (f`A) ==> inj\text{-}on\ f\ A ==> finite\ A$
⟨*proof*⟩

lemma *inj-vimage-singleton*: $inj\ f ==> f-`\{a\} \subseteq \{THE\ x.\ f\ x = a\}$
 — 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 $(\forall x{\in}A.\ finite\ (B\ x)) \wedge finite\ \{x \in A.\ B\ x \neq \{\}\}$?
We'd need to prove $finite\ C \Longrightarrow \forall A\ B.\ UNION\ A\ B \subseteq C \longrightarrow finite\ \{x \in A.\ B\ x \neq \{\}\}$ by induction.

lemma *finite-UN* [*simp*]: $finite\ A ==> finite\ (UNION\ A\ B) = (ALL\ x{:}A.\ finite\ (B\ x))$

⟨*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*($\bigcup A$) $\Longrightarrow$ *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* $(r\hat{\ }+)$ = *finite* r
⟨*proof*⟩

21.2 A fold functional for finite sets

The intended behaviour is *fold* f g z $\{x_1, ..., x_n\}$ = f $(g\ x_1)$ $(\ldots\ (f\ (g\ x_n)\ z)\ldots)$ if f is associative-commutative. For an application of *fold* se the definitions of sums and products over finite sets.

consts
foldSet :: $('a => 'a => 'a) => ('b => 'a) => 'a => ('b\ set \times 'a)\ set$

inductive *foldSet f g z*
intros
emptyI [*intro*]: $(\{\}, z)$: *foldSet f g z*
insertI [*intro*]:
$[\![\ x \notin A;\ (A, y)$: *foldSet f g z* $]\!]$
$\Longrightarrow$ (*insert* x A, f $(g\ x)$ y) : *foldSet f g z*

inductive-cases *empty-foldSetE* [*elim*!]: $(\{\}, x)$: *foldSet f g z*

constdefs
fold :: $('a => 'a => 'a) => ('b => 'a) => 'a => 'b\ set => 'a$
fold f g z A == *THE* x. (A, x) : *foldSet f g z*

A tempting alternative for the definiens is *if finite A then THE x*. $(A, x) \in$ *foldSet f g e 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*:
$(A - \{x\}, y)$: *foldSet f g z* ==> x: A ==> $(A, f\ (g\ x)\ y)$: *foldSet f g z*
⟨*proof*⟩

lemma *foldSet-imp-finite*: (A, x) : *foldSet f g z* ==> *finite* A
⟨*proof*⟩

lemma *finite-imp-foldSet*: *finite* A ==> *EX* x. (A, x) : *foldSet f g z*
⟨*proof*⟩

21.2.1 Commutative monoids

locale *ACf* =
fixes f :: $'a => 'a => 'a$ (**infixl** $\cdot$ *70*)
assumes *commute*: $x \cdot y = y \cdot x$
and *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

locale *ACe* = *ACf* +
fixes e :: $'a$

assumes *ident* [*simp*]: $x \cdot e = x$

locale *ACIf* = *ACf* +
assumes *idem*: $x \cdot x = x$

lemma (**in** *ACf*) *left-commute*: $x \cdot (y \cdot z) = y \cdot (x \cdot z)$
⟨*proof*⟩

lemmas (**in** *ACf*) *AC* = *assoc commute left-commute*

lemma (**in** *ACe*) *left-ident* [*simp*]: $e \cdot x = x$
⟨*proof*⟩

lemma (**in** *ACIf*) *idem2*: $x \cdot (x \cdot y) = x \cdot y$
⟨*proof*⟩

lemmas (**in** *ACIf*) *ACI* = *AC idem idem2*

Interpretation of locales:

interpretation *AC-add*: *ACe* [*op* + *0*::′*a*::*comm-monoid-add*]
⟨*proof*⟩

interpretation *AC-mult*: *ACe* [*op* ∗ *1*::′*a*::*comm-monoid-mult*]
⟨*proof*⟩

21.2.2 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* $(h\ m)$ $A = h$ ‘ $\{i.\ i < Suc\ m\}$; $h\ m \notin A$;
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*: $a \notin A$
and *inj-on*: *inj-on* h $\{i{::}nat.\ i{<}n\}$
shows $\exists$ *hm m*. *inj-on hm* $\{i{::}nat.\ i{<}m\}$ & $A = hm$ ‘ $\{i.\ i{<}m\}$ & $m < n$
⟨*proof*⟩

lemma (**in** *ACf*) *foldSet-determ-aux*:
!!$A\ x\ x'\ h$. ⟦ $A = h$‘$\{i{::}nat.\ i{<}n\}$; *inj-on* h $\{i.\ i{<}n\}$;
(A,x) : *foldSet f g z*; (A,x') : *foldSet f g z* ⟧
$\Longrightarrow x' = x$
⟨*proof*⟩

lemma (**in** *ACf*) *foldSet-determ*:
 (*A*,*x*) : *foldSet f g z* ==> (*A*,*y*) : *foldSet f g z* ==> *y* = *x*
⟨*proof*⟩

lemma (**in** *ACf*) *fold-equality*: (*A*, *y*) : *foldSet f g z* ==> *fold f g z A* = *y*
 ⟨*proof*⟩

The base case for *fold*:

lemma *fold-empty* [*simp*]: *fold f g z* {} = *z*
 ⟨*proof*⟩

lemma (**in** *ACf*) *fold-insert-aux*: $x \notin A$ ==>
 ((*insert x A*, *v*) : *foldSet f g z*) =
 (*EX y*. (*A*, *y*) : *foldSet f g z* & *v* = *f* (*g x*) *y*)
 ⟨*proof*⟩

The recursion equation for *fold*:

lemma (**in** *ACf*) *fold-insert*[*simp*]:
 finite A ==> $x \notin A$ ==> *fold f g z* (*insert x A*) = *f* (*g x*) (*fold f g z A*)
 ⟨*proof*⟩

lemma (**in** *ACf*) *fold-rec*:
assumes *fin*: *finite A* **and** *a*: *a*:*A*
shows *fold f g z A* = *f* (*g a*) (*fold f g z* (*A* − {*a*}))
⟨*proof*⟩

A simplified version for idempotent functions:

lemma (**in** *ACIf*) *fold-insert-idem*:
assumes *finA*: *finite A*
shows *fold f g z* (*insert a A*) = *g a* · *fold f g z A*
⟨*proof*⟩

lemma (**in** *ACIf*) *foldI-conv-id*:
 finite A ⟹ *fold f g z A* = *fold f id z* (*g* ‘ *A*)
⟨*proof*⟩

21.2.3 Lemmas about *fold*

lemma (**in** *ACf*) *fold-commute*:
 finite A ==> (!!*z*. *f x* (*fold f g z A*) = *fold f g* (*f x z*) *A*)
 ⟨*proof*⟩

lemma (**in** *ACf*) *fold-nest-Un-Int*:
 finite A ==> *finite B*
 ==> *fold f g* (*fold f g z B*) *A* = *fold f g* (*fold f g z* (*A Int B*)) (*A Un B*)
 ⟨*proof*⟩

lemma (**in** *ACf*) *fold-nest-Un-disjoint*:
 finite A ==> *finite B* ==> *A Int B* = {}

$==>$ *fold f g z* $(A\ \ Un\ B) =$ *fold f g* (*fold f g z B*) A
⟨*proof*⟩

lemma (**in** *ACf*) *fold-reindex*:
assumes *fin*: *finite A*
shows *inj-on h A* $\Longrightarrow$ *fold f g z* $(h\ `\ A) =$ *fold f* $(g \circ h)\ z\ A$
⟨*proof*⟩

lemma (**in** *ACe*) *fold-Un-Int*:
finite A $==>$ *finite B* $==>$
fold f g e A $\cdot$ *fold f g e B* $=$
fold f g e $(A\ \ Un\ B) \cdot$ *fold f g e* $(A\ \ Int\ B)$
⟨*proof*⟩

corollary (**in** *ACe*) *fold-Un-disjoint*:
finite A $==>$ *finite B* $==>$ $A\ \ Int\ B = \{\}$ $==>$
fold f g e $(A\ \ Un\ B) =$ *fold f g e A* $\cdot$ *fold f g e B*
⟨*proof*⟩

lemma (**in** *ACe*) *fold-UN-disjoint*:
$[\![$ *finite I*; *ALL i*:*I*. *finite* $(A\ i)$;
ALL i:*I*. *ALL j*:*I*. $i \neq j$ $-->$ $A\ i\ Int\ A\ j = \{\}$ $]\!]$
$\Longrightarrow$ *fold f g e* (*UNION I A*) $=$
fold f (%*i*. *fold f g e* $(A\ i)$) *e I*
⟨*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 (**in** *ACf*) *fold-fusion*:
includes *ACf g*
shows
finite A $==>$
(!!$x\ y$. $h\ (g\ x\ y) = f\ x\ (h\ y)$) $==>$
h (*fold g j w A*) $=$ *fold f j* $(h\ w)$ A
⟨*proof*⟩

lemma (**in** *ACf*) *fold-cong*:
finite A $\Longrightarrow$ (!!x. x:A $==>$ $g\ x = h\ x$) $==>$ *fold f g z A* $=$ *fold f h z A*
⟨*proof*⟩

lemma (**in** *ACe*) *fold-Sigma*: *finite A* $==>$ *ALL x*:*A*. *finite* $(B\ x)$ $==>$
fold f (%*x*. *fold f* $(g\ x)$ *e* $(B\ x)$) *e A* $=$
fold f (*split g*) *e* (*SIGMA x*:*A*. *B x*)
⟨*proof*⟩

lemma (**in** *ACe*) *fold-distrib*: *finite A* $\Longrightarrow$
fold f (%*x*. $f\ (g\ x)\ (h\ x)$) *e A* $= f$ (*fold f g e A*) (*fold f h e A*)
⟨*proof*⟩

21.3 Generalized summation over a set

constdefs
setsum :: $('a => 'b) => 'a\ set => 'b$::*comm-monoid-add*
setsum f A == *if finite A then fold* $(op\ +)\ f\ 0\ A$ *else 0*

Now: lot's of fancy syntax. First, *setsum* $(\lambda x.\ e)\ A$ is written $\sum x{\in}A.\ e$.

syntax
-setsum :: *pttrn* => *'a set* => *'b* => *'b::comm-monoid-add* ((*3SUM* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setsum :: *pttrn* => *'a set* => *'b* => *'b::comm-monoid-add* ((*3*$\sum$ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setsum :: *pttrn* => *'a set* => *'b* => *'b::comm-monoid-add* ((*3*$\sum$ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM i:A. b == *setsum* (%*i. b*) *A*
$\sum i{\in}A.\ b$ == *setsum* (%*i. b*) *A*

Instead of $\sum x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\sum x|P.\ e$.

syntax
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3SUM* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\sum$ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\sum$ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
SUM x|P. t => *setsum* (%*x. t*) {*x. P*}
$\sum x|P.\ t$ => *setsum* (%*x. t*) {*x. P*}

Finally we abbreviate $\sum x{\in}A.\ x$ by $\sum A$.

syntax
-Setsum :: *'a set* => *'a::comm-monoid-mult* ($\sum$ - [*1000*] *999*)

⟨*ML*⟩

lemma *setsum-empty* [*simp*]: *setsum f* {} = *0*
⟨*proof*⟩

lemma *setsum-insert* [*simp*]:
finite F ==> $a \notin F$ ==> *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*:
$\quad$ *inj-on* $f\ B$ ==> *setsum* $h\ (f\ `\ B)$ = *setsum* $(h \circ f)\ B$
⟨*proof*⟩

lemma *setsum-reindex-id*:
$\quad$ *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*:
$\quad$ [|*inj-on* $f\ A$; $B = f\ `\ A$; !!a. a:$A \Longrightarrow g\ a = h\ (f\ a)$|]
$\quad$ ==> *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*:
$\quad$ *finite* I ==> (*ALL* i:I. *finite* ($A\ i$)) ==>
$\quad\quad$ (*ALL* i:I. *ALL* j:I. $i \neq j$ --> $A\ i$ *Int* $A\ j$ = {}) ==>
$\quad$ *setsum* f (*UNION* $I\ A$) = ($\sum i{\in}I.$ *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*:
$[| \ (ALL\ A{:}C.\ finite\ A);$
$(ALL\ A{:}C.\ ALL\ B{:}C.\ A \neq B\ {-}{-}{>}\ A\ Int\ B = \{\})\ |]$
$==>\ setsum\ f\ (Union\ C) = setsum\ (setsum\ f)\ C$
⟨*proof*⟩

lemma *setsum-Sigma*: $finite\ A ==> ALL\ x{:}A.\ finite\ (B\ x) ==>$
$(\sum x{\in}A.\ (\sum y{\in}B\ x.\ f\ x\ y)) = (\sum (x,y){\in}(SIGMA\ x{:}A.\ B\ x).\ f\ x\ y)$
⟨*proof*⟩

Here we can eliminate the finiteness assumptions, by cases.

lemma *setsum-cartesian-product*:
$(\sum x{\in}A.\ (\sum y{\in}B.\ f\ x\ y)) = (\sum (x,y) \in A <*> B.\ f\ x\ y)$
⟨*proof*⟩

lemma *setsum-addf*: $setsum\ (\%x.\ f\ x + g\ x)\ A = (setsum\ f\ A + setsum\ g\ A)$
⟨*proof*⟩

21.3.1 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*: $finite\ A ==> finite\ B ==>$
$(setsum\ f\ (A\ Un\ B) :: 'a :: ab\text{-}group\text{-}add) =$
$setsum\ f\ A + setsum\ f\ B - setsum\ f\ (A\ Int\ B)$
⟨*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*: $finite\ A \Longrightarrow$
$(setsum\ f\ (A - \{a\}) :: ('a{::}ab\text{-}group\text{-}add)) =$
$(if\ a{:}A\ then\ setsum\ f\ A - f\ a\ else\ setsum\ f\ A)$
⟨*proof*⟩

lemma *setsum-diff1'*[*rule-format*]: $finite\ A \Longrightarrow a \in A \longrightarrow (\sum\ x \in A.\ f\ x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x)$

⟨*proof*⟩

lemma *setsum-diff-nat*:
assumes *finB*: *finite B*
shows $B \subseteq A \Longrightarrow (setsum\ f\ (A - B) :: nat) = (setsum\ f\ A) - (setsum\ f\ B)$
⟨*proof*⟩

lemma *setsum-diff*:
assumes *le*: *finite A* $B \subseteq A$
shows $setsum\ f\ (A - B) = setsum\ f\ A - ((setsum\ f\ B)::('a::ab\text{-}group\text{-}add))$
⟨*proof*⟩

lemma *setsum-mono*:
assumes *le*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i::'a) \leq ((g\ i)::('b::\{comm\text{-}monoid\text{-}add,\ pordered\text{-}ab\text{-}semigroup\text{-}add\}))$
shows $(\sum i{\in}K.\ f\ i) \leq (\sum i{\in}K.\ g\ i)$
⟨*proof*⟩

lemma *setsum-strict-mono*:
fixes $f :: 'a \Rightarrow 'b::\{pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add,comm\text{-}monoid\text{-}add\}$
assumes *fin-ne*: *finite A* $A \neq \{\}$
shows $(!!x.\ x{:}A \Longrightarrow f\ x < g\ x) \Longrightarrow setsum\ f\ A < setsum\ g\ A$
⟨*proof*⟩

lemma *setsum-negf*:
$setsum\ (\%x.\ -\ (f\ x)::'a::ab\text{-}group\text{-}add)\ A = -\ setsum\ f\ A$
⟨*proof*⟩

lemma *setsum-subtractf*:
$setsum\ (\%x.\ ((f\ x)::'a::ab\text{-}group\text{-}add) - g\ x)\ A =$
$setsum\ f\ A - setsum\ g\ A$
⟨*proof*⟩

lemma *setsum-nonneg*:
assumes *nn*: $\forall x{\in}A.\ (0::'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add,comm\text{-}monoid\text{-}add\}) \leq f$
x
shows $0 \leq setsum\ f\ A$
⟨*proof*⟩

lemma *setsum-nonpos*:
assumes *np*: $\forall x{\in}A.\ f\,x \leq (0::'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add,comm\text{-}monoid\text{-}add\})$
shows $setsum\ f\ A \leq 0$
⟨*proof*⟩

lemma *setsum-mono2*:
fixes $f :: 'a \Rightarrow 'b :: \{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le,comm\text{-}monoid\text{-}add\}$
assumes *fin*: *finite B* **and** *sub*: $A \subseteq B$ **and** *nn*: $\bigwedge b.\ b \in B{-}A \Longrightarrow 0 \leq f\ b$
shows $setsum\ f\ A \leq setsum\ f\ B$

⟨*proof*⟩

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-mult*:
fixes *f* :: ′*a* => (′*b*::*semiring-0-cancel*)
shows *r* ∗ *setsum f A* = *setsum* (%*n*. *r* ∗ *f n*) *A*
⟨*proof*⟩

lemma *setsum-left-distrib*:
setsum f A ∗ (*r*::′*a*::*semiring-0-cancel*) = ($\sum$ *n*∈*A*. *f n* ∗ *r*)
⟨*proof*⟩

lemma *setsum-divide-distrib*:
setsum f A / (*r*::′*a*::*field*) = ($\sum$ *n*∈*A*. *f n* / *r*)
⟨*proof*⟩

lemma *setsum-abs*[*iff*]:
fixes *f* :: ′*a* => (′*b*::*lordered-ab-group-abs*)
shows *abs* (*setsum f A*) ≤ *setsum* (%*i*. *abs*(*f i*)) *A*
⟨*proof*⟩

lemma *setsum-abs-ge-zero*[*iff*]:
fixes *f* :: ′*a* => (′*b*::*lordered-ab-group-abs*)
shows *0* ≤ *setsum* (%*i*. *abs*(*f i*)) *A*
⟨*proof*⟩

lemma *abs-setsum-abs*[*simp*]:
fixes *f* :: ′*a* => (′*b*::*lordered-ab-group-abs*)
shows *abs* ($\sum$ *a*∈*A*. *abs*(*f a*)) = ($\sum$ *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*:

$(\sum i{\in}A.\ \sum j{\in}B.\ f\ i\ j) = (\sum j{\in}B.\ \sum i{\in}A.\ f\ i\ j)$
⟨*proof*⟩

21.4 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*

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 $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3PROD* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*∏- | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*∏- | (-)./ -) [*0*,*0*,*10*] *10*)

translations
PROD x|*P*. *t* => *setprod* (%*x*. *t*) {*x*. *P*}
∏*x*|*P*. *t* => *setprod* (%*x*. *t*) {*x*. *P*}

Finally we abbreviate $\prod x{\in}A.\ x$ by $\prod A$.

syntax
-Setprod :: *'a set* => *'a*::*comm-monoid-mult* (∏- [*1000*] *999*)

⟨*ML*⟩

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 ≠ j --> 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 ≠ B --> 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*⟩

21.4.1 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*:
$finite\ A ==> EX\ x{:}\ A.\ f\ x = (0{::}'a{::}comm\text{-}semiring\text{-}1\text{-}cancel) ==> setprod\ f\ A = 0$
⟨*proof*⟩

lemma *setprod-nonneg* [*rule-format*]:
$(ALL\ x{:}\ A.\ (0{::}'a{::}ordered\text{-}idom) \le f\ x) --> 0 \le setprod\ f\ A$
⟨*proof*⟩

lemma *setprod-pos* [*rule-format*]: $(ALL\ x{:}\ A.\ (0{::}'a{::}ordered\text{-}idom) < f\ x)$
$--> 0 < setprod\ f\ A$
⟨*proof*⟩

lemma *setprod-nonzero* [*rule-format*]:
$(ALL\ x\ y.\ (x{::}'a{::}comm\text{-}semiring\text{-}1\text{-}cancel) * y = 0 --> x = 0\ |\ y = 0) ==>$
$finite\ A ==> (ALL\ x{:}\ A.\ f\ x \neq (0{::}'a)) --> setprod\ f\ A \neq 0$
⟨*proof*⟩

lemma *setprod-zero-eq*:
$(ALL\ x\ y.\ (x{::}'a{::}comm\text{-}semiring\text{-}1\text{-}cancel) * y = 0 --> x = 0\ |\ y = 0) ==>$
$finite\ A ==> (setprod\ f\ A = (0{::}'a)) = (EX\ x{:}\ A.\ f\ x = 0)$
⟨*proof*⟩

lemma *setprod-nonzero-field*:
$finite\ A ==> (ALL\ x{:}\ A.\ f\ x \neq (0{::}'a{::}field)) ==> setprod\ f\ A \neq 0$
⟨*proof*⟩

lemma *setprod-zero-eq-field*:
$finite\ A ==> (setprod\ f\ A = (0{::}'a{::}field)) = (EX\ x{:}\ A.\ f\ x = 0)$

⟨*proof*⟩

lemma *setprod-Un*: *finite A ==> finite B ==> (ALL x: A Int B. f x* $\neq$ *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* $\neq$ *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 $\neq$ *(0::'a::{field,division-by-zero}) ==>*
setprod (inverse $\circ$ *f) A = inverse (setprod f A)*
⟨*proof*⟩

lemma *setprod-dividef*:
[|finite A;
$\forall$ *x* $\in$ *A. g x* $\neq$ *(0::'a::{field,division-by-zero})|]*
==> setprod (%x. f x / g x) A = setprod f A / setprod g A
⟨*proof*⟩

21.5 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:

constdefs
card :: 'a set => nat
card A == setsum (%x. 1::nat) 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 $\notin$ *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*]: *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-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*; $A \subseteq B$ $]\!] \Longrightarrow$ *card A* $\leq$ *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 ==>* $A \subseteq B$ *==> card A < card B ==> A < B*
⟨*proof*⟩

lemma *insert-partition*:
$[\![x \notin F; \forall c1 \in insert\ x\ F.\ \forall c2 \in insert\ x\ F.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}]\!]$
$\Longrightarrow x \cap \bigcup F = \{\}$
⟨*proof*⟩

lemma *card-partition* [*rule-format*]:
finite C ==>
finite $(\bigcup C)$ *-->*
$(\forall c{\in}C.\ card\ c = k)$ *-->*
$(\forall c1 \in C.\ \forall c2 \in C.\ c1 \neq c2$ *-->* $c1 \cap c2 = \{\})$ *-->*
$k * card(C) = card\ (\bigcup C)$
⟨*proof*⟩

lemma *setsum-constant* [*simp*]: $(\sum x \in A.\ y) = of\text{-}nat(card\ A) * y$
⟨*proof*⟩

lemma *setprod-constant*: *finite A ==>* $(\prod x{\in} A.\ (y{::}'a{::}recpower)) = y\hat{}(card\ A)$
⟨*proof*⟩

lemma *setsum-bounded*:
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\ i \leq (K{::}'a{::}\{comm\text{-}semiring\text{-}1\text{-}cancel, pordered\text{-}ab\text{-}semigroup\text{-}add\})$
shows $setsum\ f\ A \leq of\text{-}nat(card\ A) * K$
⟨*proof*⟩

21.5.1 Cardinality of unions

lemma *of-nat-id*[*simp*]: $(of\text{-}nat\ n :: nat) = n$
⟨*proof*⟩

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) ==>

$(ALL\ A{:}C.\ ALL\ B{:}C.\ A \neq B\ {-}{-}{>}\ A\ Int\ B = \{\}) ==>$
$card\ (Union\ C) = setsum\ card\ C$
$\langle proof \rangle$

21.5.2 Cardinality of image

The image of a finite set can be expressed using *fold*.

lemma *image-eq-fold*: $finite\ A ==> f\ `\ A = fold\ (op\ Un)\ (\%x.\ \{f\ x\})\ \{\}\ A$
$\langle proof \rangle$

lemma *card-image-le*: $finite\ A ==> card\ (f\ `\ A) <= card\ A$
$\langle proof \rangle$

lemma *card-image*: $inj\text{-}on\ f\ A ==> card\ (f\ `\ A) = card\ A$
$\langle proof \rangle$

lemma *endo-inj-surj*: $finite\ A ==> f\ `\ A \subseteq A ==> inj\text{-}on\ f\ A ==> f\ `\ A = A$
$\langle proof \rangle$

lemma *eq-card-imp-inj-on*:
$[|\ finite\ A;\ card(f\ `\ A) = card\ A\ |] ==> inj\text{-}on\ f\ A$
$\langle proof \rangle$

lemma *inj-on-iff-eq-card*:
$finite\ A ==> inj\text{-}on\ f\ A = (card(f\ `\ A) = card\ A)$
$\langle proof \rangle$

lemma *card-inj-on-le*:
$[|inj\text{-}on\ f\ A;\ f\ `\ A \subseteq B;\ finite\ B\ |] ==> card\ A \leq card\ B$
$\langle proof \rangle$

lemma *card-bij-eq*:
$[|inj\text{-}on\ f\ A;\ f\ `\ A \subseteq B;\ inj\text{-}on\ g\ B;\ g\ `\ B \subseteq A;$
$finite\ A;\ finite\ B\ |] ==> card\ A = card\ B$
$\langle proof \rangle$

21.5.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
$[\![\ finite\ A;\ ALL\ a{:}A.\ finite\ (B\ a)\ ]\!]$
$\Longrightarrow card\ (SIGMA\ x{:}\ A.\ B\ x) = (\sum a{\in}A.\ card\ (B\ a))$
$\langle proof \rangle$

lemma *card-cartesian-product*: $card\ (A <{*}> B) = card(A) * card(B)$
$\langle proof \rangle$

lemma *card-cartesian-product-singleton*: $card(\{x\} <{*}> A) = card(A)$
$\langle proof \rangle$

21.5.4 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's.

lemma *dvd-partition*:
finite (Union C) ==>
ALL c : C. k dvd card c ==>
(ALL c1: C. ALL c2: C. c1 $\neq$ *c2 --> c1 Int c2 = {}) ==>*
k dvd card (Union C)
⟨*proof*⟩

21.6 A fold functional for non-empty sets

Does not require start value.

consts
fold1Set :: *('a => 'a => 'a) => ('a set* $\times$ *'a) set*

inductive *fold1Set f*
intros
fold1Set-insertI [*intro*]:
⟦ *(A,x)* $\in$ *foldSet f id a*; *a* $\notin$ *A* ⟧ $\Longrightarrow$ *(insert a A, x)* $\in$ *fold1Set f*

constdefs
fold1 :: *('a => 'a => 'a) => 'a set => 'a*
fold1 f A == THE x. (A, x) : fold1Set f

lemma *fold1Set-nonempty*:
(A, x) : fold1Set f $\Longrightarrow$ *A* $\neq$ *{}*
⟨*proof*⟩

inductive-cases *empty-fold1SetE* [*elim!*]: *({}, x) : fold1Set f*

inductive-cases *insert-fold1SetE* [*elim!*]: *(insert a X, x) : fold1Set f*

lemma *fold1Set-sing* [*iff*]: *(({a},b) : fold1Set f) = (a = b)*
⟨*proof*⟩

lemma *fold1-singleton*[*simp*]: *fold1 f {a} = a*
⟨*proof*⟩

lemma *finite-nonempty-imp-fold1Set*:
⟦ *finite A*; *A* $\neq$ *{}* ⟧ $\Longrightarrow$ *EX x. (A, x) : fold1Set f*
⟨*proof*⟩

First, some lemmas about *foldSet*.

lemma (**in** ACf) *foldSet-insert-swap*:
assumes *fold*: $(A,y) \in foldSet\ f\ id\ b$
shows $b \notin A \Longrightarrow (insert\ b\ A,\ z \cdot y) \in foldSet\ f\ id\ z$
$\langle proof \rangle$

lemma (**in** ACf) *foldSet-permute-diff*:
assumes *fold*: $(A,x) \in foldSet\ f\ id\ b$
shows $!!a.\ [\![a \in A;\ b \notin A]\!] \Longrightarrow (insert\ b\ (A-\{a\}),\ x) \in foldSet\ f\ id\ a$
$\langle proof \rangle$

lemma (**in** ACf) *fold1-eq-fold*:
$[|finite\ A;\ a \notin A|] ==> fold1\ f\ (insert\ a\ A) = fold\ f\ id\ a\ A$
$\langle proof \rangle$

lemma *nonempty-iff*: $(A \neq \{\}) = (\exists\, x\ B.\ A = insert\ x\ B\ \&\ x \notin B)$
$\langle proof \rangle$

lemma (**in** ACf) *fold1-insert*:
assumes *nonempty*: $A \neq \{\}$ **and** A: $finite\ A\ x \notin A$
shows $fold1\ f\ (insert\ x\ A) = f\ x\ (fold1\ f\ A)$
$\langle proof \rangle$

lemma (**in** $ACIf$) *fold1-insert-idem* [*simp*]:
assumes *nonempty*: $A \neq \{\}$ **and** A: $finite\ A$
shows $fold1\ f\ (insert\ x\ A) = f\ x\ (fold1\ f\ A)$
$\langle proof \rangle$

Now the recursion rules for definitions:

lemma *fold1-singleton-def*: $g \equiv fold1\ f \Longrightarrow g\ \{a\} = a$
$\langle proof \rangle$

lemma (**in** ACf) *fold1-insert-def*:
$[\![\ g \equiv fold1\ f;\ finite\ A;\ x \notin A;\ A \neq \{\}\ ]\!] \Longrightarrow g(insert\ x\ A) = x \cdot (g\ A)$
$\langle proof \rangle$

lemma (**in** $ACIf$) *fold1-insert-idem-def*:
$[\![\ g \equiv fold1\ f;\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow g(insert\ x\ A) = x \cdot (g\ A)$
$\langle proof \rangle$

21.6.1 Determinacy for *fold1Set*

Not actually used!!

lemma (**in** ACf) *foldSet-permute*:
$[|(insert\ a\ A,\ x) \in foldSet\ f\ id\ b;\ a \notin A;\ b \notin A|]$
$==> (insert\ b\ A,\ x) \in foldSet\ f\ id\ a$
$\langle proof \rangle$

lemma (**in** ACf) *fold1Set-determ*:
$(A,\ x) \in fold1Set\ f ==> (A,\ y) \in fold1Set\ f ==> y = x$

⟨*proof*⟩

lemma (**in** *ACf*) *fold1Set-equality*: $(A,\ y) : \mathit{fold1Set}\ f\ {=}{=}{>}\ \mathit{fold1}\ f\ A = y$
⟨*proof*⟩

declare
empty-foldSetE [*rule del*] *foldSet.intros* [*rule del*]
empty-fold1SetE [*rule del*] *insert-fold1SetE* [*rule del*]
— No more proves involve these relations.

21.6.2 Semi-Lattices

locale *ACIfSL* = *ACIf* +
fixes *below* :: $'a \Rightarrow 'a \Rightarrow \mathit{bool}$ (**infixl** $\sqsubseteq$ *50*)
assumes *below-def*: $(x \sqsubseteq y) = (x{\cdot}y = x)$

locale *ACIfSLlin* = *ACIfSL* +
assumes *lin*: $x{\cdot}y \in \{x,y\}$

lemma (**in** *ACIfSL*) *below-refl*[*simp*]: $x \sqsubseteq x$
⟨*proof*⟩

lemma (**in** *ACIfSL*) *below-f-conv*[*simp*]: $x \sqsubseteq y \cdot z = (x \sqsubseteq y \wedge x \sqsubseteq z)$
⟨*proof*⟩

lemma (**in** *ACIfSLlin*) *above-f-conv*:
$x \cdot y \sqsubseteq z = (x \sqsubseteq z \vee y \sqsubseteq z)$
⟨*proof*⟩

21.6.3 Lemmas about *fold1*

lemma (**in** *ACf*) *fold1-Un*:
assumes *A*: *finite* $A\ A \neq \{\}$
shows *finite* $B \Longrightarrow B \neq \{\} \Longrightarrow A\ \mathit{Int}\ B = \{\} \Longrightarrow$
$\mathit{fold1}\ f\ (A\ \mathit{Un}\ B) = f\ (\mathit{fold1}\ f\ A)\ (\mathit{fold1}\ f\ B)$
⟨*proof*⟩

lemma (**in** *ACIf*) *fold1-Un2*:
assumes *A*: *finite* $A\ A \neq \{\}$
shows *finite* $B \Longrightarrow B \neq \{\} \Longrightarrow$
$\mathit{fold1}\ f\ (A\ \mathit{Un}\ B) = f\ (\mathit{fold1}\ f\ A)\ (\mathit{fold1}\ f\ B)$
⟨*proof*⟩

lemma (**in** *ACf*) *fold1-in*:
assumes *A*: *finite* $(A)\ A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x{\cdot}y \in \{x,y\}$
shows $\mathit{fold1}\ f\ A \in A$
⟨*proof*⟩

lemma (**in** *ACIfSL*) *below-fold1-iff*:
assumes *A*: *finite* $A\ A \neq \{\}$

shows $x \sqsubseteq \mathit{fold1}\ f\ A = (\forall\, a{\in}A.\ x \sqsubseteq a)$
$\langle \mathit{proof} \rangle$

lemma (**in** *ACIfSL*) *fold1-belowI*:
assumes A: $\mathit{finite}\ A\ \ A \neq \{\}$
shows $a \in A \Longrightarrow \mathit{fold1}\ f\ A \sqsubseteq a$
$\langle \mathit{proof} \rangle$

lemma (**in** *ACIfSLlin*) *fold1-below-iff*:
assumes A: $\mathit{finite}\ A\ \ A \neq \{\}$
shows $\mathit{fold1}\ f\ A \sqsubseteq x = (\exists\, a{\in}A.\ a \sqsubseteq x)$
$\langle \mathit{proof} \rangle$

21.6.4 Lattices

locale *Lattice* = *lattice* +
  fixes Inf :: $'a\ \mathit{set} \Rightarrow {'a}$ ($\bigsqcap$- [900] 900)
  and Sup :: $'a\ \mathit{set} \Rightarrow {'a}$ ($\bigsqcup$- [900] 900)
  defines $\mathit{Inf} == \mathit{fold1}\ \mathit{inf}$ **and** $\mathit{Sup} == \mathit{fold1}\ \mathit{sup}$

locale *Distrib-Lattice* = *distrib-lattice* + *Lattice*

Lattices are semilattices

lemma (**in** *Lattice*) *ACf-inf*: *ACf inf*
$\langle \mathit{proof} \rangle$

lemma (**in** *Lattice*) *ACf-sup*: *ACf sup*
$\langle \mathit{proof} \rangle$

lemma (**in** *Lattice*) *ACIf-inf*: *ACIf inf*
$\langle \mathit{proof} \rangle$

lemma (**in** *Lattice*) *ACIf-sup*: *ACIf sup*
$\langle \mathit{proof} \rangle$

lemma (**in** *Lattice*) *ACIfSL-inf*: $\mathit{ACIfSL}\ \mathit{inf}\ (\mathit{op} \sqsubseteq)$
$\langle \mathit{proof} \rangle$

lemma (**in** *Lattice*) *ACIfSL-sup*: $\mathit{ACIfSL}\ \mathit{sup}\ (\%x\ y.\ y \sqsubseteq x)$
$\langle \mathit{proof} \rangle$

21.6.5 Fold laws in lattices

lemma (**in** *Lattice*) *Inf-le-Sup*[*simp*]: $[\![\ \mathit{finite}\ A;\ A \neq \{\}\ ]\!] \Longrightarrow \bigsqcap A \sqsubseteq \bigsqcup A$
$\langle \mathit{proof} \rangle$

lemma (**in** *Lattice*) *sup-Inf-absorb*[*simp*]:
  $[\![\ \mathit{finite}\ A;\ A \neq \{\};\ a \in A\ ]\!] \Longrightarrow (a \sqcup \bigsqcap A) = a$
$\langle \mathit{proof} \rangle$

lemma (**in** *Lattice*) *inf-Sup-absorb*[*simp*]:
$[\![$ *finite* A; $A \neq \{\}$; $a \in A$ $]\!] \Longrightarrow (a \sqcap \bigsqcup A) = a$
⟨*proof*⟩

lemma (**in** *Distrib-Lattice*) *sup-Inf1-distrib*:
assumes A: *finite* A $A \neq \{\}$
shows $(x \sqcup \bigsqcap A) = \bigsqcap\{x \sqcup a|a.\ a \in A\}$
⟨*proof*⟩

lemma (**in** *Distrib-Lattice*) *sup-Inf2-distrib*:
assumes A: *finite* A $A \neq \{\}$ **and** B: *finite* B $B \neq \{\}$
shows $(\bigsqcap A \sqcup \bigsqcap B) = \bigsqcap\{a \sqcup b|a\ b.\ a \in A \wedge b \in B\}$
⟨*proof*⟩

21.7 Min and Max

As an application of *fold1* we define the minimal and maximal element of a (non-empty) set over a linear order.

constdefs
Min :: (*'a*::*linorder*)*set* => *'a*
Min == *fold1 min*

Max :: (*'a*::*linorder*)*set* => *'a*
Max == *fold1 max*

Before we can do anything, we need to show that *min* and *max* are ACI and the ordering is linear:

interpretation *min*: *ACf* [*min*:: *'a*::*linorder* ⇒ *'a* ⇒ *'a*]
⟨*proof*⟩

interpretation *min*: *ACIf* [*min*:: *'a*::*linorder* ⇒ *'a* ⇒ *'a*]
⟨*proof*⟩

interpretation *max*: *ACf* [*max* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a*]
⟨*proof*⟩

interpretation *max*: *ACIf* [*max*:: *'a*::*linorder* ⇒ *'a* ⇒ *'a*]
⟨*proof*⟩

interpretation *min*:
ACIfSL [*min*:: *'a*::*linorder* ⇒ *'a* ⇒ *'a* *op* ≤]
⟨*proof*⟩

interpretation *min*:
ACIfSLlin [*min* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a* *op* ≤]
⟨*proof*⟩

interpretation *max*:
  $ACIfSL$ [max :: $'a$::$linorder \Rightarrow 'a \Rightarrow 'a$ %x y. $y \leq x$]
⟨*proof*⟩

interpretation *max*:
  $ACIfSLlin$ [max :: $'a$::$linorder \Rightarrow 'a \Rightarrow 'a$ %x y. $y \leq x$]
⟨*proof*⟩

interpretation *min-max*:
  Lattice [$op \leq$ min :: $'a$::$linorder \Rightarrow 'a \Rightarrow 'a$ max Min Max]
⟨*proof*⟩

interpretation *min-max*:
  Distrib-Lattice [$op \leq$ min :: $'a$::$linorder \Rightarrow 'a \Rightarrow 'a$ max Min Max]
⟨*proof*⟩

Now we instantiate the recursion equations and declare them simplification rules:

lemmas *Min-singleton* = *fold1-singleton-def* [*OF Min-def*]
lemmas *Max-singleton* = *fold1-singleton-def* [*OF Max-def*]
lemmas *Min-insert* = *min.fold1-insert-idem-def* [*OF Min-def*]
lemmas *Max-insert* = *max.fold1-insert-idem-def* [*OF Max-def*]

declare *Min-singleton* [*simp*] *Max-singleton* [*simp*]
declare *Min-insert* [*simp*] *Max-insert* [*simp*]

Now we instantiate some *fold1* properties:

lemma *Min-in* [*simp*]:
  shows $finite\ A \Longrightarrow A \neq \{\} \Longrightarrow Min\ A \in A$
⟨*proof*⟩

lemma *Max-in* [*simp*]:
  shows $finite\ A \Longrightarrow A \neq \{\} \Longrightarrow Max\ A \in A$
⟨*proof*⟩

lemma *Min-le* [*simp*]: $[\![\ finite\ A;\ A \neq \{\};\ x \in A\ ]\!] \Longrightarrow Min\ A \leq x$
⟨*proof*⟩

lemma *Max-ge* [*simp*]: $[\![\ finite\ A;\ A \neq \{\};\ x \in A\ ]\!] \Longrightarrow x \leq Max\ A$
⟨*proof*⟩

lemma *Min-ge-iff* [*simp*]:
  $[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow (x \leq Min\ A) = (\forall\, a \in A.\ x \leq a)$
⟨*proof*⟩

lemma *Max-le-iff* [*simp*]:
  $[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow (Max\ A \leq x) = (\forall\, a \in A.\ a \leq x)$
⟨*proof*⟩

lemma *Min-le-iff*:
$[\![$ *finite* A; $A \neq \{\}$ $]\!] \Longrightarrow (Min\ A \leq x) = (\exists\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *Max-ge-iff*:
$[\![$ *finite* A; $A \neq \{\}$ $]\!] \Longrightarrow (x \leq Max\ A) = (\exists\, a{\in}A.\ x \leq a)$
⟨*proof*⟩

21.8 Properties of axclass *finite*

Many of these are by Brian Huffman.

lemma *finite-set*: *finite* (A::$'a$::*finite set*)
⟨*proof*⟩

instance *unit* :: *finite*
⟨*proof*⟩

instance *bool* :: *finite*
⟨*proof*⟩

instance $*$ :: (*finite*, *finite*) *finite*
⟨*proof*⟩

instance $+$ :: (*finite*, *finite*) *finite*
⟨*proof*⟩

instance *set* :: (*finite*) *finite*
⟨*proof*⟩

lemma *inj-graph*: *inj* $(\%f.\ \{(x,\ y).\ y = f\, x\})$
⟨*proof*⟩

instance *fun* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

end

22 Wellfounded-Relations: Well-founded Relations

theory *Wellfounded-Relations*
imports *Finite-Set*
begin

Derived WF relations such as inverse image, lexicographic product and measure. The simple relational product, in which (x', y') precedes (x, y) if x' $< x$ and $y' < y$, is a subset of the lexicographic product, and therefore does not need to be defined separately.

constdefs
less-than :: (*nat*∗*nat*)*set*
less-than == *trancl pred-nat*

measure :: ('*a* => *nat*) => ('*a* ∗ '*a*)*set*
measure == *inv-image less-than*

lex-prod :: [('*a*∗'*a*)*set*, ('*b*∗'*b*)*set*] => (('*a*∗'*b*)∗('*a*∗'*b*))*set*
(**infixr** <∗*lex*∗> *80*)
ra <∗*lex*∗> *rb* == {((*a*,*b*),(*a*',*b*')). (*a*,*a*') : *ra* | *a*=*a*' & (*b*,*b*') : *rb*}

finite-psubset :: ('*a set* ∗ '*a set*) *set*
— finite proper subset
finite-psubset == {(*A*,*B*). *A* < *B* & *finite B*}

same-fst :: ('*a* => *bool*) => ('*a* => ('*b* ∗ '*b*)*set*) => (('*a*∗'*b*)∗('*a*∗'*b*))*set*
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.

22.1 Measure Functions make Wellfounded Relations

22.1.1 'Less than' on the natural numbers

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 *full-nat-induct*:
assumes *ih*: (!!*n*. (*ALL m*. *Suc m* <= *n* −−> *P m*) ==> *P n*)
shows *P n*
⟨*proof*⟩

22.1.2 The Inverse Image into a Wellfounded Relation is Wellfounded.

lemma *wf-inv-image* [*simp*,*intro*!]: *wf*(*r*) ==> *wf*(*inv-image r* (*f*::'*a*=>'*b*))
⟨*proof*⟩

22.1.3 Finally, All Measures are Wellfounded.

lemma wf-measure [iff]: wf (measure f)
⟨proof⟩

lemmas measure-induct =
    wf-measure [THEN wf-induct, unfolded measure-def inv-image-def,
        simplified, standard]

22.2 Other Ways of Constructing Wellfounded Relations

Wellfoundedness of lexicographic combinations

lemma wf-lex-prod [intro!]: [| wf(ra); wf(rb) |] ==> wf(ra <*lex*> rb)
⟨proof⟩

Transitivity of WF combinators.

lemma trans-lex-prod [intro!]:
    [| trans R1; trans R2 |] ==> trans (R1 <*lex*> R2)
⟨proof⟩

22.2.1 Wellfoundedness of proper subset on finite sets.

lemma wf-finite-psubset: wf(finite-psubset)
⟨proof⟩

lemma trans-finite-psubset: trans finite-psubset
⟨proof⟩

22.2.2 Wellfoundedness of finite acyclic relations

This proof belongs in this theory because it needs Finite.

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⟩

22.2.3 Wellfoundedness of *same-fst*

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-fst P R)

⟨*proof*⟩

22.3 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*ˆ∗ |] ==> (*f* (*i*+*k*), *f i*) : *r*ˆ∗
⟨*proof*⟩

lemma *lemma2*: [| *ALL i.* (*f* (*Suc i*), *f i*) : *r*ˆ∗; *wf* (*r*ˆ+) |]
==> *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*ˆ∗; *wf* (*r*ˆ+) |]
==> *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*⟩

⟨*ML*⟩

end

23 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *Relation Finite-Set*
begin

23.1 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*⟩

23.2 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*⟩

23.3 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 *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*: $\{x\}//r = \{r\ ``\ \{x\}\}$
⟨*proof*⟩

lemma *quotient-diff1*:
$[\![\ inj\text{-}on\ (\%a.\ \{a\}//r)\ A;\ a \in A\ ]\!] \Longrightarrow (A - \{a\})//r = A//r - \{a\}//r$
⟨*proof*⟩

23.4 Defining unary operations upon equivalence classes

A congruence-preserving function

locale *congruent* =
  fixes r **and** f
  assumes *congruent*: $(y,z) \in r$ ==> $f\ y = f\ z$

syntax
  RESPECTS ::['a => 'b, ('a * 'a) *set*] => *bool* (**infixr** *respects 80*)

translations
  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* ==> $a \in A$
    ==> $(\bigcup x \in r``\{a\}.\ f\ x) = f\ a$
  — 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. $y \in A$ ==> $f\ y \in B$.

lemma *UN-equiv-class-inject*:
  equiv A r ==> *f respects r* ==>
    $(\bigcup x \in X.\ f\ x) = (\bigcup y \in Y.\ f\ y)$ ==> $X \in A//r$ ==> $Y \in A//r$
    ==> (!!$x\ y$. $x \in A$ ==> $y \in A$ ==> $f\ x = f\ y$ ==> $(x,\ y) \in r$)
    ==> $X = Y$
  ⟨*proof*⟩

23.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

locale *congruent2* =
  fixes *r1* **and** *r2* **and** f
  assumes *congruent2*:
    $(y1,z1) \in r1$ ==> $(y2,z2) \in r2$ ==> $f\ y1\ y2 = f\ z1\ z2$

Abbreviation for the common case where the relations are identical

syntax

RESPECTS2 ::['a => 'b, ('a * 'a) set] => *bool* (**infixr** *respects2 80*)

translations
f respects2 r => congruent2 r r f

lemma *congruent2-implies-congruent*:
equiv A r1 ==> congruent2 r1 r2 f ==> a $\in$ *A ==> congruent r2 (f a)*
⟨*proof*⟩

lemma *congruent2-implies-congruent-UN*:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a $\in$ *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 $\in$ *A1 ==> a2* $\in$ *A2*
$==> (\bigcup x1 \in r1``\{a1\}. \bigcup x2 \in r2``\{a2\}. f\ x1\ x2) = f\ a1\ a2$
⟨*proof*⟩

lemma *UN-equiv-class-type2*:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f
==> X1 $\in$ *A1//r1 ==> X2* $\in$ *A2//r2*
==> (!!x1 x2. x1 $\in$ *A1 ==> x2* $\in$ *A2 ==> f x1 x2* $\in$ *B)*
$==> (\bigcup x1 \in X1. \bigcup x2 \in X2. f\ x1\ x2) \in B$
⟨*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 $\in$ *A2 ==> (y,z)* $\in$ *r1 ==> f y w = f z w)*
==> (!!y z w. w $\in$ *A1 ==> (y,z)* $\in$ *r2 ==> f w y = f w z)*
==> congruent2 r1 r2 f
— Suggested by John Harrison – the two subproofs may be
— *much* simpler than the direct proof.
⟨*proof*⟩

lemma *congruent2-commuteI*:
assumes *equivA*: *equiv A r*
and *commute*: *!!y z. y* $\in$ *A ==> z* $\in$ *A ==> f y z = f z y*
and *congt*: *!!y z w. w* $\in$ *A ==> (y,z)* $\in$ *r ==> f w y = f w z*
shows *f respects2 r*
⟨*proof*⟩

23.6 Cardinality results

Suggested by Florian Kammüller

lemma *finite-quotient*: $finite\ A\ ==>\ r \subseteq A \times A\ ==>\ finite\ (A//r)$
— recall $equiv\ ?A\ ?r \Longrightarrow ?r \subseteq ?A \times ?A$
⟨*proof*⟩

lemma *finite-equiv-class*:
$finite\ A\ ==>\ r \subseteq A \times A\ ==>\ X \in A//r\ ==>\ finite\ X$
⟨*proof*⟩

lemma *equiv-imp-dvd-card*:
$finite\ A\ ==>\ equiv\ A\ r\ ==>\ \forall X \in A//r.\ k\ dvd\ card\ X$
$==>\ k\ dvd\ card\ A$
⟨*proof*⟩

lemma *card-quotient-disjoint*:
$[\![\ finite\ A;\ inj\text{-}on\ (\lambda x.\ \{x\}\ //\ r)\ A\ ]\!] \Longrightarrow card(A//r) = card\ A$
⟨*proof*⟩

⟨*ML*⟩

end

24 IntDef: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *IntDef*
imports *Equiv-Relations NatArith*
begin

constdefs
$intrel :: ((nat * nat) * (nat * nat))\ set$
— the equivalence relation underlying the integers
$intrel == \{((x,y),(u,v))\ |\ x\ y\ u\ v.\ x+v = u+y\}$

typedef (*Integ*)
$int = UNIV//intrel$
⟨*proof*⟩

instance $int :: \{ord,\ zero,\ one,\ plus,\ times,\ minus\}$ ⟨*proof*⟩

constdefs
$int :: nat\ =>\ int$
$int\ m == Abs\text{-}Integ(intrel\ ``\ \{(m,0)\})$

defs (overloaded)

Zero-int-def: $0 == int\ 0$
One-int-def: $1 == int\ 1$

minus-int-def:
$- z == Abs\text{-}Integ\ (\bigcup (x,y) \in Rep\text{-}Integ\ z.\ intrel``\{(y,x)\})$

add-int-def:
$z + w ==$
$Abs\text{-}Integ\ (\bigcup (x,y) \in Rep\text{-}Integ\ z.\ \bigcup (u,v) \in Rep\text{-}Integ\ w.$
$intrel``\{(x+u,\ y+v)\})$

diff-int-def: $z - (w::int) == z + (-w)$

mult-int-def:
$z * w ==$
$Abs\text{-}Integ\ (\bigcup (x,y) \in Rep\text{-}Integ\ z.\ \bigcup (u,v) \in Rep\text{-}Integ\ w.$
$intrel``\{(x*u + y*v,\ x*v + y*u)\})$

le-int-def:
$z \le (w::int) ==$
$\exists x\ y\ u\ v.\ x+v \le u+y\ \&\ (x,y) \in Rep\text{-}Integ\ z\ \&\ (u,v) \in Rep\text{-}Integ\ w$

less-int-def: $(z < (w::int)) == (z \le w\ \&\ z \ne w)$

24.1 Construction of the Integers

24.1.1 Preliminary Lemmas about the Equivalence Relation

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* = *eq-equiv-class-iff* [*OF equiv-intrel UNIV-I UNIV-I*]

declare *equiv-intrel-iff* [*simp*]

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*] *Abs-Integ-inverse* [*simp*]

Case analysis on the representation of an integer as an equivalence class of pairs of naturals.

lemma *eq-Abs-Integ* [*case-names Abs-Integ*, *cases type*: *int*]:
  (!!*x y*. *z* = *Abs-Integ*(*intrel*``{(*x*,*y*)}) ==> *P*) ==> *P*
⟨*proof*⟩

24.1.2 *int*: Embedding the Naturals into the Integers

lemma *inj-int*: *inj int*
⟨*proof*⟩

lemma *int-int-eq* [*iff*]: (*int m* = *int n*) = (*m* = *n*)
⟨*proof*⟩

24.1.3 Integer Unary Negation

lemma *minus*: − *Abs-Integ*(*intrel*``{(*x*,*y*)}) = *Abs-Integ*(*intrel* `` {(*y*,*x*)})
⟨*proof*⟩

lemma *zminus-zminus*: − (− *z*) = (*z*::*int*)
⟨*proof*⟩

lemma *zminus-0*: − *0* = (*0*::*int*)
⟨*proof*⟩

24.2 Integer Addition

lemma *add*:
  Abs-Integ (*intrel*``{(*x*,*y*)}) + *Abs-Integ* (*intrel*``{(*u*,*v*)}) =
   Abs-Integ (*intrel*``{(*x*+*u*, *y*+*v*)})
⟨*proof*⟩

lemma *zminus-zadd-distrib*: − (*z* + *w*) = (− *z*) + (− *w*::*int*)
⟨*proof*⟩

lemma *zadd-commute*: (*z*::*int*) + *w* = *w* + *z*
⟨*proof*⟩

lemma *zadd-assoc*: ((*z1*::*int*) + *z2*) + *z3* = *z1* + (*z2* + *z3*)
⟨*proof*⟩

lemma *zadd-left-commute*: *x* + (*y* + *z*) = *y* + ((*x* + *z*) ::*int*)
 ⟨*proof*⟩

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

lemmas *zmult-ac* = *OrderedGroup.mult-ac*

lemma *zadd-int*: $(int\ m) + (int\ n) = int\ (m + n)$
⟨*proof*⟩

lemma *zadd-int-left*: $(int\ m) + (int\ n + z) = int\ (m + n) + z$
⟨*proof*⟩

lemma *int-Suc*: $int\ (Suc\ m) = 1 + (int\ m)$
⟨*proof*⟩

lemma *zadd-0*: $(0::int) + z = z$
⟨*proof*⟩

lemma *zadd-0-right*: $z + (0::int) = z$
⟨*proof*⟩

lemma *zadd-zminus-inverse2*: $(-\ z) + z = (0::int)$
⟨*proof*⟩

24.3 Integer Multiplication

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*:
$Abs\text{-}Integ((intrel``\{(x,y)\})) * Abs\text{-}Integ((intrel``\{(u,v)\})) =$
$Abs\text{-}Integ(intrel\ ``\ \{(x*u + y*v,\ x*v + y*u)\})$
⟨*proof*⟩

lemma *zmult-zminus*: $(-\ z) * w = -\ (z * (w::int))$
⟨*proof*⟩

lemma *zmult-commute*: $(z::int) * w = w * z$
⟨*proof*⟩

lemma *zmult-assoc*: $((z1::int) * z2) * z3 = z1 * (z2 * z3)$
⟨*proof*⟩

lemma *zadd-zmult-distrib*: $((z1::int) + z2) * w = (z1 * w) + (z2 * w)$
⟨*proof*⟩

lemma *zadd-zmult-distrib2*: $(w::int) * (z1 + z2) = (w * z1) + (w * z2)$
⟨*proof*⟩

lemma *zdiff-zmult-distrib*: $((z1::int) - z2) * w = (z1 * w) - (z2 * w)$

⟨*proof*⟩

lemma *zdiff-zmult-distrib2*: $(w::int) * (z1 - z2) = (w * z1) - (w * z2)$
⟨*proof*⟩

lemmas *int-distrib* =
zadd-zmult-distrib zadd-zmult-distrib2
zdiff-zmult-distrib zdiff-zmult-distrib2

lemma *int-mult*: $int\ (m * n) = (int\ m) * (int\ n)$
⟨*proof*⟩

Compatibility binding

lemmas *zmult-int* = *int-mult* [*symmetric*]

lemma *zmult-1*: $(1::int) * z = z$
⟨*proof*⟩

lemma *zmult-1-right*: $z * (1::int) = z$
⟨*proof*⟩

The integers form a *comm-ring-1*

instance *int* :: *comm-ring-1*
⟨*proof*⟩

24.4 The ≤ Ordering

lemma *le*:
$(Abs\text{-}Integ(intrel``\{(x,y)\}) \le Abs\text{-}Integ(intrel``\{(u,v)\})) = (x+v \le u+y)$
⟨*proof*⟩

lemma *zle-refl*: $w \le (w::int)$
⟨*proof*⟩

lemma *zle-trans*: $[|\ i \le j;\ j \le k\ |] ==> i \le (k::int)$
⟨*proof*⟩

lemma *zle-anti-sym*: $[|\ z \le w;\ w \le z\ |] ==> z = (w::int)$
⟨*proof*⟩

lemma *zless-le*: $((w::int) < z) = (w \le z\ \&\ w \ne z)$
⟨*proof*⟩

instance *int* :: *order*
⟨*proof*⟩

lemma *zle-linear*: $(z::int) \le w\ |\ w \le z$

⟨*proof*⟩

instance *int* :: *linorder*
⟨*proof*⟩

lemmas *zless-linear* = *linorder-less-linear* [**where** $'a = int$]
lemmas *linorder-neqE-int* = *linorder-neqE*[**where** $'a = int$]

lemma *int-eq-0-conv* [*simp*]: $(int\ n = 0) = (n = 0)$
⟨*proof*⟩

lemma *zless-int* [*simp*]: $(int\ m < int\ n) = (m<n)$
⟨*proof*⟩

lemma *int-less-0-conv* [*simp*]: $\sim\ (int\ k < 0)$
⟨*proof*⟩

lemma *zero-less-int-conv* [*simp*]: $(0 < int\ n) = (0 < n)$
⟨*proof*⟩

lemma *int-0-less-1*: $0 < (1{::}int)$
⟨*proof*⟩

lemma *int-0-neq-1* [*simp*]: $0 \neq (1{::}int)$
⟨*proof*⟩

lemma *zle-int* [*simp*]: $(int\ m \leq int\ n) = (m{\leq}n)$
⟨*proof*⟩

lemma *zero-zle-int* [*simp*]: $(0 \leq int\ n)$
⟨*proof*⟩

lemma *int-le-0-conv* [*simp*]: $(int\ n \leq 0) = (n = 0)$
⟨*proof*⟩

lemma *int-0* [*simp*]: $int\ 0 = (0{::}int)$
⟨*proof*⟩

lemma *int-1* [*simp*]: $int\ 1 = 1$
⟨*proof*⟩

lemma *int-Suc0-eq-1*: $int\ (Suc\ 0) = 1$
⟨*proof*⟩

24.5 Monotonicity results

lemma *zadd-left-mono*: $i \leq j ==> k + i \leq k + (j{::}int)$

⟨*proof*⟩

lemma *zadd-strict-right-mono*: $i < j ==> i + k < j + (k::int)$
⟨*proof*⟩

lemma *zadd-zless-mono*: $[| w'<w; z'\leq z |] ==> w' + z' < w + (z::int)$
⟨*proof*⟩

24.6 Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on k¿0

lemma *zmult-zless-mono2-lemma*:
$i<j ==> 0<k ==> int\ k * i < int\ k * j$
⟨*proof*⟩

lemma *zero-le-imp-eq-int*: $0 \leq k ==> \exists n.\ k = int\ n$
⟨*proof*⟩

lemma *zmult-zless-mono2*: $[| i<j;\ (0::int) < k |] ==> k*i < k*j$
⟨*proof*⟩

defs (overloaded)
zabs-def: $abs(i::int) == if\ i < 0\ then\ -i\ else\ i$

The integers form an ordered *comm-ring-1*

instance *int* :: *ordered-idom*
⟨*proof*⟩

lemma *zless-imp-add1-zle*: $w<z ==> w + (1::int) \leq z$
⟨*proof*⟩

24.7 Magnitide of an Integer, as a Natural Number: *nat*

constdefs
$nat :: int => nat$
$nat\ z == contents\ (\bigcup (x,y) \in Rep\text{-}Integ\ z.\ \{ x-y \})$

lemma *nat*: $nat\ (Abs\text{-}Integ\ (intrel``\{(x,y)\})) = x-y$
⟨*proof*⟩

lemma *nat-int* [*simp*]: $nat(int\ n) = n$
⟨*proof*⟩

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

lemma *int-nat-eq* [*simp*]: $int\ (nat\ z) = (if\ 0 \leq z\ then\ z\ else\ 0)$

⟨*proof*⟩

corollary *nat-0-le*: $0 \leq z$ ==> *int* (*nat* z) = z
⟨*proof*⟩

lemma *nat-le-0* [*simp*]: $z \leq 0$ ==> *nat* z = 0
⟨*proof*⟩

lemma *nat-le-eq-zle*: $0 < w$ | $0 \leq z$ ==> (*nat* $w \leq$ *nat* z) = ($w \leq z$)
⟨*proof*⟩

An alternative condition is $(0::'a) \leq w$

corollary *nat-mono-iff*: $0 < z$ ==> (*nat* $w <$ *nat* z) = ($w < z$)
⟨*proof*⟩

corollary *nat-less-eq-zless*: $0 \leq w$ ==> (*nat* $w <$ *nat* z) = ($w<z$)
⟨*proof*⟩

lemma *zless-nat-conj*: (*nat* $w <$ *nat* z) = ($0 < z$ & $w < z$)
⟨*proof*⟩

lemma *nonneg-eq-int*: [| $0 \leq z$; !!m. z = *int* m ==> P |] ==> P
⟨*proof*⟩

lemma *nat-eq-iff*: (*nat* w = m) = (*if* $0 \leq w$ *then* w = *int* m *else* m=0)
⟨*proof*⟩

corollary *nat-eq-iff2*: (m = *nat* w) = (*if* $0 \leq w$ *then* w = *int* m *else* m=0)
⟨*proof*⟩

lemma *nat-less-iff*: $0 \leq w$ ==> (*nat* $w < m$) = ($w <$ *int* m)
⟨*proof*⟩

lemma *int-eq-iff*: (*int* m = z) = (m = *nat* z & $0 \leq z$)
⟨*proof*⟩

lemma *zero-less-nat-eq* [*simp*]: ($0 <$ *nat* z) = ($0 < z$)
⟨*proof*⟩

lemma *nat-add-distrib*:
    [| $(0::int) \leq z$; $0 \leq z'$ |] ==> *nat* $(z+z')$ = *nat* z + *nat* z'
⟨*proof*⟩

lemma *nat-diff-distrib*:
    [| $(0::int) \leq z'$; $z' \leq z$ |] ==> *nat* $(z-z')$ = *nat* z − *nat* z'
⟨*proof*⟩

lemma *nat-zminus-int* [*simp*]: *nat* (− (*int* n)) = 0

⟨*proof*⟩

lemma *zless-nat-eq-int-zless*: $(m < nat\ z) = (int\ m < z)$
⟨*proof*⟩

24.8 Lemmas about the Function *int* and Orderings

lemma *negative-zless-0*: $- (int\ (Suc\ n)) < 0$
⟨*proof*⟩

lemma *negative-zless* [*iff*]: $- (int\ (Suc\ n)) < int\ m$
⟨*proof*⟩

lemma *negative-zle-0*: $- int\ n \leq 0$
⟨*proof*⟩

lemma *negative-zle* [*iff*]: $- int\ n \leq int\ m$
⟨*proof*⟩

lemma *not-zle-0-negative* [*simp*]: $\tilde{}\ (0 \leq - (int\ (Suc\ n)))$
⟨*proof*⟩

lemma *int-zle-neg*: $(int\ n \leq - int\ m) = (n = 0\ \&\ m = 0)$
⟨*proof*⟩

lemma *not-int-zless-negative* [*simp*]: $\tilde{}\ (int\ n < - int\ m)$
⟨*proof*⟩

lemma *negative-eq-positive* [*simp*]: $(- int\ n = int\ m) = (n = 0\ \&\ m = 0)$
⟨*proof*⟩

lemma *zle-iff-zadd*: $(w \leq z) = (\exists\, n.\ z = w + int\ n)$
⟨*proof*⟩

lemma *abs-int-eq* [*simp*]: $abs\ (int\ m) = int\ m$
⟨*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*]:
$P(abs(a::'a::ordered\text{-}idom)) = ((0 \leq a \dashrightarrow P\ a)\ \&\ (a < 0 \dashrightarrow P(-a)))$
⟨*proof*⟩

24.9 The Constants *neg* and *iszero*

constdefs

neg :: *'a::ordered-idom => bool*

neg(*Z*) == *Z* < *0*

iszero :: *'a*::*comm-semiring-1-cancel* => *bool*
iszero z == *z* = (*0*)

lemma *not-neg-int* [*simp*]: ~ *neg*(*int n*)
⟨*proof*⟩

lemma *neg-zminus-int* [*simp*]: *neg*(− (*int* (*Suc n*)))
⟨*proof*⟩

lemmas *neg-eq-less-0* = *neg-def*

lemma *not-neg-eq-ge-0*: (~*neg x*) = (*0* ≤ *x*)
⟨*proof*⟩

24.10 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* ==> *int* (*nat z*) = *z*
⟨*proof*⟩

24.11 The Set of Natural Numbers

constdefs
Nats :: *'a*::*comm-semiring-1-cancel set*
Nats == *range of-nat*

syntax (*xsymbols*) *Nats* :: *'a set* (ℕ)

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ *Nats*
⟨*proof*⟩

lemma *Nats-0* [*simp*]: $0 \in Nats$
⟨*proof*⟩

lemma *Nats-1* [*simp*]: $1 \in Nats$
⟨*proof*⟩

lemma *Nats-add* [*simp*]: $[|a \in Nats;\ b \in Nats|] ==> a+b \in Nats$
⟨*proof*⟩

lemma *Nats-mult* [*simp*]: $[|a \in Nats;\ b \in Nats|] ==> a*b \in Nats$
⟨*proof*⟩

Agreement with the specific embedding for the integers

lemma *int-eq-of-nat*: $int = (of\text{-}nat :: nat => int)$
⟨*proof*⟩

lemma *of-nat-eq-id* [*simp*]: $of\text{-}nat = (id :: nat => nat)$
⟨*proof*⟩

24.12 Embedding of the Integers into any *comm-ring-1*: *of-int*

constdefs
  $of\text{-}int :: int => 'a::comm\text{-}ring\text{-}1$
  $of\text{-}int\ z == contents\ (\bigcup (i,j) \in Rep\text{-}Integ\ z.\ \{\ of\text{-}nat\ i\ -\ of\text{-}nat\ j\ \})$

lemma *of-int*: $of\text{-}int\ (Abs\text{-}Integ\ (intrel\ ``\ \{(i,j)\})) = of\text{-}nat\ i\ -\ of\text{-}nat\ j$
⟨*proof*⟩

lemma *of-int-0* [*simp*]: $of\text{-}int\ 0 = 0$
⟨*proof*⟩

lemma *of-int-1* [*simp*]: $of\text{-}int\ 1 = 1$
⟨*proof*⟩

lemma *of-int-add* [*simp*]: $of\text{-}int\ (w+z) = of\text{-}int\ w + of\text{-}int\ z$
⟨*proof*⟩

lemma *of-int-minus* [*simp*]: $of\text{-}int\ (-z) = -\ (of\text{-}int\ z)$
⟨*proof*⟩

lemma *of-int-diff* [*simp*]: $of\text{-}int\ (w-z) = of\text{-}int\ w - of\text{-}int\ z$
⟨*proof*⟩

lemma *of-int-mult* [*simp*]: $of\text{-}int\ (w*z) = of\text{-}int\ w * of\text{-}int\ z$
⟨*proof*⟩

lemma *of-int-le-iff* [*simp*]:

$(of\text{-}int\ w \le (of\text{-}int\ z::'a::ordered\text{-}idom)) = (w \le z)$
⟨*proof*⟩

Special cases where either operand is zero

lemmas *of-int-0-le-iff* = *of-int-le-iff* [*of 0*, *simplified*]
lemmas *of-int-le-0-iff* = *of-int-le-iff* [*of - 0*, *simplified*]
declare *of-int-0-le-iff* [*simp*]
declare *of-int-le-0-iff* [*simp*]

lemma *of-int-less-iff* [*simp*]:
$(of\text{-}int\ w < (of\text{-}int\ z::'a::ordered\text{-}idom)) = (w < z)$
⟨*proof*⟩

Special cases where either operand is zero

lemmas *of-int-0-less-iff* = *of-int-less-iff* [*of 0*, *simplified*]
lemmas *of-int-less-0-iff* = *of-int-less-iff* [*of - 0*, *simplified*]
declare *of-int-0-less-iff* [*simp*]
declare *of-int-less-0-iff* [*simp*]

The ordering on the *comm-ring-1* is necessary. See *of-nat-eq-iff* above.

lemma *of-int-eq-iff* [*simp*]:
$(of\text{-}int\ w = (of\text{-}int\ z::'a::ordered\text{-}idom)) = (w = z)$
⟨*proof*⟩

Special cases where either operand is zero

lemmas *of-int-0-eq-iff* = *of-int-eq-iff* [*of 0*, *simplified*]
lemmas *of-int-eq-0-iff* = *of-int-eq-iff* [*of - 0*, *simplified*]
declare *of-int-0-eq-iff* [*simp*]
declare *of-int-eq-0-iff* [*simp*]

lemma *of-int-eq-id* [*simp*]: *of-int* = (*id* :: *int* => *int*)
⟨*proof*⟩

24.13 The Set of Integers

constdefs
Ints :: *'a::comm-ring-1 set*
Ints == *range of-int*

syntax (*xsymbols*)
Ints :: *'a set* (ℤ)

lemma *Ints-0* [*simp*]: $0 \in Ints$
⟨*proof*⟩

lemma *Ints-1* [*simp*]: $1 \in Ints$
⟨*proof*⟩

lemma *Ints-add* [*simp*]: $[|a \in Ints;\ b \in Ints|] ==> a+b \in Ints$
$\langle proof \rangle$

lemma *Ints-minus* [*simp*]: $a \in Ints ==> -a \in Ints$
$\langle proof \rangle$

lemma *Ints-diff* [*simp*]: $[|a \in Ints;\ b \in Ints|] ==> a-b \in Ints$
$\langle proof \rangle$

lemma *Ints-mult* [*simp*]: $[|a \in Ints;\ b \in Ints|] ==> a*b \in Ints$
$\langle proof \rangle$

Collapse nested embeddings

lemma *of-int-of-nat-eq* [*simp*]: *of-int* (*of-nat n*) = *of-nat n*
$\langle proof \rangle$

lemma *of-int-int-eq* [*simp*]: *of-int* (*int n*) = *of-nat n*
$\langle proof \rangle$

lemma *Ints-cases* [*case-names of-int*, *cases set*: *Ints*]:
  $q \in \mathbb{Z} ==> (!!z.\ q = of\text{-}int\ z ==> C) ==> C$
$\langle proof \rangle$

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
  $q \in \mathbb{Z} ==> (!!z.\ P\ (of\text{-}int\ z)) ==> P\ q$
  $\langle proof \rangle$

declare *int-Suc* [*simp*]

24.14 More Properties of *setsum* and *setprod*

By Jeremy Avigad

lemma *of-nat-setsum*: $of\text{-}nat\ (setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}nat(f\ x))$
  $\langle proof \rangle$

lemma *of-int-setsum*: $of\text{-}int\ (setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}int(f\ x))$
  $\langle proof \rangle$

lemma *int-setsum*: $int\ (setsum\ f\ A) = (\sum x{\in}A.\ int(f\ x))$
  $\langle proof \rangle$

lemma *of-nat-setprod*: $of\text{-}nat\ (setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}nat(f\ x))$
  $\langle proof \rangle$

lemma *of-int-setprod*: $of\text{-}int\ (setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}int(f\ x))$
  $\langle proof \rangle$

lemma *int-setprod*: $int\ (setprod\ f\ A) = (\prod x{\in}A.\ int(f\ x))$
⟨*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*⟩

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

lemma *zless-iff-Suc-zadd*:
$(w < z) = (\exists n.\ z = w + int(Suc\ n))$
⟨*proof*⟩

lemma *negD*: $x{<}0 \Longrightarrow \exists n.\ x = -\ (int\ (Suc\ n))$
⟨*proof*⟩

theorem *int-cases* [*cases type*: *int*, *case-names nonneg neg*]:
$[|!!\ n.\ z = int\ n \Longrightarrow P;\ \ !!\ n.\ z = \ -\ (int\ (Suc\ n)) \Longrightarrow P\ |] \Longrightarrow P$
⟨*proof*⟩

theorem *int-induct* [*induct type*: *int*, *case-names nonneg neg*]:
$[|!!\ n.\ P\ (int\ n);\ \ !!n.\ P\ (-\ (int\ (Suc\ n)))\ |] \Longrightarrow P\ z$
⟨*proof*⟩

Contributed by Brian Huffman

theorem *int-diff-cases* [*case-names diff*]:
assumes *prem*: $!!m\ n.\ z = int\ m - int\ n \Longrightarrow P$ **shows** P
⟨*proof*⟩

lemma *of-nat-nat*: $0 \leq z \Longrightarrow of\text{-}nat\ (nat\ z) = of\text{-}int\ z$
⟨*proof*⟩

24.15 Configuration of the code generator

types-code
int (*int*)
attach (*term-of*) ⟪

val term-of-int = HOLogic.mk-int o IntInf.fromInt;
⟫
attach (*test*) ⟪
fun gen-int i = one-of [~1, 1] ∗ random-range 0 i;
⟫

constdefs
int-aux :: int ⇒ nat ⇒ int
int-aux i n == (i + int n)
nat-aux :: nat ⇒ int ⇒ nat
nat-aux n i == (n + nat i)

lemma [*code*]:
int-aux i 0 = i
int-aux i (Suc n) = int-aux (i + 1) n — tail recursive
int n = int-aux 0 n
⟨*proof*⟩

lemma [*code*]: *nat-aux n i = (if i <= 0 then n else nat-aux (Suc n) (i − 1))*
— tail recursive
⟨*proof*⟩
lemma [*code*]: *nat i = nat-aux 0 i*
⟨*proof*⟩

consts-code
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 ((- <=/ -))
neg ((- < *0*))

⟨*ML*⟩

quickcheck-params [*default-type = int*]

⟨*ML*⟩

end

25 Numeral: Arithmetic on Binary Integers

theory *Numeral*
imports *IntDef Datatype*

uses *../Tools/numeral-syntax.ML*
begin

The file *numeral-syntax.ML* hides the constructors Pls and Min. Only qualified access Numeral.Pls and Numeral.Min is allowed. The datatype constructors bit.B0 and bit.B1 are similarly hidden. We do not hide Bit because we need the BIT infix syntax.

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Integ/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*.

typedef (*Bin*)
bin = UNIV::int set
⟨*proof*⟩

This datatype avoids the use of type *bool*, which would make all of the rewrite rules higher-order. If the use of datatype causes problems, this two-element type can easily be formalized using typedef.

datatype *bit = B0 | B1*

constdefs
Pls :: bin
Pls == Abs-Bin 0

Min :: bin
Min == Abs-Bin (− 1)

Bit :: [bin,bit] => bin (**infixl** *BIT 90*)
— That is, 2w+b
w BIT b == Abs-Bin ((case b of B0 => 0 | B1 => 1) + Rep-Bin w + Rep-Bin w)

axclass
number < type — for numeric types: nat, int, real, . . .

consts
number-of :: bin => 'a::number

syntax
-Numeral :: num-const => 'a (-)
Numeral0 :: 'a
Numeral1 :: 'a

translations

Numeral0 == *number-of Numeral.Pls*
Numeral1 == *number-of* (*Numeral.Pls BIT bit.B1*)

⟨*ML*⟩

syntax (*xsymbols*)
-square :: *'a => 'a* ((-^{2}) [*1000*] *999*)
syntax (*HTML* **output**)
-square :: *'a => 'a* ((-^{2}) [*1000*] *999*)
syntax (**output**)
-square :: *'a => 'a* ((- ^/ *2*) [*81*] *80*)
translations
x^2 == *x^2*
x^2 <= *x^(2::nat)*

lemma *Let-number-of* [*simp*]: *Let* (*number-of v*) *f* == *f* (*number-of v*)
— Unfold all *let*s involving constants
⟨*proof*⟩

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

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

constdefs
bin-succ :: *bin=>bin*
bin-succ w == *Abs-Bin*(*Rep-Bin w* + *1*)

bin-pred :: *bin=>bin*
bin-pred w == *Abs-Bin*(*Rep-Bin w* − *1*)

bin-minus :: *bin=>bin*
bin-minus w == *Abs-Bin*(− (*Rep-Bin w*))

bin-add :: [*bin*,*bin*]=>*bin*
bin-add v w == *Abs-Bin*(*Rep-Bin v* + *Rep-Bin w*)

bin-mult :: [*bin*,*bin*]=>*bin*
bin-mult v w == *Abs-Bin*(*Rep-Bin v* ∗ *Rep-Bin w*)

lemmas *Bin-simps* =
bin-succ-def bin-pred-def bin-minus-def bin-add-def bin-mult-def
Pls-def Min-def Bit-def Abs-Bin-inverse Rep-Bin-inverse Bin-def

Removal of leading zeroes

lemma *Pls-0-eq* [*simp*]: *Numeral.Pls BIT bit.B0* = *Numeral.Pls*
⟨*proof*⟩

lemma *Min-1-eq* [*simp*]: *Numeral.Min BIT bit.B1* = *Numeral.Min*
⟨*proof*⟩

25.1 The Functions *bin-succ*, *bin-pred* and *bin-minus*

lemma *bin-succ-Pls* [*simp*]: *bin-succ Numeral.Pls* = *Numeral.Pls BIT bit.B1*
⟨*proof*⟩

lemma *bin-succ-Min* [*simp*]: *bin-succ Numeral.Min* = *Numeral.Pls*
⟨*proof*⟩

lemma *bin-succ-1* [*simp*]: *bin-succ*(*w BIT bit.B1*) = (*bin-succ w*) *BIT bit.B0*
⟨*proof*⟩

lemma *bin-succ-0* [*simp*]: *bin-succ*(*w BIT bit.B0*) = *w BIT bit.B1*
⟨*proof*⟩

lemma *bin-pred-Pls* [*simp*]: *bin-pred Numeral.Pls* = *Numeral.Min*
⟨*proof*⟩

lemma *bin-pred-Min* [*simp*]: *bin-pred Numeral.Min* = *Numeral.Min BIT bit.B0*
⟨*proof*⟩

lemma *bin-pred-1* [*simp*]: *bin-pred*(*w BIT bit.B1*) = *w BIT bit.B0*
⟨*proof*⟩

lemma *bin-pred-0* [*simp*]: *bin-pred*(*w BIT bit.B0*) = (*bin-pred w*) *BIT bit.B1*
⟨*proof*⟩

lemma *bin-minus-Pls* [*simp*]: *bin-minus Numeral.Pls* = *Numeral.Pls*
⟨*proof*⟩

lemma *bin-minus-Min* [*simp*]: *bin-minus Numeral.Min* = *Numeral.Pls BIT bit.B1*
⟨*proof*⟩

lemma *bin-minus-1* [*simp*]:
bin-minus (*w BIT bit.B1*) = *bin-pred* (*bin-minus w*) *BIT bit.B1*
⟨*proof*⟩

lemma *bin-minus-0* [*simp*]: *bin-minus*(*w BIT bit.B0*) = (*bin-minus w*) *BIT bit.B0*
⟨*proof*⟩

25.2 Binary Addition and Multiplication: *bin-add* and *bin-mult*

lemma *bin-add-Pls* [*simp*]: *bin-add Numeral.Pls w* = *w*
⟨*proof*⟩

lemma *bin-add-Min* [*simp*]: *bin-add Numeral.Min w = bin-pred w*
⟨*proof*⟩

lemma *bin-add-BIT-11* [*simp*]:
bin-add (*v BIT bit.B1*) (*w BIT bit.B1*) = *bin-add v* (*bin-succ w*) *BIT bit.B0*
⟨*proof*⟩

lemma *bin-add-BIT-10* [*simp*]:
bin-add (*v BIT bit.B1*) (*w BIT bit.B0*) = (*bin-add v w*) *BIT bit.B1*
⟨*proof*⟩

lemma *bin-add-BIT-0* [*simp*]:
bin-add (*v BIT bit.B0*) (*w BIT y*) = *bin-add v w BIT y*
⟨*proof*⟩

lemma *bin-add-Pls-right* [*simp*]: *bin-add w Numeral.Pls = w*
⟨*proof*⟩

lemma *bin-add-Min-right* [*simp*]: *bin-add w Numeral.Min = bin-pred w*
⟨*proof*⟩

lemma *bin-mult-Pls* [*simp*]: *bin-mult Numeral.Pls w = Numeral.Pls*
⟨*proof*⟩

lemma *bin-mult-Min* [*simp*]: *bin-mult Numeral.Min w = bin-minus w*
⟨*proof*⟩

lemma *bin-mult-1* [*simp*]:
bin-mult (*v BIT bit.B1*) *w = bin-add* ((*bin-mult v w*) *BIT bit.B0*) *w*
⟨*proof*⟩

lemma *bin-mult-0* [*simp*]: *bin-mult* (*v BIT bit.B0*) *w* = (*bin-mult v w*) *BIT bit.B0*
⟨*proof*⟩

25.3 Converting Numerals to Rings: *number-of*

axclass *number-ring* ⊆ *number*, *comm-ring-1*
number-of-eq: *number-of w = of-int* (*Rep-Bin w*)

lemma *number-of-succ*:
number-of(*bin-succ w*) = (*1* + *number-of w* ::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-pred*:
number-of(*bin-pred w*) = (− *1* + *number-of w* ::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-minus*:

$number\text{-}of(bin\text{-}minus\ w) = (-\ (number\text{-}of\ w)::'a::number\text{-}ring)$
⟨proof⟩

lemma *number-of-add*:
$number\text{-}of(bin\text{-}add\ v\ w) = (number\text{-}of\ v + number\text{-}of\ w::'a::number\text{-}ring)$
⟨proof⟩

lemma *number-of-mult*:
$number\text{-}of(bin\text{-}mult\ v\ w) = (number\text{-}of\ v * number\text{-}of\ w::'a::number\text{-}ring)$
⟨proof⟩

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

lemma *double-number-of-BIT*:
$(1+1) * number\text{-}of\ w = (number\text{-}of\ (w\ BIT\ bit.B0)\ ::'a::number\text{-}ring)$
⟨proof⟩

Converting numerals 0 and 1 to their abstract versions

lemma *numeral-0-eq-0* [*simp*]: $Numeral0 = (0::'a::number\text{-}ring)$
⟨proof⟩

lemma *numeral-1-eq-1* [*simp*]: $Numeral1 = (1::'a::number\text{-}ring)$
⟨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*: $(-1::'a::number\text{-}ring) = -\ 1$
⟨proof⟩

lemma *mult-minus1* [*simp*]: $-1 * z = -(z::'a::number\text{-}ring)$
⟨proof⟩

lemma *mult-minus1-right* [*simp*]: $z * -1 = -(z::'a::number\text{-}ring)$
⟨proof⟩

lemma *minus-number-of-mult* [*simp*]:
$-\ (number\text{-}of\ w) * z = number\text{-}of(bin\text{-}minus\ w) * (z::'a::number\text{-}ring)$
⟨proof⟩

Subtraction

lemma *diff-number-of-eq*:
$number\text{-}of\ v - number\text{-}of\ w =$
$(number\text{-}of(bin\text{-}add\ v\ (bin\text{-}minus\ w))::'a::number\text{-}ring)$
⟨proof⟩

lemma *number-of-Pls*: $number\text{-}of\ Numeral.Pls = (0::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *number-of-Min*: $number\text{-}of\ Numeral.Min = (-\ 1::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *number-of-BIT*:
$number\text{-}of(w\ BIT\ x) = (case\ x\ of\ bit.B0 \Rightarrow 0 \mid bit.B1 \Rightarrow (1::'a::number\text{-}ring))$
+
$(number\text{-}of\ w) + (number\text{-}of\ w)$
⟨*proof*⟩

25.4 Equality of Binary Numbers

First version by Norbert Voelker

lemma *eq-number-of-eq*:
$((number\text{-}of\ x::'a::number\text{-}ring) = number\text{-}of\ y) =$
$iszero\ (number\text{-}of\ (bin\text{-}add\ x\ (bin\text{-}minus\ y)) :: 'a)$
⟨*proof*⟩

lemma *iszero-number-of-Pls*: $iszero\ ((number\text{-}of\ Numeral.Pls)::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *nonzero-number-of-Min*: $\sim\ iszero\ ((number\text{-}of\ Numeral.Min)::'a::number\text{-}ring)$
⟨*proof*⟩

25.5 Comparisons, for Ordered Rings

lemma *double-eq-0-iff*: $(a + a = 0) = (a = (0::'a::ordered\text{-}idom))$
⟨*proof*⟩

lemma *le-imp-0-less*:
assumes *le*: $0 \leq z$ **shows** $(0::int) < 1 + z$
⟨*proof*⟩

lemma *odd-nonzero*: $1 + z + z \neq (0::int)$
⟨*proof*⟩

The premise involving $\mathbb{Z}$ prevents $a = (1::'a) / (2::'a)$.

lemma *Ints-odd-nonzero*: $a \in Ints \Longrightarrow 1 + a + a \neq (0::'a::ordered\text{-}idom)$
⟨*proof*⟩

lemma *Ints-number-of*: $(number\text{-}of\ w :: 'a::number\text{-}ring) \in Ints$
⟨*proof*⟩

lemma *iszero-number-of-BIT*:

iszero (*number-of* (*w BIT x*)::'*a*) =
(*x*=*bit.B0* & *iszero* (*number-of w*::'*a*::{*ordered-idom*,*number-ring*}))
⟨*proof*⟩

lemma *iszero-number-of-0*:
iszero (*number-of* (*w BIT bit.B0*) :: '*a*::{*ordered-idom*,*number-ring*}) =
iszero (*number-of w* :: '*a*)
⟨*proof*⟩

lemma *iszero-number-of-1*:
~ *iszero* (*number-of* (*w BIT bit.B1*)::'*a*::{*ordered-idom*,*number-ring*})
⟨*proof*⟩

25.6 The Less-Than Relation

lemma *less-number-of-eq-neg*:
((*number-of x*::'*a*::{*ordered-idom*,*number-ring*}) < *number-of y*)
= *neg* (*number-of* (*bin-add x* (*bin-minus 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 Numeral.Pls* ::'*a*::{*ordered-idom*,*number-ring*})
⟨*proof*⟩

lemma *neg-number-of-Min*:
neg (*number-of Numeral.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*⟩

The premise involving $\mathbb{Z}$ prevents $a = (1::'a) / (2::'a)$.

lemma *Ints-odd-less-0*:
a ∈ *Ints* ==> (*1* + *a* + *a* < *0*) = (*a* < (*0*::'*a*::*ordered-idom*))
⟨*proof*⟩

lemma *neg-number-of-BIT*:
neg (*number-of* (*w BIT x*)::'*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*:
$((number\text{-}of\ x::'a::\{ordered\text{-}idom,number\text{-}ring\}) \leq number\text{-}of\ y)$
$= (\sim\ (neg\ (number\text{-}of\ (bin\text{-}add\ y\ (bin\text{-}minus\ x))\ ::\ 'a)))$
⟨*proof*⟩

Absolute value (*abs*)

lemma *abs-number-of*:
$abs(number\text{-}of\ x::'a::\{ordered\text{-}idom,number\text{-}ring\}) =$
$(if\ number\text{-}of\ x < (0::'a)\ then\ -number\text{-}of\ x\ else\ number\text{-}of\ x)$
⟨*proof*⟩

Re-orientation of the equation nnn=x

lemma *number-of-reorient*: $(number\text{-}of\ w = x) = (x = number\text{-}of\ w)$
⟨*proof*⟩

25.7 Simplification of arithmetic operations on integer constants.

lemmas *bin-arith-extra-simps* =
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 *bin-arith-simps* =
Numeral.bit.distinct
Pls-0-eq Min-1-eq
bin-pred-Pls bin-pred-Min bin-pred-1 bin-pred-0
bin-succ-Pls bin-succ-Min bin-succ-1 bin-succ-0
bin-add-Pls bin-add-Min bin-add-BIT-0 bin-add-BIT-10 bin-add-BIT-11
bin-minus-Pls bin-minus-Min bin-minus-1 bin-minus-0
bin-mult-Pls bin-mult-Min bin-mult-1 bin-mult-0
bin-add-Pls-right bin-add-Min-right
abs-zero abs-one bin-arith-extra-simps

Simplification of relational operations

lemmas *bin-rel-simps* =
eq-number-of-eq iszero-number-of-Pls nonzero-number-of-Min
iszero-number-of-0 iszero-number-of-1
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-BIT

le-number-of-eq

declare *bin-arith-extra-simps* [*simp*]
declare *bin-rel-simps* [*simp*]

25.8 Simplification of arithmetic when nested to the right

lemma *add-number-of-left* [*simp*]:
$number\text{-}of\ v + (number\text{-}of\ w + z) =$
$(number\text{-}of(bin\text{-}add\ v\ w) + z::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *mult-number-of-left* [*simp*]:
$number\text{-}of\ v * (number\text{-}of\ w * z) =$
$(number\text{-}of(bin\text{-}mult\ v\ w) * z::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *add-number-of-diff1*:
$number\text{-}of\ v + (number\text{-}of\ w - c) =$
$number\text{-}of(bin\text{-}add\ v\ w) - (c::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *add-number-of-diff2* [*simp*]: $number\text{-}of\ v + (c - number\text{-}of\ w) =$
$number\text{-}of\ (bin\text{-}add\ v\ (bin\text{-}minus\ w)) + (c::'a::number\text{-}ring)$
⟨*proof*⟩

end

26 IntArith: Integer arithmetic

theory *IntArith*
imports *Numeral*
uses (*int-arith1.ML*)
begin

Duplicate: can't understand why it's necessary

declare *numeral-0-eq-0* [*simp*]

26.1 Instantiating Binary Arithmetic for the Integers

instance
int :: *number* ⟨*proof*⟩

defs (overloaded)
int-number-of-def: $(number\text{-}of\ w :: int) == of\text{-}int\ (Rep\text{-}Bin\ w)$
— the type constraint is essential!

instance *int* :: *number-ring*

⟨*proof*⟩

26.2 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*⟩

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

26.3 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* =
add-special diff-special eq-special less-special le-special

⟨*ML*⟩

26.4 Lemmas About Small Numerals

lemma *of-int-m1* [*simp*]: *of-int* -1 = (-1 :: ′*a* :: *number-ring*)
⟨*proof*⟩

lemma *abs-minus-one* [*simp*]: *abs* (-1) = (*1*::′*a*::{*ordered-idom*,*number-ring*})
⟨*proof*⟩

lemma *abs-power-minus-one* [*simp*]:
abs(-1 ^ *n*) = (*1*::′*a*::{*ordered-idom*,*number-ring*,*recpower*})
⟨*proof*⟩

lemma *of-int-number-of-eq*:
of-int (*number-of v*) = (*number-of v* :: ′*a* :: *number-ring*)
⟨*proof*⟩

Lemmas for specialist use, NOT as default simprules

lemma *mult-2*: *2* ∗ *z* = (*z*+*z*::′*a*::*number-ring*)
⟨*proof*⟩

lemma *mult-2-right*: *z* ∗ *2* = (*z*+*z*::′*a*::*number-ring*)
⟨*proof*⟩

26.5 More Inequality Reasoning

lemma *zless-add1-eq*: $(w < z + (1::int)) = (w<z \mid w=z)$
⟨*proof*⟩

lemma *add1-zle-eq*: $(w + (1::int) \leq z) = (w<z)$
⟨*proof*⟩

lemma *zle-diff1-eq* [*simp*]: $(w \leq z - (1::int)) = (w<z)$
⟨*proof*⟩

lemma *zle-add1-eq-le* [*simp*]: $(w < z + (1::int)) = (w \leq z)$
⟨*proof*⟩

lemma *int-one-le-iff-zero-less*: $((1::int) \leq z) = (0 < z)$
⟨*proof*⟩

26.6 The Functions *nat* and *int*

Simplify the terms *int 0*, *int* (*Suc 0*) and $w + - z$

declare *Zero-int-def* [*symmetric*, *simp*]
declare *One-int-def* [*symmetric*, *simp*]

cooper.ML refers to this theorem

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* = *int-eq-iff* [*of - number-of v*, *standard*]
declare *int-eq-iff-number-of* [*simp*]

lemma *split-nat* [*arith-split*]:
$P(nat(i::int)) = ((\forall n.\ i = int\ n \longrightarrow P\ n)\ \&\ (i < 0 \longrightarrow P\ 0))$
(**is** $?P = (?L\ \&\ ?R)$)
⟨*proof*⟩

lemma *zdiff-int*: $n \leq m$ ==> *int* m − *int* n = *int* $(m-n)$
⟨*proof*⟩

lemma *nat-mult-distrib*: $(0::int) \leq z$ ==> *nat* $(z*z')$ = *nat* z ∗ *nat* z'
⟨*proof*⟩

lemma *nat-mult-distrib-neg*: $z \leq (0::int)$ ==> *nat*$(z*z')$ = *nat*$(-z)$ ∗ *nat*$(-z')$
⟨*proof*⟩

lemma *nat-abs-mult-distrib*: *nat* (*abs* $(w * z)$) = *nat* (*abs* w) ∗ *nat* (*abs* z)
⟨*proof*⟩

26.7 Induction principles for int

theorem *int-ge-induct*[*case-names base step*,*induct set*:*int*]:
  assumes *ge*: $k \leq (i::int)$ **and**
    base: $P(k)$ **and**
    step: $\bigwedge i.\ [\![k \leq i;\ P\ i]\!] \Longrightarrow P(i+1)$
  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: $\bigwedge i.\ [\![k < i;\ P\ i]\!] \Longrightarrow P(i+1)$
  shows $P\ i$
⟨*proof*⟩

theorem *int-le-induct*[*consumes 1*,*case-names base step*]:
  assumes *le*: $i \leq (k::int)$ **and**
    base: $P(k)$ **and**
    step: $\bigwedge i.\ [\![i \leq k;\ P\ i]\!] \Longrightarrow P(i - 1)$
  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: $\bigwedge i.\ [\![i < k;\ P\ i]\!] \Longrightarrow P(i - 1)$
  shows $P\ i$
⟨*proof*⟩

26.8 Intermediate value theorems

lemma *int-val-lemma*:
  $(\forall\, i<n::nat.\ abs(f(i+1) - f\ i) \leq 1)$ -->
   $f\ 0 \leq k$ --> $k \leq f\ n$ --> $(\exists\, i \leq n.\ f\ i = (k::int))$
⟨*proof*⟩

lemmas *nat0-intermed-int-val* = *int-val-lemma* [*rule-format* (*no-asm*)]

lemma *nat-intermed-int-val*:
$[| \forall i.\ m \le i\ \&\ i < n \dashrightarrow abs(f(i + 1{::}nat) - f\ i) \le 1;\ m < n;$
$f\ m \le k;\ k \le f\ n\ |] ==> ?\ i.\ m \le i\ \&\ i \le n\ \&\ f\ i = (k{::}int)$
⟨*proof*⟩

26.9 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*⟩

⟨*ML*⟩

end

27 SetInterval: Set intervals

theory *SetInterval*
imports *IntArith*
begin

constdefs
lessThan :: ('*a*::*ord*) => '*a set* ((*1*{..<-}))
{..<*u*} == {*x*. *x*<*u*}

atMost :: ('*a*::*ord*) => '*a set* ((*1*{..-}))
{..*u*} == {*x*. *x*<=*u*}

greaterThan :: ('*a*::*ord*) => '*a set* ((*1*{-<..}))
{*l*<..} == {*x*. *l*<*x*}

atLeast :: ('*a*::*ord*) => '*a set* ((*1*{-..}))
{*l*..} == {*x*. *l*<=*x*}

```
greaterThanLessThan :: ['a::ord, 'a] => 'a set  ((1{-<..<-}))
{l<..<u} == {l<..} Int {..<u}

atLeastLessThan :: ['a::ord, 'a] => 'a set      ((1{-..<-}))
{l..<u} == {l..} Int {..<u}

greaterThanAtMost :: ['a::ord, 'a] => 'a set    ((1{-<..-}))
{l<..u} == {l<..} Int {..u}

atLeastAtMost :: ['a::ord, 'a] => 'a set        ((1{-..-}))
{l..u} == {l..} Int {..u}
```

syntax
```
  -lessThan    :: ('a::ord) => 'a set        ((1{..-'(}))
  -greaterThan :: ('a::ord) => 'a set        ((1{')-..}))
  -greaterThanLessThan :: ['a::ord, 'a] => 'a set  ((1{')-..-'(}))
  -atLeastLessThan :: ['a::ord, 'a] => 'a set      ((1{-..-'(}))
  -greaterThanAtMost :: ['a::ord, 'a] => 'a set    ((1{')-..-}))
```
translations
```
  {..m(} => {..<m}
  {)m..} => {m<..}
  {)m..n(} => {m<..<n}
  {m..n(} => {m..<n}
  {)m..n} => {m<..n}
```

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
```

$UN\ i{<}n.\ A\quad == UN\ i{:}\{..{<}n\}.\ A$
$INT\ i{<=}n.\ A == INT\ i{:}\{..n\}.\ A$
$INT\ i{<}n.\ A\ == INT\ i{:}\{..{<}n\}.\ A$

27.1 Various equivalences

lemma *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 *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 *atLeast-iff* [*iff*]: $(i{:}\ atLeast\ k) = (k{<=}i)$
⟨*proof*⟩

lemma *Compl-atLeast* [*simp*]:
$!!k{::}\ 'a{::}linorder.\ -atLeast\ k = lessThan\ k$
⟨*proof*⟩

lemma *atMost-iff* [*iff*]: $(i{:}\ atMost\ k) = (i{<=}k)$
⟨*proof*⟩

lemma *atMost-Int-atLeast*: $!!n{::}\ 'a{::}order.\ atMost\ n\ Int\ atLeast\ n = \{n\}$
⟨*proof*⟩

27.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-subset-iff* [*iff*]:
$(atLeast\ x \subseteq atLeast\ y) = (y \le (x{::}'a{::}order))$
⟨*proof*⟩

lemma *atLeast-eq-iff* [*iff*]:
$(atLeast\ x = atLeast\ y) = (x = (y{::}'a{::}linorder))$
⟨*proof*⟩

lemma *greaterThan-subset-iff* [*iff*]:

$(greaterThan\ x \subseteq greaterThan\ y) = (y \le (x::'a::linorder))$
⟨*proof*⟩

lemma *greaterThan-eq-iff* [*iff*]:
$(greaterThan\ x = greaterThan\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *atMost-subset-iff* [*iff*]: $(atMost\ x \subseteq atMost\ y) = (x \le (y::'a::order))$
⟨*proof*⟩

lemma *atMost-eq-iff* [*iff*]: $(atMost\ x = atMost\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *lessThan-subset-iff* [*iff*]:
$(lessThan\ x \subseteq lessThan\ y) = (x \le (y::'a::linorder))$
⟨*proof*⟩

lemma *lessThan-eq-iff* [*iff*]:
$(lessThan\ x = lessThan\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

27.3 Two-sided intervals

lemma *greaterThanLessThan-iff* [*simp*]:
$(i : \{l{<}..{<}u\}) = (l < i\ \&\ i < u)$
⟨*proof*⟩

lemma *atLeastLessThan-iff* [*simp*]:
$(i : \{l..{<}u\}) = (l <= i\ \&\ i < u)$
⟨*proof*⟩

lemma *greaterThanAtMost-iff* [*simp*]:
$(i : \{l{<}..u\}) = (l < i\ \&\ i <= u)$
⟨*proof*⟩

lemma *atLeastAtMost-iff* [*simp*]:
$(i : \{l..u\}) = (l <= i\ \&\ i <= u)$
⟨*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).

27.3.1 Emptyness and singletons

lemma *atLeastAtMost-empty* [*simp*]: $n < m ==> \{m::'a::order..n\} = \{\}$
⟨*proof*⟩

lemma *atLeastLessThan-empty*[*simp*]: $n \le m ==> \{m..{<}n::'a::order\} = \{\}$

⟨*proof*⟩

lemma *greaterThanAtMost-empty*[*simp*]:$l \le k$ ==> {k<..(l::'*a*::*order*)} = {}
⟨*proof*⟩

lemma *greaterThanLessThan-empty*[*simp*]:$l \le k$ ==> {k<..(l::'*a*::*order*)} = {}
⟨*proof*⟩

lemma *atLeastAtMost-singleton* [*simp*]: {*a*::'*a*::*order*..*a*} = {*a*}
⟨*proof*⟩

27.4 Intervals of natural numbers

27.4.1 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*⟩

27.4.2 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*⟩

27.4.3 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*⟩

27.4.4 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*⟩

27.4.5 The Constant *atLeastLessThan*

But not a simprule because some concepts are better left in terms of *atLeastLessThan*

lemma *atLeast0LessThan*: {*0*::*nat*..<*n*} = {..<*n*}
⟨*proof*⟩

27.4.6 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* ≤ *Suc n* ⟹ {*m*..*Suc n*} = *insert* (*Suc n*) {*m*..*n*}
⟨*proof*⟩

27.4.7 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*⟩

27.4.8 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*⟩

27.4.9 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*⟩

27.5 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*⟩

27.5.1 Finiteness

lemma *image-atLeastZeroLessThan-int*: $0 \le u ==>$
$\{(0::int)..<u\} = int\ `\ \{..<nat\ u\}$
⟨*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*⟩

27.5.2 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*⟩

27.6 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy

27.6.1 Disjoint Unions

Singletons and open intervals

lemma *ivl-disj-un-singleton*:
$\{l::'a::linorder\}\ Un\ \{l<..\} = \{l..\}$
$\{..<u\}\ Un\ \{u::'a::linorder\} = \{..u\}$
$(l::'a::linorder) < u \Longrightarrow \{l\}\ Un\ \{l<..<u\} = \{l..<u\}$
$(l::'a::linorder) < u \Longrightarrow \{l<..<u\}\ Un\ \{u\} = \{l<..u\}$
$(l::'a::linorder) <= u \Longrightarrow \{l\}\ Un\ \{l<..u\} = \{l..u\}$
$(l::'a::linorder) <= u \Longrightarrow \{l..<u\}\ Un\ \{u\} = \{l..u\}$
⟨*proof*⟩

One- and two-sided intervals

lemma *ivl-disj-un-one*:
$(l::'a::linorder) < u \Longrightarrow \{..l\}\ Un\ \{l<..<u\} = \{..<u\}$
$(l::'a::linorder) <= u \Longrightarrow \{..<l\}\ Un\ \{l..<u\} = \{..<u\}$
$(l::'a::linorder) <= u \Longrightarrow \{..l\}\ Un\ \{l<..u\} = \{..u\}$
$(l::'a::linorder) <= u \Longrightarrow \{..<l\}\ Un\ \{l..u\} = \{..u\}$
$(l::'a::linorder) <= u \Longrightarrow \{l<..u\}\ Un\ \{u<..\} = \{l<..\}$
$(l::'a::linorder) < u \Longrightarrow \{l<..<u\}\ Un\ \{u..\} = \{l<..\}$
$(l::'a::linorder) <= u \Longrightarrow \{l..u\}\ Un\ \{u<..\} = \{l..\}$
$(l::'a::linorder) <= u \Longrightarrow \{l..<u\}\ Un\ \{u..\} = \{l..\}$
⟨*proof*⟩

Two- and two-sided intervals

lemma *ivl-disj-un-two*:
$[|\ (l::'a::linorder) < m;\ m <= u\ |] \Longrightarrow \{l<..<m\}\ Un\ \{m..<u\} = \{l<..<u\}$

$[| (l::'a::linorder) <= m;\ m < u\ |] ==> \{l<..m\}\ Un\ \{m<..<u\} = \{l<..<u\}$
$[| (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..<u\} = \{l..<u\}$
$[| (l::'a::linorder) <= m;\ m < u\ |] ==> \{l..m\}\ Un\ \{m<..<u\} = \{l..<u\}$
$[| (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..u\} = \{l<..u\}$
$[| (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l<..m\}\ Un\ \{m<..u\} = \{l<..u\}$
$[| (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..u\} = \{l..u\}$
$[| (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..m\}\ Un\ \{m<..u\} = \{l..u\}$
⟨*proof*⟩

lemmas *ivl-disj-un* = *ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two*

27.6.2 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*

27.6.3 Some Differences

lemma *ivl-diff* [*simp*]:
$i \leq n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
$\langle proof \rangle$

27.6.4 Some Subset Conditions

lemma *ivl-subset*[*simp*]:
$(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \mid m \leq i \;\&\; j \leq (n::'a::linorder))$
$\langle proof \rangle$

27.7 Summation indexed over intervals

syntax
-from-to-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM* - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM* - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM* -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM* -<=-./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-from-to-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ -≤-./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-from-to-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*3*$\sum$ -≤-./ -) [*0*,*0*,*10*] *10*)
syntax (*latex-sum* **output**)
-from-to-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
((*3*$\sum_{- = -}^{-}$ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
((*3*$\sum_{- = -}^{<-}$ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
((*3*$\sum_{- < -}$ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
((*3*$\sum_{- \leq -}$ -) [*0*,*0*,*10*] *10*)

translations
$\sum x{=}a..b.\ t$ == *setsum* (%*x*. *t*) $\{a..b\}$
$\sum x{=}a..{<}b.\ t$ == *setsum* (%*x*. *t*) $\{a..<b\}$
$\sum i{\leq}n.\ t$ == *setsum* ($\lambda i.\ t$) $\{..n\}$
$\sum i{<}n.\ t$ == *setsum* ($\lambda i.\ t$) $\{..<n\}$

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	LaTeX
$\sum x{\in}\{a..b\}.\ e$	$\sum x = a..b.\ e$	$\sum_{x = a}^{b} e$
$\sum x{\in}\{a..{<}b\}.\ e$	$\sum x = a..{<}b.\ e$	$\sum_{x = a}^{<b} e$
$\sum x{\in}\{..b\}.\ e$	$\sum x{\le}b.\ e$	$\sum_{x \le b} e$
$\sum x{\in}\{..{<}b\}.\ e$	$\sum x{<}b.\ e$	$\sum_{x < b} e$

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 $\sum x = 0..{<}n.\ e$ rather than $\sum x{<}n.\ e$: *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*⟩

27.8 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*⟩

27.9 The formula for geometric sums

lemma *geometric-sum*:
$x\ \tilde{}= 1 ==> (\sum i=0..<n.\ x\ \hat{}\ i) =$
$(x\ \hat{}\ n - 1)\ /\ (x - 1::'a::\{field,\ recpower,\ division\text{-}by\text{-}zero\})$
⟨*proof*⟩

⟨*ML*⟩

end

28 Recdef: TFL: recursive function definitions

theory *Recdef*
imports *Wellfounded-Relations Datatype*
uses
(../*TFL/casesplit.ML*)
(../*TFL/utils.ML*)
(../*TFL/usyntax.ML*)
(../*TFL/dcterm.ML*)
(../*TFL/thms.ML*)
(../*TFL/rules.ML*)
(../*TFL/thry.ML*)
(../*TFL/tfl.ML*)
(../*TFL/post.ML*)

(*Tools/recdef-package.ML*)
begin

lemma *tfl-eq-True*: $(x = True) \dashrightarrow x$
⟨*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 ==> $P = True$
⟨*proof*⟩

lemma *tfl-imp-trans*: $(A$ --> $B)$ ==> $(B$ --> $C)$ ==> $(A$ --> $C)$
⟨*proof*⟩

lemma *tfl-disj-assoc*: $(a \vee b) \vee c$ == $a \vee (b \vee c)$
⟨*proof*⟩

lemma *tfl-disjE*: $P \vee Q$ ==> P --> R ==> Q --> R ==> R
⟨*proof*⟩

lemma *tfl-exE*: $\exists x.\ P\ x$ ==> $\forall x.\ P\ x$ --> Q ==> 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 image-cong*

lemma *let-cong* [*recdef-cong*]:
$M = N$ ==> (!!$x.\ x = N$ ==> $f\ x = g\ x$) ==> $Let\ M\ f = Let\ N\ g$
⟨*proof*⟩

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-pred-nat

wf-same-fst
wf-empty

lemma *insert-None-conv-UNIV*: *insert None* (*range Some*) = *UNIV*
⟨*proof*⟩

instance *option* :: (*finite*) *finite*
⟨*proof*⟩

end

29 IntDiv: The Division Operators div and mod; the Divides Relation dvd

theory *IntDiv*
imports *SetInterval Recdef*
uses (*IntDiv-setup.ML*)
begin

declare *zless-nat-conj* [*simp*]

constdefs
quorem :: (*int*∗*int*) ∗ (*int*∗*int*) => *bool*
— definition of quotient and remainder
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
adjust b == %(*q*,*r*). *if 0* ≤ *r*−*b then* (*2*∗*q* + *1*, *r*−*b*)
else (*2*∗*q*, *r*)

algorithm for the case $a \geq 0$, $b > 0$

consts *posDivAlg* :: *int*∗*int* => *int*∗*int*
recdef *posDivAlg measure* (%(*a*,*b*). *nat*(*a* − *b* + *1*))
posDivAlg (*a*,*b*) =
(*if* (*a*<*b* | *b*≤*0*) *then* (*0*,*a*)
else adjust b (*posDivAlg*(*a*, *2*∗*b*)))

algorithm for the case $a < 0$, $b > 0$

consts *negDivAlg* :: *int*∗*int* => *int*∗*int*
recdef *negDivAlg measure* (%(*a*,*b*). *nat*(− *a* − *b*))
negDivAlg (*a*,*b*) =
(*if* (*0*≤*a*+*b* | *b*≤*0*) *then* (−*1*,*a*+*b*)

else adjust b (negDivAlg(a, 2∗b)))

algorithm for the general case $b \neq (0::'a)$

constdefs
negateSnd :: *int∗int => int∗int*
negateSnd == %(*q*,*r*). (*q*,−*r*)

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 $a < (0::'a)$.
divAlg ==
%(*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*))

instance
int :: *Divides.div* ⟨*proof*⟩

The operators are defined with reference to the algorithm, which is proved to satisfy the specification.

defs
div-def: *a div b == fst* (*divAlg* (*a*,*b*))
mod-def: *a mod b == snd* (*divAlg* (*a*,*b*))

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));
```

29.1 Uniqueness and Monotonicity of Quotients and Remainders

lemma *unique-quotient-lemma*:
$[| \; b*q' + r' \; \leq b*q + r; \;\; 0 \leq r'; \;\; r' < b; \;\; r < b \; |]$
$==> q' \leq (q{::}int)$
⟨*proof*⟩

lemma *unique-quotient-lemma-neg*:
$[| \; b*q' + r' \leq b*q + r; \;\; r \leq 0; \;\; b < r; \;\; b < r' \; |]$
$==> q \leq (q'{::}int)$
⟨*proof*⟩

lemma *unique-quotient*:
$[| \; quorem \; ((a,b), (q,r)); \;\; quorem \; ((a,b), (q',r')); \;\; b \neq 0 \; |]$
$==> q = q'$
⟨*proof*⟩

lemma *unique-remainder*:
$[| \; quorem \; ((a,b), (q,r)); \;\; quorem \; ((a,b), (q',r')); \;\; b \neq 0 \; |]$
$==> r = r'$
⟨*proof*⟩

29.2 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* [*rule-format*]:
  $0 \leq a \dashrightarrow 0 < b \dashrightarrow$ *quorem* ((*a*, *b*), *posDivAlg* (*a*, *b*))
⟨*proof*⟩

29.3 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* $0 \leq a+b$ *then* $(-1, a+b)$ *else adjust b* (*negDivAlg*(*a*, $2*b$)))
⟨*proof*⟩

lemma *negDivAlg-correct* [*rule-format*]:
  $a < 0 \dashrightarrow 0 < b \dashrightarrow$ *quorem* ((*a*, *b*), *negDivAlg* (*a*, *b*))
⟨*proof*⟩

29.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *quorem-0*: $b \neq 0 ==>$ *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*: $b \neq 0 ==>$ *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*⟩

⟨*ML*⟩

lemma *pos-mod-conj* : $(0::int) < b ==> 0 \leq a\ mod\ b\ \&\ a\ mod\ b < b$
⟨*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* : $b < (0::int) ==> a\ mod\ b \leq 0\ \&\ b < a\ mod\ b$
⟨*proof*⟩

lemmas *neg-mod-sign*[*simp*] = *neg-mod-conj* [*THEN conjunct1*, *standard*]
and *neg-mod-bound*[*simp*] = *neg-mod-conj* [*THEN conjunct2*, *standard*]

29.5 General Properties of div and mod

lemma *quorem-div-mod*: $b \neq 0 ==> quorem\ ((a, b), (a\ div\ b, a\ mod\ b))$
⟨*proof*⟩

lemma *quorem-div*: $[|\ quorem((a,b),(q,r));\ \ b \neq 0\ |] ==> a\ div\ b = q$
⟨*proof*⟩

lemma *quorem-mod*: $[|\ quorem((a,b),(q,r));\ \ b \neq 0\ |] ==> a\ mod\ b = r$
⟨*proof*⟩

lemma *div-pos-pos-trivial*: $[|\ (0::int) \leq a;\ \ a < b\ |] ==> a\ div\ b = 0$
⟨*proof*⟩

lemma *div-neg-neg-trivial*: $[|\ a \leq (0::int);\ \ b < a\ |] ==> a\ div\ b = 0$
⟨*proof*⟩

lemma *div-pos-neg-trivial*: $[|\ (0::int) < a;\ \ a+b \leq 0\ |] ==> a\ div\ b = -1$
⟨*proof*⟩

lemma *mod-pos-pos-trivial*: $[|\ (0::int) \leq a;\ \ a < b\ |] ==> a\ mod\ b = a$

⟨proof⟩

lemma *mod-neg-neg-trivial*: $[\![a \le (0::int);\ \ b < a]\!] ==> a\ mod\ b = a$
⟨proof⟩

lemma *mod-pos-neg-trivial*: $[\![(0::int) < a;\ \ a+b \le 0]\!] ==> a\ mod\ b = a+b$
⟨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⟩

29.6 Laws for div and mod with Unary Minus

lemma *zminus1-lemma*:
$quorem((a,b),(q,r))$
$==> quorem\ ((-a,b),\ (if\ r{=}0\ then\ -q\ else\ -q\ -\ 1),$
$(if\ r{=}0\ then\ 0\ else\ b{-}r))$
⟨proof⟩

lemma *zdiv-zminus1-eq-if*:
$b \ne (0::int)$
$==> (-a)\ div\ b =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ \ -\ (a\ div\ b)\ -\ 1)$
⟨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*:
$b \ne (0::int)$
$==> a\ div\ (-b) =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ \ -\ (a\ div\ b)\ -\ 1)$
⟨proof⟩

lemma *zmod-zminus2-eq-if*:
$a\ mod\ (-b::int) = (if\ a\ mod\ b = 0\ then\ 0\ else\ \ (a\ mod\ b)\ -\ b)$

⟨*proof*⟩

29.7 Division of a Number by Itself

lemma *self-quotient-aux1*: $[| (0::int) < a;\ a = r + a*q;\ r < a |] ==> 1 \le q$
⟨*proof*⟩

lemma *self-quotient-aux2*: $[| (0::int) < a;\ a = r + a*q;\ 0 \le r |] ==> q \le 1$
⟨*proof*⟩

lemma *self-quotient*: $[| quorem((a,a),(q,r));\ \ a \neq (0::int) |] ==> q = 1$
⟨*proof*⟩

lemma *self-remainder*: $[| quorem((a,a),(q,r));\ \ a \neq (0::int) |] ==> r = 0$
⟨*proof*⟩

lemma *zdiv-self* [*simp*]: $a \neq 0 ==> a\ div\ a = (1::int)$
⟨*proof*⟩

lemma *zmod-self* [*simp*]: $a\ mod\ a = (0::int)$
⟨*proof*⟩

29.8 Computation of Division and Remainder

lemma *zdiv-zero* [*simp*]: $(0::int)\ div\ b = 0$
⟨*proof*⟩

lemma *div-eq-minus1*: $(0::int) < b ==> -1\ div\ b = -1$
⟨*proof*⟩

lemma *zmod-zero* [*simp*]: $(0::int)\ mod\ b = 0$
⟨*proof*⟩

lemma *zdiv-minus1*: $(0::int) < b ==> -1\ div\ b = -1$
⟨*proof*⟩

lemma *zmod-minus1*: $(0::int) < b ==> -1\ mod\ b = b - 1$
⟨*proof*⟩

a positive, b positive

lemma *div-pos-pos*: $[| 0 < a;\ \ 0 \le b |] ==> a\ div\ b = fst\ (posDivAlg(a,b))$
⟨*proof*⟩

lemma *mod-pos-pos*: $[| 0 < a;\ \ 0 \le b |] ==> a\ mod\ b = snd\ (posDivAlg(a,b))$
⟨*proof*⟩

a negative, b positive

lemma *div-neg-pos*: $[| a < 0;\ \ 0 < b |] ==> a\ div\ b = fst\ (negDivAlg(a,b))$

⟨*proof*⟩

lemma *mod-neg-pos*: $[\mid a < 0;\ \ 0 < b \mid] ==> a \ mod \ b = snd \ (negDivAlg(a,b))$
⟨*proof*⟩

a positive, b negative

lemma *div-pos-neg*:
$[\mid 0 < a;\ \ b < 0 \mid] ==> a \ div \ b = fst \ (negateSnd(negDivAlg(-a,-b)))$
⟨*proof*⟩

lemma *mod-pos-neg*:
$[\mid 0 < a;\ \ b < 0 \mid] ==> a \ mod \ b = snd \ (negateSnd(negDivAlg(-a,-b)))$
⟨*proof*⟩

a negative, b negative

lemma *div-neg-neg*:
$[\mid a < 0;\ \ b \le 0 \mid] ==> a \ div \ b = fst \ (negateSnd(posDivAlg(-a,-b)))$
⟨*proof*⟩

lemma *mod-neg-neg*:
$[\mid a < 0;\ \ b \le 0 \mid] ==> a \ mod \ b = snd \ (negateSnd(posDivAlg(-a,-b)))$
⟨*proof*⟩

Simplify expresions in which div and mod combine numerical constants

lemmas *div-pos-pos-number-of* =
div-pos-pos [*of number-of v number-of w*, *standard*]
declare *div-pos-pos-number-of* [*simp*]

lemmas *div-neg-pos-number-of* =
div-neg-pos [*of number-of v number-of w*, *standard*]
declare *div-neg-pos-number-of* [*simp*]

lemmas *div-pos-neg-number-of* =
div-pos-neg [*of number-of v number-of w*, *standard*]
declare *div-pos-neg-number-of* [*simp*]

lemmas *div-neg-neg-number-of* =
div-neg-neg [*of number-of v number-of w*, *standard*]
declare *div-neg-neg-number-of* [*simp*]

lemmas *mod-pos-pos-number-of* =
mod-pos-pos [*of number-of v number-of w*, *standard*]
declare *mod-pos-pos-number-of* [*simp*]

lemmas *mod-neg-pos-number-of* =
mod-neg-pos [*of number-of v number-of w*, *standard*]
declare *mod-neg-pos-number-of* [*simp*]

lemmas *mod-pos-neg-number-of* =
mod-pos-neg [*of number-of v number-of w*, *standard*]
declare *mod-pos-neg-number-of* [*simp*]

lemmas *mod-neg-neg-number-of* =
mod-neg-neg [*of number-of v number-of w*, *standard*]
declare *mod-neg-neg-number-of* [*simp*]

lemmas *posDivAlg-eqn-number-of* =
posDivAlg-eqn [*of number-of v number-of w*, *standard*]
declare *posDivAlg-eqn-number-of* [*simp*]

lemmas *negDivAlg-eqn-number-of* =
negDivAlg-eqn [*of number-of v number-of w*, *standard*]
declare *negDivAlg-eqn-number-of* [*simp*]

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* =
div-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *div-pos-pos-1-number-of* [*simp*]

lemmas *div-pos-neg-1-number-of* =
div-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *div-pos-neg-1-number-of* [*simp*]

lemmas *mod-pos-pos-1-number-of* =
mod-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *mod-pos-pos-1-number-of* [*simp*]

lemmas *mod-pos-neg-1-number-of* =
mod-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *mod-pos-neg-1-number-of* [*simp*]

lemmas *posDivAlg-eqn-1-number-of* =
posDivAlg-eqn [*of* **concl**: *1 number-of w*, *standard*]
declare *posDivAlg-eqn-1-number-of* [*simp*]

lemmas *negDivAlg-eqn-1-number-of* =
negDivAlg-eqn [*of* **concl**: *1 number-of w*, *standard*]
declare *negDivAlg-eqn-1-number-of* [*simp*]

29.9 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*: $[|\ a \le a';\ \ 0 < (b{::}int)\ |] ==> a\ div\ b \le a'\ div\ b$
$\langle proof \rangle$

lemma *zdiv-mono1-neg*: $[|\ a \le a';\ \ (b{::}int) < 0\ |] ==> a'\ div\ b \le a\ div\ b$
$\langle proof \rangle$

29.10 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
$[|\ 0 \le b'{*}q' + r';\ r' < b';\ \ 0 < b'\ |] ==> 0 \le (q'{::}int)$
$\langle proof \rangle$

lemma *zdiv-mono2-lemma*:
$[|\ b{*}q + r = b'{*}q' + r';\ \ 0 \le b'{*}q' + r';$
$r' < b';\ \ 0 \le r;\ \ 0 < b';\ \ b' \le b\ |]$
$==> q \le (q'{::}int)$
$\langle proof \rangle$

lemma *zdiv-mono2*:
$[|\ (0{::}int) \le a;\ \ 0 < b';\ \ b' \le b\ |] ==> a\ div\ b \le a\ div\ b'$
$\langle proof \rangle$

lemma *q-neg-lemma*:
$[|\ b'{*}q' + r' < 0;\ \ 0 \le r';\ \ 0 < b'\ |] ==> q' \le (0{::}int)$
$\langle proof \rangle$

lemma *zdiv-mono2-neg-lemma*:
$[|\ b{*}q + r = b'{*}q' + r';\ \ b'{*}q' + r' < 0;$
$r < b;\ \ 0 \le r';\ \ 0 < b';\ \ b' \le b\ |]$
$==> q' \le (q{::}int)$
$\langle proof \rangle$

lemma *zdiv-mono2-neg*:
$[|\ a < (0{::}int);\ \ 0 < b';\ \ b' \le b\ |] ==> a\ div\ b' \le a\ div\ b$
$\langle proof \rangle$

29.11 More Algebraic Laws for div and mod

proving (a*b) div c = a * (b div c) + a * (b mod c)

lemma *zmult1-lemma*:
  [| *quorem*((*b*,*c*),(*q*,*r*)); *c* $\neq$ *0* |]
  ==> *quorem* ((*a*∗*b*, *c*), (*a*∗*q* + *a*∗*r div c*, *a*∗*r mod c*))
⟨*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*) ==> (*a*∗*b*) *div b* = *a*
⟨*proof*⟩

lemma *zdiv-zmult-self2* [*simp*]: *b* $\neq$ (*0*::*int*) ==> (*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* = *zmod-eq-0-iff* [*THEN iffD1*]
declare *zmod-eq-0D* [*dest*!]

proving (a+b) div c = a div c + b div c + ((a mod c + b mod c) div c)

lemma *zadd1-lemma*:
  [| *quorem*((*a*,*c*),(*aq*,*ar*)); *quorem*((*b*,*c*),(*bq*,*br*)); *c* $\neq$ *0* |]
  ==> *quorem* ((*a*+*b*, *c*), (*aq* + *bq* + (*ar*+*br*) *div c*, (*ar*+*br*) *mod c*))
⟨*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*]: $a \neq (0::int) ==> (a+b)\ div\ a = b\ div\ a + 1$
⟨*proof*⟩

lemma *zdiv-zadd-self2* [*simp*]: $a \neq (0::int) ==> (b+a)\ div\ a = b\ div\ a + 1$
⟨*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*⟩

29.12 Proving $a\ div\ (b * c) = a\ div\ b\ div\ c$

first, four lemmas to bound the remainder for the cases b¡0 and b¿0

lemma *zmult2-lemma-aux1*: $[|\ (0::int) < c;\ \ b < r;\ \ r \leq 0\ |] ==> b*c < b*(q\ mod\ c) + r$
⟨*proof*⟩

lemma *zmult2-lemma-aux2*:
$[|\ (0::int) < c;\ \ b < r;\ \ r \leq 0\ |] ==> b * (q\ mod\ c) + r \leq 0$
⟨*proof*⟩

lemma *zmult2-lemma-aux3*: $[|\ (0::int) < c;\ \ 0 \leq r;\ \ r < b\ |] ==> 0 \leq b * (q\ mod\ c) + r$
⟨*proof*⟩

lemma *zmult2-lemma-aux4*: $[|\ (0::int) < c;\ 0 \leq r;\ r < b\ |] ==> b * (q\ mod\ c) + r < b * c$
⟨*proof*⟩

lemma *zmult2-lemma*: $[|\ quorem\ ((a,b),\ (q,r));\ \ b \neq 0;\ \ 0 < c\ |]$
$==> quorem\ ((a,\ b*c),\ (q\ div\ c,\ b*(q\ mod\ c) + r))$
⟨*proof*⟩

lemma *zdiv-zmult2-eq*: $(0::int) < c ==> a\ div\ (b*c) = (a\ div\ b)\ div\ c$
⟨*proof*⟩

lemma *zmod-zmult2-eq*:
$(0::int) < c ==> a\ mod\ (b*c) = b*(a\ div\ b\ mod\ c) + a\ mod\ b$
⟨*proof*⟩

29.13 Cancellation of Common Factors in div

lemma *zdiv-zmult-zmult1-aux1*:
$[|\ (0::int) < b;\ \ c \neq 0\ |] ==> (c*a)\ div\ (c*b) = a\ div\ b$
⟨*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-zmult2*: $c \neq (0::int) ==> (a*c)\ div\ (b*c) = a\ div\ b$
⟨*proof*⟩

29.14 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*⟩

29.15 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*:
$P(n\ div\ k :: int) =$
$((k = 0 \ {-}{-}{>}\ P\ 0)\ \&$
$(0<k \ {-}{-}{>}\ (\forall\, i\, j.\ 0\leq j\ \&\ j<k\ \&\ n = k*i + j \ {-}{-}{>}\ P\ i))\ \&$
$(k<0 \ {-}{-}{>}\ (\forall\, i\, j.\ k<j\ \&\ j\leq 0\ \&\ n = k*i + j \ {-}{-}{>}\ P\ i)))$
⟨*proof*⟩

lemma *split-zmod*:
$P(n\ mod\ k :: int) =$
$((k = 0 \ {-}{-}{>}\ P\ n)\ \&$
$(0<k \ {-}{-}{>}\ (\forall\, i\, j.\ 0\leq j\ \&\ j<k\ \&\ n = k*i + j \ {-}{-}{>}\ P\ j))\ \&$
$(k<0 \ {-}{-}{>}\ (\forall\, i\, j.\ k<j\ \&\ j\leq 0\ \&\ n = k*i + j \ {-}{-}{>}\ P\ j)))$
⟨*proof*⟩

declare *split-zdiv* [*of - - number-of k*, *simplified*, *standard*, *arith-split*]
declare *split-zmod* [*of - - number-of k*, *simplified*, *standard*, *arith-split*]

29.16 Speeding up the Division Algorithm with Shifting

computing div by shifting

lemma *pos-zdiv-mult-2*: $(0::int) \leq a ==> (1 + 2*b)\ div\ (2*a) = b\ div\ a$
⟨*proof*⟩

lemma *neg-zdiv-mult-2*: $a \leq (0::int) ==> (1 + 2*b)\ div\ (2*a) = (b+1)\ div\ a$
⟨*proof*⟩

lemma *not-0-le-lemma*: $\sim\ 0 \leq x ==> x \leq (0::int)$
⟨*proof*⟩

lemma *zdiv-number-of-BIT*[*simp*]:
$number\text{-}of\ (v\ BIT\ b)\ div\ number\text{-}of\ (w\ BIT\ bit.B0) =$
$(if\ b{=}bit.B0\ |\ (0::int) \leq number\text{-}of\ w$
$then\ number\text{-}of\ v\ div\ (number\text{-}of\ w)$
$else\ (number\text{-}of\ v + (1::int))\ div\ (number\text{-}of\ w))$
⟨*proof*⟩

29.17 Computing mod by Shifting (proofs resemble those for div)

lemma *pos-zmod-mult-2*:
$(0::int) \leq a ==> (1 + 2*b)\ mod\ (2*a) = 1 + 2 * (b\ mod\ a)$
⟨*proof*⟩

lemma *neg-zmod-mult-2*:
$a \leq (0::int) ==> (1 + 2*b)\ mod\ (2*a) = 2 * ((b+1)\ mod\ a) - 1$

⟨*proof*⟩

lemma *zmod-number-of-BIT* [*simp*]:
  number-of (*v BIT b*) *mod number-of* (*w BIT bit.B0*) =
   (*case b of*
     bit.B0 => $2 *$ (*number-of v mod number-of w*)
    | *bit.B1* => *if* (*0::int*) $\leq$ *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*⟩

29.18 Quotients of Signs

lemma *div-neg-pos-less0*: [| $a < (0::int)$; $0 < b$ |] ==> a *div* $b < 0$
⟨*proof*⟩

lemma *div-nonneg-neg-le0*: [| $(0::int) \leq a$; $b < 0$ |] ==> a *div* $b \leq 0$
⟨*proof*⟩

lemma *pos-imp-zdiv-nonneg-iff*: $(0::int) < b$ ==> ($0 \leq a$ *div* b) = ($0 \leq a$)
⟨*proof*⟩

lemma *neg-imp-zdiv-nonneg-iff*:
  $b < (0::int)$ ==> ($0 \leq a$ *div* b) = ($a \leq (0::int)$)
⟨*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*⟩

29.19 The Divides Relation

lemma *zdvd-iff-zmod-eq-0*: (*m dvd n*) = (*n mod m* = (*0::int*))
⟨*proof*⟩

lemma *zdvd-0-right* [*iff*]: (*m::int*) *dvd 0*
⟨*proof*⟩

lemma *zdvd-0-left* [*iff*]: (*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-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-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 *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*: $(nat\ z\ dvd\ m) = (if\ 0 \le z\ then\ (z\ dvd\ int\ m)\ else\ m = 0)$
⟨*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*: $[|\ z\ dvd\ n;\ 0 < n\ |] ==> z \le (n::int)$
⟨*proof*⟩

29.20 Integer Powers

instance *int* :: *power* ⟨*proof*⟩

primrec
$p\ \hat{}\ 0 = 1$
$p\ \hat{}\ (Suc\ n) = (p::int) * (p\ \hat{}\ n)$

instance *int* :: *recpower*
⟨*proof*⟩

lemma *zpower-zmod*: $((x::int)\ mod\ m)\hat{}y\ mod\ m = x\hat{}y\ mod\ m$
⟨*proof*⟩

lemma *zpower-zadd-distrib*: $x\hat{}(y+z) = ((x\hat{}y)*(x\hat{}z)::int)$
⟨*proof*⟩

lemma *zpower-zpower*: $(x\hat{}y)\hat{}z = (x\hat{}(y*z)::int)$
⟨*proof*⟩

lemma *zero-less-zpower-abs-iff* [*simp*]:
$(0 < (abs\ x)\hat{}n) = (x \neq (0::int)\ |\ n=0)$

⟨*proof*⟩

lemma *zero-le-zpower-abs* [*simp*]: $(0::int) <= (abs\ x)\hat{}n$
⟨*proof*⟩

lemma *int-power*: $int\ (m\hat{}n) = (int\ m)\ \hat{}\ n$
⟨*proof*⟩

Compatibility binding

lemmas *zpower-int* = *int-power* [*symmetric*]

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*:
$[\![0 < m;\ 0 < k]\!] \Longrightarrow k\ \hat{}\ m\ div\ k = (k::int)\ \hat{}\ (m - Suc\ 0)$
⟨*proof*⟩

⟨*ML*⟩

end

30 NatBin: Binary arithmetic for the natural numbers

theory *NatBin*
imports *IntDiv*
begin

Arithmetic for naturals is reduced to that for the non-negative integers.

instance *nat* :: *number* ⟨*proof*⟩

defs (**overloaded**)
nat-number-of-def:
$(number\text{-}of::bin => nat)\ v == nat\ ((number\text{-}of :: bin => int)\ v)$

30.1 Function *nat*: Coercion from Type *int* to *nat*

declare *nat-0* [*simp*] *nat-1* [*simp*]

lemma *nat-number-of* [*simp*]: $nat\ (number\text{-}of\ w) = number\text{-}of\ w$
⟨*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*: $[\![0 < x;\ 1 < k]\!] \Longrightarrow x\ div\ k < (x::int)$
⟨*proof*⟩

30.2 Function *int*: Coercion from Type *nat* to *int*

lemma *int-nat-number-of* [*simp*]:
$int\ (number\text{-}of\ v :: nat) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 0$
$else\ (number\text{-}of\ v :: int))$
⟨*proof*⟩

30.2.1 Successor

lemma *Suc-nat-eq-nat-zadd1*: $(0::int) <= z ==> Suc\ (nat\ z) = nat\ (1 + z)$
⟨*proof*⟩

lemma *Suc-nat-number-of-add*:
$Suc\ (number\text{-}of\ v + n) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 1+n\ else\ number\text{-}of\ (bin\text{-}succ\ v) + n)$
⟨*proof*⟩

lemma *Suc-nat-number-of* [*simp*]:
$Suc\ (number\text{-}of\ v) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 1\ else\ number\text{-}of\ (bin\text{-}succ\ v))$
⟨*proof*⟩

30.2.2 Addition

lemma *add-nat-number-of* [*simp*]:
$(\textit{number-of } v :: \textit{nat}) + \textit{number-of } v' =$
$(\textit{if neg } (\textit{number-of } v :: \textit{int}) \textit{ then number-of } v'$
$\textit{else if neg } (\textit{number-of } v' :: \textit{int}) \textit{ then number-of } v$
$\textit{else number-of } (\textit{bin-add } v\ v'))$
⟨*proof*⟩

30.2.3 Subtraction

lemma *diff-nat-eq-if*:
$\textit{nat } z - \textit{nat } z' =$
$(\textit{if neg } z' \textit{ then nat } z$
$\textit{else let } d = z-z' \textit{ in}$
$\textit{if neg } d \textit{ then } 0 \textit{ else nat } d)$
⟨*proof*⟩

lemma *diff-nat-number-of* [*simp*]:
$(\textit{number-of } v :: \textit{nat}) - \textit{number-of } v' =$
$(\textit{if neg } (\textit{number-of } v' :: \textit{int}) \textit{ then number-of } v$
$\textit{else let } d = \textit{number-of } (\textit{bin-add } v\ (\textit{bin-minus } v')) \textit{ in}$
$\textit{if neg } d \textit{ then } 0 \textit{ else nat } d)$
⟨*proof*⟩

30.2.4 Multiplication

lemma *mult-nat-number-of* [*simp*]:
$(\textit{number-of } v :: \textit{nat}) * \textit{number-of } v' =$
$(\textit{if neg } (\textit{number-of } v :: \textit{int}) \textit{ then } 0 \textit{ else number-of } (\textit{bin-mult } v\ v'))$
⟨*proof*⟩

30.2.5 Quotient

lemma *div-nat-number-of* [*simp*]:
$(\textit{number-of } v :: \textit{nat}) \textit{ div } \textit{number-of } v' =$
$(\textit{if neg } (\textit{number-of } v :: \textit{int}) \textit{ then } 0$
$\textit{else nat } (\textit{number-of } v \textit{ div number-of } v'))$
⟨*proof*⟩

lemma *one-div-nat-number-of* [*simp*]:
$(\textit{Suc } 0) \textit{ div } \textit{number-of } v' = (\textit{nat } (1 \textit{ div number-of } v'))$
⟨*proof*⟩

30.2.6 Remainder

lemma *mod-nat-number-of* [*simp*]:
$(\textit{number-of } v :: \textit{nat}) \textit{ mod } \textit{number-of } v' =$
$(\textit{if neg } (\textit{number-of } v :: \textit{int}) \textit{ then } 0$
$\textit{else if neg } (\textit{number-of } v' :: \textit{int}) \textit{ then number-of } v$

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*⟩

⟨*ML*⟩

30.3 Comparisons

30.3.1 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 (bin-add v (bin-minus v′)) :: int))
⟨*proof*⟩

30.3.2 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 (bin-minus v′) :: int)*
else neg (number-of (bin-add v (bin-minus v′)) :: int))
⟨*proof*⟩

lemmas *numerals* = *nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2*

30.4 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*: $(a::'a::\{comm\text{-}semiring\text{-}1\text{-}cancel,recpower\})^2 = a * a$
⟨*proof*⟩

lemma *zero-power2* [*simp*]: $(0::'a::\{comm\text{-}semiring\text{-}1\text{-}cancel,recpower\})^2 = 0$
⟨*proof*⟩

lemma *one-power2* [*simp*]: $(1::'a::\{comm\text{-}semiring\text{-}1\text{-}cancel,recpower\})^2 = 1$
⟨*proof*⟩

lemma *power3-eq-cube*: $(x::'a::recpower)\ \hat{}\ 3 = x * x * x$
⟨*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*: $0 \le (a^2::'a::\{ordered\text{-}idom,recpower\})$
⟨*proof*⟩

lemma *zero-less-power2*:
$(0 < a^2) = (a \neq (0::'a::\{ordered\text{-}idom,recpower\}))$
⟨*proof*⟩

lemma *power2-less-0*:
fixes $a :: 'a::\{ordered\text{-}idom,recpower\}$
shows $\sim (a^2 < 0)$
⟨*proof*⟩

lemma *zero-eq-power2*:
$(a^2 = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\}))$
⟨*proof*⟩

lemma *abs-power2*:
$abs(a^2) = (a^2::'a::\{ordered\text{-}idom,recpower\})$
⟨*proof*⟩

lemma *power2-abs*:
$(abs\ a)^2 = (a^2::'a::\{ordered\text{-}idom,recpower\})$
⟨*proof*⟩

lemma *power2-minus*:
$(-\ a)^2 = (a^2::'a::\{comm\text{-}ring\text{-}1,recpower\})$
⟨*proof*⟩

lemma *power-minus1-even*: $(-\ 1)\ \hat{}\ (2*n) = (1::'a::\{comm\text{-}ring\text{-}1,recpower\})$
⟨*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'*:
$0 \le (a::'a::\{ordered\text{-}idom,recpower\})\ \hat{}\ (2*n)$
⟨*proof*⟩

lemma *odd-power-less-zero*:
$(a::'a::\{ordered\text{-}idom,recpower\}) < 0 ==> a\ \hat{}\ Suc(2*n) < 0$
⟨*proof*⟩

lemma *odd-0-le-power-imp-0-le*:
$0 \le a\ \ \hat{}\ Suc(2*n) ==> 0 \le (a::'a::\{ordered\text{-}idom,recpower\})$
⟨*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

30.4.1 Nat

lemma *Suc-pred'*: $0 < n ==> n = Suc(n - 1)$
⟨*proof*⟩

lemmas *expand-Suc* = *Suc-pred'* [*of number-of v*, *standard*]

30.4.2 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*: $(m{::}nat) * n = (\mathit{if}\ m{=}0\ \mathit{then}\ 0\ \mathit{else}\ n + ((m - 1) * n))$
⟨proof⟩

lemma *power-eq-if*: $(p\ \hat{}\ m :: nat) = (\mathit{if}\ m{=}0\ \mathit{then}\ 1\ \mathit{else}\ p * (p\ \hat{}\ (m - 1)))$
⟨proof⟩

30.5 Comparisons involving (0::nat)

Simplification already does $n < (0{::}'a)$, $n \leq (0{::}'a)$ and $(0{::}'a) \leq n$.

lemma *eq-number-of-0* [*simp*]:
  (*number-of v* = (*0*::*nat*)) =
   (*if neg* (*number-of v* :: *int*) *then True else iszero* (*number-of v* :: *int*))
⟨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* (*bin-minus v*) :: *int*)
⟨proof⟩

lemma *neg-imp-number-of-eq-0*: *neg* (*number-of v* :: *int*) ==> *number-of v* = (*0*::*nat*)
⟨proof⟩

30.6 Comparisons involving Suc

lemma *eq-number-of-Suc* [*simp*]:
  (*number-of v* = *Suc n*) =
    (*let pv* = *number-of* (*bin-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* (*bin-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* (*bin-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* (*bin-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* (*bin-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* (*bin-pred v*) *in*
     if neg pv then False else n <= *nat pv*)
⟨*proof*⟩

declare *zadd-int* [*symmetric*, *simp*]

lemma *lemma1*: (*m*+*m* = *n*+*n*) = (*m* = (*n*::*int*))
⟨*proof*⟩

lemma *lemma2*: *m*+*m* ~= (*1*::*int*) + (*n* + *n*)
⟨*proof*⟩

lemma *eq-number-of-BIT-BIT*:
  ((*number-of* (*v BIT x*) ::*int*) = *number-of* (*w BIT y*)) =
   (*x*=*y* & (((*number-of v*) ::*int*) = *number-of w*))
⟨*proof*⟩

lemma *eq-number-of-BIT-Pls*:
  ((*number-of* (*v BIT x*) ::*int*) = *Numeral0*) =
   (*x*=*bit.B0* & (((*number-of v*) ::*int*) = *Numeral0*))
⟨*proof*⟩

lemma *eq-number-of-BIT-Min*:
  ((*number-of* (*v BIT x*) ::*int*) = *number-of Numeral.Min*) =
   (*x*=*bit.B1* & (((*number-of v*) ::*int*) = *number-of Numeral.Min*))
⟨*proof*⟩

lemma *eq-number-of-Pls-Min*: (*Numeral0* ::*int*) ~= *number-of Numeral.Min*
⟨*proof*⟩

30.7 Literal arithmetic involving powers

lemma *nat-power-eq*: $(0::int) <= z ==> nat\ (z\hat{}n) = nat\ z\ \hat{}\ n$
⟨*proof*⟩

lemma *power-nat-number-of*:
$(number\text{-}of\ v :: nat)\ \hat{}\ n =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 0\hat{}n\ else\ nat\ ((number\text{-}of\ v :: int)\ \hat{}\ n))$
⟨*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 the integers

lemma *zpower-number-of-even*:
$(z::int)\ \hat{}\ number\text{-}of\ (w\ BIT\ bit.B0) =$
$(let\ w = z\ \hat{}\ (number\text{-}of\ w)\ in\ \ w*w)$
⟨*proof*⟩

lemma *zpower-number-of-odd*:
$(z::int)\ \hat{}\ number\text{-}of\ (w\ BIT\ bit.B1) =$
$(if\ (0::int) <= number\text{-}of\ w$
$then\ (let\ w = z\ \hat{}\ (number\text{-}of\ w)\ in\ \ z*w*w)$
$else\ 1)$
⟨*proof*⟩

lemmas *zpower-number-of-even-number-of* =
zpower-number-of-even [*of number-of v*, *standard*]
declare *zpower-number-of-even-number-of* [*simp*]

lemmas *zpower-number-of-odd-number-of* =
zpower-number-of-odd [*of number-of v*, *standard*]
declare *zpower-number-of-odd-number-of* [*simp*]

⟨*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-of Numeral.Min* = (*0*::*nat*)
⟨*proof*⟩

lemma *nat-number-of-BIT-1*:
number-of (*w BIT bit.B1*) =
(*if neg* (*number-of w* :: *int*) *then 0*
else let n = *number-of w in Suc* (*n* + *n*))
⟨*proof*⟩

lemma *nat-number-of-BIT-0*:
number-of (*w BIT bit.B0*) = (*let n*::*nat* = *number-of w in n* + *n*)
⟨*proof*⟩

lemmas *nat-number* =
nat-number-of-Pls nat-number-of-Min
nat-number-of-BIT-1 nat-number-of-BIT-0

lemma *Let-Suc* [*simp*]: *Let* (*Suc n*) *f* == *f* (*Suc n*)
⟨*proof*⟩

lemma *power-m1-even*: (-1) ^ $(2{*}n) = (1{::}'a{::}\{number\text{-}ring,recpower\})$
⟨*proof*⟩

lemma *power-m1-odd*: (-1) ^ $Suc(2{*}n) = (-1{::}'a{::}\{number\text{-}ring,recpower\})$
⟨*proof*⟩

30.8 Literal arithmetic and *of-nat*

lemma *of-nat-double*:
$0 \leq x$ ==> *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*⟩

30.9 Lemmas for the Combination and Cancellation Simprocs

lemma *nat-number-of-add-left*:
$number\text{-}of\ v + (number\text{-}of\ v' + (k{::}nat)) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ number\text{-}of\ v' + k$
$else\ if\ neg\ (number\text{-}of\ v' :: int)\ then\ number\text{-}of\ v + k$
$else\ number\text{-}of\ (bin\text{-}add\ v\ v') + k)$
⟨*proof*⟩

lemma *nat-number-of-mult-left*:
$number\text{-}of\ v * (number\text{-}of\ v' * (k{::}nat)) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 0$
$else\ number\text{-}of\ (bin\text{-}mult\ v\ v') * k)$
⟨*proof*⟩

30.9.1 For *combine-numerals*

lemma *left-add-mult-distrib*: $i*u + (j*u + k) = (i+j)*u + (k{::}nat)$
⟨*proof*⟩

30.9.2 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*⟩

30.9.3 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*⟩

30.9.4 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*: $(k*m = k*n) = (k = (0::nat) \mid m=n)$
⟨*proof*⟩

lemma *nat-mult-div-cancel-disj*:
  $(k*m)\ div\ (k*n) = (if\ k = (0::nat)\ then\ 0\ else\ m\ div\ n)$
⟨*proof*⟩

⟨*ML*⟩

end

31 NatSimprocs: Simprocs for the Naturals

theory *NatSimprocs*
imports *NatBin*
uses *int-factor-simprocs.ML nat-simprocs.ML*
begin

⟨*ML*⟩

31.1 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-bin-pred-iff-0*:
  neg (number-of (bin-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 (bin-minus v)::int) ==>
   Suc m − (number-of v) = m − (number-of (bin-pred v))
⟨*proof*⟩

lemma *diff-Suc-eq-diff-pred*: *m − Suc n = (m − 1) − n*
⟨*proof*⟩

31.2 For *nat-case* and *nat-rec*

lemma *nat-case-number-of* [*simp*]:
  nat-case a f (number-of v) =
    (let pv = number-of (bin-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 (bin-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 (bin-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 (bin-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*⟩

31.3 Various Other Lemmas

31.3.1 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*: $(n::nat) < 2 ==> n = 0 \mid n = Suc\ 0$
⟨*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*⟩

31.3.2 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*]

31.4 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*]

lemmas *divide-less-0-iff-number-of* =
divide-less-0-iff [*of number-of w*, *standard*]
declare *divide-less-0-iff-number-of* [*simp*]

lemmas *zero-le-divide-iff-number-of* =
zero-le-divide-iff [*of number-of w*, *standard*]
declare *zero-le-divide-iff-number-of* [*simp*]

lemmas *divide-le-0-iff-number-of* =
divide-le-0-iff [*of number-of w*, *standard*]
declare *divide-le-0-iff-number-of* [*simp*]

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*]

lemmas *le-minus-iff-number-of* =
le-minus-iff [*of number-of v*, *standard*]
declare *le-minus-iff-number-of* [*simp*]

lemmas *equation-minus-iff-number-of* =
equation-minus-iff [*of number-of v*, *standard*]
declare *equation-minus-iff-number-of* [*simp*]

lemmas *minus-less-iff-number-of* =
minus-less-iff [*of - number-of v*, *standard*]
declare *minus-less-iff-number-of* [*simp*]

lemmas *minus-le-iff-number-of* =
minus-le-iff [*of - number-of v*, *standard*]
declare *minus-le-iff-number-of* [*simp*]

lemmas *minus-equation-iff-number-of* =
minus-equation-iff [*of - number-of v*, *standard*]
declare *minus-equation-iff-number-of* [*simp*]

These simplify inequalities where one side is the constant 1.

lemmas *less-minus-iff-1* = *less-minus-iff* [*of 1*, *simplified*]
declare *less-minus-iff-1* [*simp*]

lemmas *le-minus-iff-1* = *le-minus-iff* [*of 1*, *simplified*]
declare *le-minus-iff-1* [*simp*]

lemmas *equation-minus-iff-1* = *equation-minus-iff* [*of 1*, *simplified*]
declare *equation-minus-iff-1* [*simp*]

lemmas *minus-less-iff-1* = *minus-less-iff* [*of - 1*, *simplified*]

declare *minus-less-iff-1* [*simp*]

lemmas *minus-le-iff-1* = *minus-le-iff* [*of - 1*, *simplified*]
declare *minus-le-iff-1* [*simp*]

lemmas *minus-equation-iff-1* = *minus-equation-iff* [*of - 1*, *simplified*]
declare *minus-equation-iff-1* [*simp*]

Cancellation of constant factors in comparisons ($<$ and $\leq$)

lemmas *mult-less-cancel-left-number-of* =
mult-less-cancel-left [*of number-of v*, *standard*]
declare *mult-less-cancel-left-number-of* [*simp*]

lemmas *mult-less-cancel-right-number-of* =
mult-less-cancel-right [*of - number-of v*, *standard*]
declare *mult-less-cancel-right-number-of* [*simp*]

lemmas *mult-le-cancel-left-number-of* =
mult-le-cancel-left [*of number-of v*, *standard*]
declare *mult-le-cancel-left-number-of* [*simp*]

lemmas *mult-le-cancel-right-number-of* =
mult-le-cancel-right [*of - number-of v*, *standard*]
declare *mult-le-cancel-right-number-of* [*simp*]

Multiplying out constant divisors in comparisons ($<$, $\leq$ and $=$)

lemmas *le-divide-eq-number-of* = *le-divide-eq* [*of - - number-of w*, *standard*]
declare *le-divide-eq-number-of* [*simp*]

lemmas *divide-le-eq-number-of* = *divide-le-eq* [*of - number-of w*, *standard*]
declare *divide-le-eq-number-of* [*simp*]

lemmas *less-divide-eq-number-of* = *less-divide-eq* [*of - - number-of w*, *standard*]
declare *less-divide-eq-number-of* [*simp*]

lemmas *divide-less-eq-number-of* = *divide-less-eq* [*of - number-of w*, *standard*]
declare *divide-less-eq-number-of* [*simp*]

lemmas *eq-divide-eq-number-of* = *eq-divide-eq* [*of - - number-of w*, *standard*]
declare *eq-divide-eq-number-of* [*simp*]

lemmas *divide-eq-eq-number-of* = *divide-eq-eq* [*of - number-of w*, *standard*]
declare *divide-eq-eq-number-of* [*simp*]

31.5 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

31.5.1 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*, *simp*]

lemma *nat-dvd-not-less*:
[| $0 < m$; $m < n$ |] ==> ¬ *n dvd* (*m::nat*)
⟨*proof*⟩

⟨*ML*⟩

end

32 Presburger: Presburger Arithmetic: Cooper's Algorithm

theory *Presburger*
imports *NatSimprocs SetInterval*
uses (*cooper-dec.ML*) (*cooper-proof.ML*) (*qelim.ML*)
(*reflected-presburger.ML*) (*reflected-cooper.ML*) (*presburger.ML*)
begin

Theorem for unitifying the coeffitients of x in an existential formula

theorem *unity-coeff-ex*: $(\exists\, x::int.\ P\ (l * x)) = (\exists\, x.\ l\ dvd\ (1*x+0) \wedge P\ x)$
⟨*proof*⟩

lemma *uminus-dvd-conv*: $(d\ dvd\ (t::int)) = (-d\ dvd\ t)$
⟨*proof*⟩

lemma *uminus-dvd-conv'*: $(d\ dvd\ (t::int)) = (d\ dvd\ -t)$
⟨*proof*⟩

Theorems for the combination of proofs of the equality of P and $P\text{-}m$ for integers x less than some integer z.

theorem *eq-minf-conjI*: $\exists\, z1::int.\ \forall\, x.\ x < z1 \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2::int.\ \forall\, x.\ x < z2 \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z::int.\ \forall\, x.\ x < z \longrightarrow ((A1\ x \wedge B1\ x) = (A2\ x \wedge B2\ x))$
⟨*proof*⟩

theorem *eq-minf-disjI*: $\exists\, z1::int.\ \forall\, x.\ x < z1 \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2::int.\ \forall\, x.\ x < z2 \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z::int.\ \forall\, x.\ x < z \longrightarrow ((A1\ x \vee B1\ x) = (A2\ x \vee B2\ x))$

⟨*proof*⟩

Theorems for the combination of proofs of the equality of P and $P\text{-}m$ for integers x greather than some integer z.

theorem *eq-pinf-conjI*: $\exists\, z1::int.\ \forall\, x.\ z1 < x \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2::int.\ \forall\, x.\ z2 < x \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z::int.\ \forall\, x.\ z < x \longrightarrow ((A1\ x \wedge B1\ x) = (A2\ x \wedge B2\ x))$
⟨*proof*⟩

theorem *eq-pinf-disjI*: $\exists\, z1::int.\ \forall\, x.\ z1 < x \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2::int.\ \forall\, x.\ z2 < x \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z::int.\ \forall\, x.\ z < x \longrightarrow ((A1\ x \vee B1\ x) = (A2\ x \vee B2\ x))$
⟨*proof*⟩

Theorems for the combination of proofs of the modulo D property for P *plusinfinity*

FIXME: This is THE SAME theorem as for the *minusinf* version, but with $+k$.. instead of $-k$.. In the future replace these both with only one.

theorem *modd-pinf-conjI*: $\forall\, (x::int)\ k.\ A\ x = A\ (x+k*d) \Longrightarrow$
$\forall\, (x::int)\ k.\ B\ x = B\ (x+k*d) \Longrightarrow$
$\forall\, (x::int)\ (k::int).\ (A\ x \wedge B\ x) = (A\ (x+k*d) \wedge B\ (x+k*d))$
⟨*proof*⟩

theorem *modd-pinf-disjI*: $\forall\, (x::int)\ k.\ A\ x = A\ (x+k*d) \Longrightarrow$
$\forall\, (x::int)\ k.\ B\ x = B\ (x+k*d) \Longrightarrow$

$\forall (x::int)\ (k::int).\ (A\ x \vee B\ x) = (A\ (x+k*d) \vee B\ (x+k*d))$
⟨*proof*⟩

This is one of the cases where the simplifed formula is prooved to habe some property (in relation to *P-m*) but we need to prove the property for the original formula (*P-m*)

FIXME: This is exaclty the same thm as for *minusinf*.

lemma *pinf-simp-eq*: $ALL\ x.\ P(x) = Q(x)$ ==> $(EX\ (x::int).\ P(x))$ −−> $(EX\ (x::int).\ F(x))$ ==> $(EX\ (x::int).\ Q(x))$ −−> $(EX\ (x::int).\ F(x))$
⟨*proof*⟩

Theorems for the combination of proofs of the modulo D property for P *minusinfinity*

theorem *modd-minf-conjI*: $\forall (x::int)\ k.\ A\ x = A\ (x-k*d) \Longrightarrow$
$\forall (x::int)\ k.\ B\ x = B\ (x-k*d) \Longrightarrow$
$\forall (x::int)\ (k::int).\ (A\ x \wedge B\ x) = (A\ (x-k*d) \wedge B\ (x-k*d))$
⟨*proof*⟩

theorem *modd-minf-disjI*: $\forall (x::int)\ k.\ A\ x = A\ (x-k*d) \Longrightarrow$
$\forall (x::int)\ k.\ B\ x = B\ (x-k*d) \Longrightarrow$
$\forall (x::int)\ (k::int).\ (A\ x \vee B\ x) = (A\ (x-k*d) \vee B\ (x-k*d))$
⟨*proof*⟩

This is one of the cases where the simplifed formula is prooved to have some property (in relation to *P-m*) but we need to prove the property for the original formula (*P-m*).

lemma *minf-simp-eq*: $ALL\ x.\ P(x) = Q(x)$ ==> $(EX\ (x::int).\ P(x))$ −−> $(EX\ (x::int).\ F(x))$ ==> $(EX\ (x::int).\ Q(x))$ −−> $(EX\ (x::int).\ F(x))$
⟨*proof*⟩

Theorem needed for proving at runtime divide properties using the arithmetic tactic (which knows only about modulo = 0).

lemma *zdvd-iff-zmod-eq-0*: $(m\ dvd\ n) = (n\ mod\ m = (0::int))$
⟨*proof*⟩

Theorems used for the combination of proof for the backwards direction of Cooper's Theorem. They rely exclusively on Predicate calculus.

lemma *not-ast-p-disjI*: $(ALL\ x.\ Q(x::int) \wedge {}^\sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j))$ −−> $P1(x)$ −−> $P1(x+d))$
==>
$(ALL\ x.\ Q(x::int) \wedge {}^\sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j))$ −−> $P2(x)$ −−> $P2(x+d))$
==>
$(ALL\ x.\ Q(x::int) \wedge {}^\sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j))$ −−>$(P1(x) \vee P2(x))$ −−> $(P1(x+d) \vee P2(x+d)))$
⟨*proof*⟩

lemma *not-ast-p-conjI*: $(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j))\ {-}{-}{>}\ P1(x)\ {-}{-}{>}\ P1(x + d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a - j))\ {-}{-}{>}\ P2(x)\ {-}{-}{>}\ P2(x + d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a - j))\ {-}{-}{>}(P1(x) \wedge P2(x))\ {-}{-}{>}\ (P1(x + d) \wedge P2(x + d)))$
⟨*proof*⟩

lemma *not-ast-p-Q-elim*:
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a - j))\ {-}{-}{>}P(x)\ {-}{-}{>}\ P(x + d))$
$==> (\ P = Q\ )$
$==> (ALL\ x.\ \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ P(a - j))\ {-}{-}{>}P(x)\ {-}{-}{>}\ P(x + d))$
⟨*proof*⟩

Theorems used for the combination of proof for the backwards direction of Cooper's Theorem. They rely exclusively on Predicate calculus.

lemma *not-bst-p-disjI*: $(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j))\ {-}{-}{>}\ P1(x)\ {-}{-}{>}\ P1(x - d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j))\ {-}{-}{>}\ P2(x)\ {-}{-}{>}\ P2(x - d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j))\ {-}{-}{>}(P1(x) \vee P2(x))\ {-}{-}{>}\ (P1(x - d) \vee P2(x{-}d)))$
⟨*proof*⟩

lemma *not-bst-p-conjI*: $(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j))\ {-}{-}{>}\ P1(x)\ {-}{-}{>}\ P1(x - d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j))\ {-}{-}{>}\ P2(x)\ {-}{-}{>}\ P2(x - d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j))\ {-}{-}{>}(P1(x) \wedge P2(x))\ {-}{-}{>}\ (P1(x - d) \wedge P2(x{-}d)))$
⟨*proof*⟩

lemma *not-bst-p-Q-elim*:
$(ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j))\ {-}{-}{>}P(x)\ {-}{-}{>}\ P(x - d))$
$==> (\ P = Q\ )$

==> (*ALL x*. ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*b*::*int*) : *B*. *P*(*b*+*j*)) -->*P*(*x*) --> *P*(*x* - *d*))
⟨*proof*⟩

This is the first direction of Cooper's Theorem.

lemma *cooper-thm*: (*R* --> (*EX x*::*int*. *P x*)) ==> (*Q* -->(*EX x*::*int*. *P x*)) ==> ((*R*|*Q*) --> (*EX x*::*int*. *P x*))
⟨*proof*⟩

The full Cooper's Theorem in its equivalence Form. Given the premises it is trivial too, it relies exclusively on prediacte calculus.

lemma *cooper-eq-thm*: (*R* --> (*EX x*::*int*. *P x*)) ==> (*Q* -->(*EX x*::*int*. *P x*)) ==> ((~*Q*)
--> (*EX x*::*int*. *P x*) --> *R*) ==> (*EX x*::*int*. *P x*) = *R*|*Q*
⟨*proof*⟩

Some of the atomic theorems generated each time the atom does not depend on x, they are trivial.

lemma *fm-eq-minf*: *EX z*::*int*. *ALL x*. *x* < *z* --> (*P* = *P*)
⟨*proof*⟩

lemma *fm-modd-minf*: *ALL* (*x*::*int*). *ALL* (*k*::*int*). (*P* = *P*)
⟨*proof*⟩

lemma *not-bst-p-fm*: *ALL* (*x*::*int*). *Q*(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*b*::*int*) : *B*. *Q*(*b*+*j*)) --> *fm* --> *fm*
⟨*proof*⟩

lemma *fm-eq-pinf*: *EX z*::*int*. *ALL x*. *z* < *x* --> (*P* = *P*)
⟨*proof*⟩

The next two thms are the same as the *minusinf* version.

lemma *fm-modd-pinf*: *ALL* (*x*::*int*). *ALL* (*k*::*int*). (*P* = *P*)
⟨*proof*⟩

lemma *not-ast-p-fm*: *ALL* (*x*::*int*). *Q*(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*a*::*int*) : *A*. *Q*(*a* - *j*)) --> *fm* --> *fm*
⟨*proof*⟩

Theorems to be deleted from simpset when proving simplified formulaes.

lemma *P-eqtrue*: (*P*=*True*) = *P*
⟨*proof*⟩

lemma *P-eqfalse*: (*P*=*False*) = (~*P*)
⟨*proof*⟩

Theorems for the generation of the bachwards direction of Cooper's Theorem.

These are the 6 interesting atomic cases which have to be proved relying on the properties of B-set and the arithmetic and contradiction proofs.

lemma *not-bst-p-lt*: $0 < (d{::}int) ==>$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (b{::}int) : B.\ Q(b+j)) --> (\ 0 < -x + a) --> (0 < -(x - d) + a\ )$
⟨*proof*⟩

lemma *not-bst-p-gt*: $[\![\ (g{::}int) \in B;\ g = -a\ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (b{::}int) : B.\ Q(b+j)) --> (0 < (x) + a) --> (\ 0 < (x - d) + a)$
⟨*proof*⟩

lemma *not-bst-p-eq*: $[\![\ 0 < d;\ (g{::}int) \in B;\ g = -a - 1\ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (b{::}int) : B.\ Q(b+j)) --> (0 = x + a) --> (0 = (x - d) + a\ )$
⟨*proof*⟩

lemma *not-bst-p-ne*: $[\![\ 0 < d;\ (g{::}int) \in B;\ g = -a\ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (b{::}int) : B.\ Q(b+j)) --> \sim(0 = x + a) --> \sim(0 = (x - d) + a)$
⟨*proof*⟩

lemma *not-bst-p-dvd*: $(d1{::}int)\ dvd\ d ==>$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (b{::}int) : B.\ Q(b+j)) --> d1\ dvd\ (x + a) --> d1\ dvd\ ((x - d) + a\ )$
⟨*proof*⟩

lemma *not-bst-p-ndvd*: $(d1{::}int)\ dvd\ d ==>$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (b{::}int) : B.\ Q(b+j)) --> \sim(d1\ dvd\ (x + a)) --> \sim(d1\ dvd\ ((x - d) + a\ ))$
⟨*proof*⟩

Theorems for the generation of the bachwards direction of Cooper's Theorem.

These are the 6 interesting atomic cases which have to be proved relying on the properties of A-set ant the arithmetic and contradiction proofs.

lemma *not-ast-p-gt*: $0 < (d{::}int) ==>$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (a{::}int) : A.\ Q(a - j)) --> (\ 0 < x + t) --> (0 < (x + d) + t\ )$
⟨*proof*⟩

lemma *not-ast-p-lt*: $[\![0 < d\ ;(t{::}int) \in A\ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x{::}int) \wedge \sim(EX\ (j{::}int) : \{1..d\}.\ EX\ (a{::}int) : A.\ Q(a - j)) --> (0$

$< -x + t) --> (0 < -(x + d) + t)$
⟨*proof*⟩

lemma *not-ast-p-eq*: $[\![\ 0 < d; (g::int) \in A; g = -t + 1 \ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a - j)) --> (0 = x + t) --> (0 = (x + d) + t\ )$
⟨*proof*⟩

lemma *not-ast-p-ne*: $[\![\ 0 < d; (g::int) \in A; g = -t \ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a - j)) --> \tilde{}(0 = x + t) --> \tilde{}(0 = (x + d) + t)$
⟨*proof*⟩

lemma *not-ast-p-dvd*: $(d1::int)\ dvd\ d ==>$
$ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a - j)) --> d1\ dvd\ (x + t) --> d1\ dvd\ ((x + d) + t\ )$
⟨*proof*⟩

lemma *not-ast-p-ndvd*: $(d1::int)\ dvd\ d ==>$
$ALL\ x.\ Q(x::int) \wedge \tilde{}(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a - j)) --> \tilde{}(d1\ dvd\ (x + t)) --> \tilde{}(d1\ dvd\ ((x + d) + t\ ))$
⟨*proof*⟩

These are the atomic cases for the proof generation for the modulo D property for P *plusinfinity*

They are fully based on arithmetics.

lemma *dvd-modd-pinf*: $((d::int)\ dvd\ d1) ==>$
$(ALL\ (x::int).\ ALL\ (k::int).\ (((d::int)\ dvd\ (x + t)) = (d\ dvd\ (x{+}k{*}d1 + t))))$
⟨*proof*⟩

lemma *not-dvd-modd-pinf*: $((d::int)\ dvd\ d1) ==>$
$(ALL\ (x::int).\ ALL\ k.\ (\tilde{}((d::int)\ dvd\ (x + t))) = (\tilde{}(d\ dvd\ (x{+}k{*}d1 + t))))$
⟨*proof*⟩

These are the atomic cases for the proof generation for the equivalence of P and P *plusinfinity* for integers x greater than some integer z.

They are fully based on arithmetics.

lemma *eq-eq-pinf*: $EX\ z::int.\ ALL\ x.\ z < x --> ((\ 0 = x + t\ ) = False\ )$
⟨*proof*⟩

lemma *neq-eq-pinf*: $EX\ z::int.\ ALL\ x.\ z < x --> ((\tilde{}(\ 0 = x + t\ )) = True\ )$
⟨*proof*⟩

lemma *le-eq-pinf*: $EX\ z::int.\ ALL\ x.\ z < x --> (\ 0 < x + t\ = True\ )$
⟨*proof*⟩

lemma *len-eq-pinf*: $EX\ z::int.\ ALL\ x.\ z < x --> (0 < -x + t\ = False\ )$

⟨*proof*⟩

lemma *dvd-eq-pinf*: $EX\ z{::}int.\ ALL\ x.\ \ z < x \ {-}{-}{>}\ ((d\ dvd\ (x + t)) = (d\ dvd\ (x + t)))$
⟨*proof*⟩

lemma *not-dvd-eq-pinf*: $EX\ z{::}int.\ ALL\ x.\ z < x\ \ {-}{-}{>}\ ((\tilde{}(d\ dvd\ (x + t))) = (\tilde{}(d\ dvd\ (x + t))))$
⟨*proof*⟩

These are the atomic cases for the proof generation for the modulo D property for P *minusinfinity*.

They are fully based on arithmetics.

lemma *dvd-modd-minf*: $((d{::}int)\ dvd\ d1) ==>$
$(ALL\ (x{::}int).\ ALL\ (k{::}int).\ (((d{::}int)\ dvd\ (x + t)) = (d\ dvd\ (x{-}k{*}d1 + t))))$
⟨*proof*⟩

lemma *not-dvd-modd-minf*: $((d{::}int)\ dvd\ d1) ==>$
$(ALL\ (x{::}int).\ ALL\ k.\ (\tilde{}((d{::}int)\ dvd\ (x + t))) = (\tilde{}(d\ dvd\ (x{-}k{*}d1 + t))))$
⟨*proof*⟩

These are the atomic cases for the proof generation for the equivalence of P and P *minusinfinity* for integers x less than some integer z.

They are fully based on arithmetics.

lemma *eq-eq-minf*: $EX\ z{::}int.\ ALL\ x.\ x < z\ {-}{-}{>}\ ((\ 0 = x + t\ ) = False\ )$
⟨*proof*⟩

lemma *neq-eq-minf*: $EX\ z{::}int.\ ALL\ x.\ x < z\ {-}{-}{>}\ ((\tilde{}(\ 0 = x + t\ )) = True\ )$
⟨*proof*⟩

lemma *le-eq-minf*: $EX\ z{::}int.\ ALL\ x.\ x < z\ {-}{-}{>}\ (\ 0 < x + t\ \ = False\ )$
⟨*proof*⟩

lemma *len-eq-minf*: $EX\ z{::}int.\ ALL\ x.\ x < z\ {-}{-}{>}\ (0 < -x + t\ \ = True\ )$
⟨*proof*⟩

lemma *dvd-eq-minf*: $EX\ z{::}int.\ ALL\ x.\ x < z\ {-}{-}{>}\ ((d\ dvd\ (x + t)) = (d\ dvd\ (x + t)))$
⟨*proof*⟩

lemma *not-dvd-eq-minf*: $EX\ z{::}int.\ ALL\ x.\ x < z\ {-}{-}{>}\ ((\tilde{}(d\ dvd\ (x + t))) = (\tilde{}(d\ dvd\ (x + t))))$
⟨*proof*⟩

This Theorem combines whithnesses about P *minusinfinity* to show one component of the equivalence proof for Cooper's Theorem.

FIXME: remove once they are part of the distribution.

theorem *int-ge-induct*[*consumes 1*,*case-names base step*]:
 assumes *ge*: $k \leq (i::int)$ **and**
  base: $P(k)$ **and**
  step: $\bigwedge i.\ [\![k \leq i;\ P\ i]\!] \Longrightarrow P(i+1)$
 shows $P\ i$
⟨*proof*⟩

theorem *int-gr-induct*[*consumes 1*,*case-names base step*]:
 assumes *gr*: $k < (i::int)$ **and**
  base: $P(k+1)$ **and**
  step: $\bigwedge i.\ [\![k < i;\ P\ i]\!] \Longrightarrow P(i+1)$
 shows $P\ i$
⟨*proof*⟩

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*⟩

lemma *minusinfinity*:
 assumes $0 < d$ **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*⟩

This Theorem combines whithnesses about *P minusinfinity* to show one component of the equivalence proof for Cooper's Theorem.

lemma *plusinfinity*:
 assumes $0 < d$ **and**
  P1eqP1: $ALL\ (x::int)\ (k::int).\ P1\ x = P1\ (x + k * d)$ **and**
  ePeqP1: $EX\ z::int.\ ALL\ x.\ z < x\ \ {-}{-}{>}\ (P\ x = P1\ x)$
 shows $(EX\ x::int.\ P1\ x)\ {-}{-}{>}\ (EX\ x::int.\ P\ x)$
⟨*proof*⟩

Theorem for periodic function on discrete sets.

lemma *minf-vee*:
 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*⟩

Theorem for periodic function on discrete sets.

lemma *pinf-vee*:
 assumes *dpos*: $0 < (d::int)$ **and** *modd*: $ALL\ (x::int)\ (k::int).\ P\ x = P\ (x+k*d)$

shows $(EX\ x{::}int.\ P\ x) = (EX\ (j{::}int) : \{1..d\}\ .\ P\ j)$
(**is** $?LHS = ?RHS$)
⟨*proof*⟩

lemma *decr-mult-lemma*:
assumes *dpos*: $(0{::}int) < d$ **and**
minus: $ALL\ x{::}int.\ P\ x \longrightarrow P(x - d)$ **and**
knneg: $0 <= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x - k*d)$
⟨*proof*⟩

lemma *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 *cpmi-eq*: $0 < D \Longrightarrow (EX\ z{::}int.\ ALL\ x.\ x < z \dashrightarrow (P\ x = P1\ x))$
$==> ALL\ x.\tilde{}(EX\ (j{::}int) : \{1..D\}.\ EX\ (b{::}int) : B.\ P(b{+}j)) \dashrightarrow P\ (x) \dashrightarrow P\ (x - D)$
$==> (ALL\ (x{::}int).\ ALL\ (k{::}int).\ ((P1\ x){=}\ (P1\ (x{-}k{*}D))))$
$==> (EX\ (x{::}int).\ P(x)) = ((EX\ (j{::}int) : \{1..D\}\ .\ (P1(j))) \mid (EX\ (j{::}int) : \{1..D\}.\ EX\ (b{::}int) : B.\ P\ (b{+}j)))$
⟨*proof*⟩

Cooper Theorem, plus infinity version.

lemma *cppi-eq*: $0 < D \Longrightarrow (EX\ z{::}int.\ ALL\ x.\ z < x \dashrightarrow (P\ x = P1\ x))$
$==> ALL\ x.\tilde{}(EX\ (j{::}int) : \{1..D\}.\ EX\ (a{::}int) : A.\ P(a - j)) \dashrightarrow P\ (x) \dashrightarrow P\ (x + D)$
$==> (ALL\ (x{::}int).\ ALL\ (k{::}int).\ ((P1\ x){=}\ (P1\ (x{+}k{*}D))))$
$==> (EX\ (x{::}int).\ P(x)) = ((EX\ (j{::}int) : \{1..D\}\ .\ (P1(j))) \mid (EX\ (j{::}int) : \{1..D\}.\ EX\ (a{::}int) : A.\ P\ (a - j)))$
⟨*proof*⟩

Theorems for the quantifier elminination Functions.

lemma *qe-ex-conj*: $(EX\ (x{::}int).\ A\ x) = R$
$==> (EX\ (x{::}int).\ P\ x) = (Q\ \&\ (EX\ x{::}int.\ A\ x))$
$==> (EX\ (x{::}int).\ P\ x) = (Q\ \&\ R)$
⟨*proof*⟩

lemma *qe-ex-nconj*: $(EX\ (x{::}int).\ P\ x) = (True\ \&\ Q)$
$==> (EX\ (x{::}int).\ P\ x) = Q$
⟨*proof*⟩

lemma *qe-conjI*: $P1 = P2 ==> Q1 = Q2 ==> (P1\ \&\ Q1) = (P2\ \&\ Q2)$
⟨*proof*⟩

lemma *qe-disjI*: $P1 = P2 ==> Q1 = Q2 ==> (P1 \mid Q1) = (P2 \mid Q2)$
⟨*proof*⟩

lemma *qe-impI*: $P1 = P2 ==> Q1 = Q2 ==> (P1 --> Q1) = (P2 --> Q2)$
⟨*proof*⟩

lemma *qe-eqI*: $P1 = P2 ==> Q1 = Q2 ==> (P1 = Q1) = (P2 = Q2)$
⟨*proof*⟩

lemma *qe-Not*: $P = Q ==> (\sim P) = (\sim Q)$
⟨*proof*⟩

lemma *qe-ALL*: $(EX\ x.\ \sim P\ x) = R ==> (ALL\ x.\ P\ x) = (\sim R)$
⟨*proof*⟩

Theorems for proving NNF

lemma *nnf-im*: $((\sim P) = P1) ==> (Q=Q1) ==> ((P --> Q) = (P1 \mid Q1))$
⟨*proof*⟩

lemma *nnf-eq*: $((P\ \&\ Q) = (P1\ \&\ Q1)) ==> (((\sim P)\ \&\ (\sim Q)) = (P2\ \&\ Q2)) ==> ((P = Q) = ((P1\ \&\ Q1)|(P2\ \&\ Q2)))$
⟨*proof*⟩

lemma *nnf-nn*: $(P = Q) ==> ((\sim\sim P) = Q)$
⟨*proof*⟩
lemma *nnf-ncj*: $((\sim P) = P1) ==> ((\sim Q) = Q1) ==> ((\sim(P\ \&\ Q)) = (P1 \mid Q1))$
⟨*proof*⟩

lemma *nnf-ndj*: $((\sim P) = P1) ==> ((\sim Q) = Q1) ==> ((\sim(P \mid Q)) = (P1\ \&\ Q1))$
⟨*proof*⟩
lemma *nnf-nim*: $(P = P1) ==> ((\sim Q) = Q1) ==> ((\sim(P --> Q)) = (P1\ \&\ Q1))$
⟨*proof*⟩
lemma *nnf-neq*: $((P\ \&\ (\sim Q)) = (P1\ \&\ Q1)) ==> (((\sim P)\ \&\ Q) = (P2\ \&\ Q2)) ==> ((\sim(P = Q)) = ((P1\ \&\ Q1)|(P2\ \&\ Q2)))$
⟨*proof*⟩
lemma *nnf-sdj*: $((A\ \&\ (\sim B)) = (A1\ \&\ B1)) ==> ((C\ \&\ (\sim D)) = (C1\ \&\ D1)) ==> (A = (\sim C)) ==> ((\sim((A\ \&\ B) \mid (C\ \&\ D))) = ((A1\ \&\ B1) \mid (C1\ \&\ D1)))$
⟨*proof*⟩

lemma *qe-exI2*: $A = B ==> (EX\ (x::int).\ A(x)) = (EX\ (x::int).\ B(x))$
⟨*proof*⟩

lemma *qe-exI*: $(!!x::int.\ A\ x = B\ x) ==> (EX\ (x::int).\ A(x)) = (EX\ (x::int).\ B(x))$

⟨proof⟩

lemma *qe-ALLI*: (!!x::int. A x = B x) ==> (ALL (x::int). A(x)) = (ALL (x::int). B(x))
⟨proof⟩

lemma *cp-expand*: (EX (x::int). P (x)) = (EX (j::int) : {1..d}. EX (b::int) : B. (P1 (j) | P(b+j)))
==>(EX (x::int). P (x)) = (EX (j::int) : {1..d}. EX (b::int) : B. (P1 (j) | P(b+j)))
⟨proof⟩

lemma *cppi-expand*: (EX (x::int). P (x)) = (EX (j::int) : {1..d}. EX (a::int) : A. (P1 (j) | P(a − j)))
==>(EX (x::int). P (x)) = (EX (j::int) : {1..d}. EX (a::int) : A. (P1 (j) | P(a − j)))
⟨proof⟩

lemma *simp-from-to*: {i..j::int} = (if j < i then {} else insert i {i+1..j})
⟨proof⟩

Theorems required for the *adjustcoeffitienteq*

lemma *ac-dvd-eq*: **assumes** *not0*: 0 ~= (k::int)
shows ((m::int) dvd (c∗n+t)) = (k∗m dvd ((k∗c)∗n+(k∗t))) (**is** ?P = ?Q)
⟨proof⟩

lemma *ac-lt-eq*: **assumes** *gr0*: 0 < (k::int)
shows ((m::int) < (c∗n+t)) = (k∗m <((k∗c)∗n+(k∗t))) (**is** ?P = ?Q)
⟨proof⟩

lemma *ac-eq-eq* : **assumes** *not0*: 0 ~= (k::int) **shows** ((m::int) = (c∗n+t)) = (k∗m =((k∗c)∗n+(k∗t))) (**is** ?P = ?Q)
⟨proof⟩

lemma *ac-pi-eq*: **assumes** *gr0*: 0 < (k::int) **shows** (~((0::int) < (c∗n + t))) = (0 < ((−k)∗c)∗n + ((−k)∗t + k))
⟨proof⟩

lemma *binminus-uminus-conv*: (a::int) − b = a + (−b)
⟨proof⟩

lemma *linearize-dvd*: (t::int) = t1 ==> (d dvd t) = (d dvd t1)
⟨proof⟩

lemma *lf-lt*: (l::int) = ll ==> (r::int) = lr ==> (l < r) =(ll < lr)
⟨proof⟩

lemma *lf-eq*: $(l{::}int) = ll$ ==> $(r{::}int) = lr$ ==> $(l = r)$ =$(ll = lr)$
⟨*proof*⟩

lemma *lf-dvd*: $(l{::}int) = ll$ ==> $(r{::}int) = lr$ ==> $(l\ dvd\ r)$ =$(ll\ dvd\ lr)$
⟨*proof*⟩

Theorems for transforming predicates on nat to predicates on *int*

theorem *all-nat*: $(\forall x{::}nat.\ P\ x) = (\forall x{::}int.\ 0 <= x \longrightarrow P\ (nat\ x))$
⟨*proof*⟩

theorem *ex-nat*: $(\exists x{::}nat.\ P\ x) = (\exists x{::}int.\ 0 <= x \wedge P\ (nat\ x))$
⟨*proof*⟩

theorem *zdiff-int-split*: $P\ (int\ (x - y)) =$
$((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
⟨*proof*⟩

theorem *zdvd-int*: $(x\ dvd\ y) = (int\ x\ dvd\ int\ y)$
⟨*proof*⟩

theorem *number-of1*: $(0{::}int) <= number\text{-}of\ n \Longrightarrow (0{::}int) <= number\text{-}of\ (n\ BIT\ b)$
⟨*proof*⟩

theorem *number-of2*: $(0{::}int) <= Numeral0$ ⟨*proof*⟩

theorem *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*⟩

⟨*ML*⟩

end

33 Relation-Power: Powers of Relations and Functions

theory *Relation-Power*

imports *Nat*
begin

instance
  set :: (*type*) *power* ⟨*proof*⟩

primrec (*relpow*)
  $R\hat{}0 = Id$
  $R\hat{}(Suc\ n) = R\ O\ (R\hat{}n)$

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

primrec (*funpow*)
  $f\hat{}0 = id$
  $f\hat{}(Suc\ n) = f\ o\ (f\hat{}n)$

WARNING: due to the limits of Isabelle's type classes, exponentiation on functions and relations has too general a domain, namely $('a \times 'b)$ *set* and $'a \Rightarrow 'b$. Explicit type constraints may therefore be necessary. For example, $range\ (f\ \hat{}\ n) = A$ and $Range\ (R\ \hat{}\ n) = B$ need constraints.

lemma *funpow-add*: $f\ \hat{}\ (m+n) = f\hat{}m\ o\ f\hat{}n$
⟨*proof*⟩

lemma *rel-pow-1*: $!!R{::}\ ('a*'a)set.\ R\hat{}1 = R$
⟨*proof*⟩
declare *rel-pow-1* [*simp*]

lemma *rel-pow-0-I*: $(x,x) : R\hat{}0$
⟨*proof*⟩

lemma *rel-pow-Suc-I*: $[|\ (x,y) : R\hat{}n;\ (y,z){:}R\ |] ==> (x,z){:}R\hat{}(Suc\ n)$
⟨*proof*⟩

lemma *rel-pow-Suc-I2* [*rule-format*]:
  $\forall z.\ (x,y) : R \dashrightarrow (y,z){:}R\hat{}n \dashrightarrow (x,z){:}R\hat{}(Suc\ n)$
⟨*proof*⟩

lemma *rel-pow-0-E*: $[|\ (x,y) : R\hat{}0;\ x=y ==> P\ |] ==> P$
⟨*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^n; [| n=0; x = z |] ==> P;
 !!y m. [| n = Suc m; (x,y) : R^m; (y,z) : R |] ==> P
 |] ==> P
⟨proof⟩

lemma *rel-pow-Suc-D2* [*rule-format*]:
 ∀ x z. (x,z):R^(Suc n) --> (∃ y. (x,y):R & (y,z):R^n)
⟨proof⟩

lemma *rel-pow-Suc-D2'*:
 ∀ x y z. (x,y) : R^n & (y,z) : R --> (∃ w. (x,w) : R & (w,z) : R^n)
⟨proof⟩

lemma *rel-pow-E2*:
 [| (x,z) : R^n; [| n=0; x = z |] ==> P;
 !!y m. [| n = Suc m; (x,y) : R; (y,z) : R^m |] ==> P
 |] ==> P
⟨proof⟩

lemma *rtrancl-imp-UN-rel-pow*: !!p. p:R^* ==> p : (UN n. R^n)
⟨proof⟩

lemma *rel-pow-imp-rtrancl*: !!p. p:R^n ==> p:R^*
⟨proof⟩

lemma *rtrancl-is-UN-rel-pow*: R^* = (UN n. R^n)
⟨proof⟩

lemma *single-valued-rel-pow* [*rule-format*]:
 !!r::('a * 'a)set. single-valued r ==> single-valued (r^n)
⟨proof⟩

⟨ML⟩

end

34 Parity: Even and Odd for ints and nats

theory *Parity*
imports *Divides IntDiv NatSimprocs*
begin

axclass *even-odd* < *type*

instance *int* :: *even-odd* ⟨proof⟩
instance *nat* :: *even-odd* ⟨proof⟩

consts
even :: *'a::even-odd => bool*

syntax
odd :: *'a::even-odd => bool*

translations
odd x == ~even x

defs (overloaded)
even-def: *even (x::int) == x mod 2 = 0*
even-nat-def: *even (x::nat) == even (int x)*

34.1 Even and odd are mutually exclusive

lemma *int-pos-lt-two-imp-zero-or-one*:
0 <= x ==> (x::int) < 2 ==> x = 0 | x = 1
⟨*proof*⟩

lemma *neq-one-mod-two* [*simp*]: *((x::int) mod 2 ~= 0) = (x mod 2 = 1)*
⟨*proof*⟩

34.2 Behavior under integer arithmetic operations

lemma *even-times-anything*: *even (x::int) ==> even (x * y)*
⟨*proof*⟩

lemma *anything-times-even*: *even (y::int) ==> even (x * y)*
⟨*proof*⟩

lemma *odd-times-odd*: *odd (x::int) ==> odd y ==> odd (x * y)*
⟨*proof*⟩

lemma *even-product*: *even((x::int) * y) = (even x | even y)*
⟨*proof*⟩

lemma *even-plus-even*: *even (x::int) ==> even y ==> even (x + y)*
⟨*proof*⟩

lemma *even-plus-odd*: *even (x::int) ==> odd y ==> odd (x + y)*
⟨*proof*⟩

lemma *odd-plus-even*: *odd (x::int) ==> even y ==> odd (x + y)*
⟨*proof*⟩

lemma *odd-plus-odd*: *odd (x::int) ==> odd y ==> even (x + y)*
⟨*proof*⟩

lemma *even-sum*: *even ((x::int) + y) = ((even x & even y) | (odd x & odd y))*

⟨*proof*⟩

lemma *even-neg*: $even\ (-(x::int)) = even\ x$
⟨*proof*⟩

lemma *even-difference*:
$even\ ((x::int) - y) = ((even\ x\ \&\ even\ y)\ |\ (odd\ x\ \&\ odd\ y))$
⟨*proof*⟩

lemma *even-pow-gt-zero* [*rule-format*]:
$even\ (x::int) ==> 0 < n --> even\ (x\hat{}n)$
⟨*proof*⟩

lemma *odd-pow*: $odd\ x ==> odd((x::int)\hat{}n)$
⟨*proof*⟩

lemma *even-power*: $even\ ((x::int)\hat{}n) = (even\ x\ \&\ 0 < n)$
⟨*proof*⟩

lemma *even-zero*: $even\ (0::int)$
⟨*proof*⟩

lemma *odd-one*: $odd\ (1::int)$
⟨*proof*⟩

lemmas *even-odd-simps* [*simp*] = *even-def*[*of number-of v*,*standard*] *even-zero* *odd-one even-product even-sum even-neg even-difference even-power*

34.3 Equivalent definitions

lemma *two-times-even-div-two*: $even\ (x::int) ==> 2 * (x\ div\ 2) = x$
⟨*proof*⟩

lemma *two-times-odd-div-two-plus-one*: $odd\ (x::int) ==>$
$2 * (x\ div\ 2) + 1 = x$
⟨*proof*⟩

lemma *even-equiv-def*: $even\ (x::int) = (EX\ y.\ x = 2 * y)$
⟨*proof*⟩

lemma *odd-equiv-def*: $odd\ (x::int) = (EX\ y.\ x = 2 * y + 1)$
⟨*proof*⟩

34.4 even and odd for nats

lemma *pos-int-even-equiv-nat-even*: $0 \leq x ==> even\ x = even\ (nat\ x)$
⟨*proof*⟩

lemma *even-nat-product*: $even((x::nat) * y) = (even\ x\ |\ even\ y)$
⟨*proof*⟩

lemma *even-nat-sum*: *even* $((x::nat) + y) =$
$((even\ x\ \&\ even\ y)\ |\ (odd\ x\ \&\ odd\ y))$
⟨*proof*⟩

lemma *even-nat-difference*:
$even\ ((x::nat) - y) = (x < y\ |\ (even\ x\ \&\ even\ y)\ |\ (odd\ x\ \&\ odd\ y))$
⟨*proof*⟩

lemma *even-nat-Suc*: $even\ (Suc\ x) = odd\ x$
⟨*proof*⟩

lemma *even-nat-power*: $even\ ((x::nat)\,\hat{}\,y) = (even\ x\ \&\ 0 < y)$
⟨*proof*⟩

lemma *even-nat-zero*: $even\ (0::nat)$
⟨*proof*⟩

lemmas *even-odd-nat-simps* [*simp*] = *even-nat-def*[*of number-of v*,*standard*]
even-nat-zero even-nat-Suc even-nat-product even-nat-sum even-nat-power

34.5 Equivalent definitions

lemma *nat-lt-two-imp-zero-or-one*: $(x::nat) < Suc\ (Suc\ 0) ==>$
$x = 0\ |\ x = Suc\ 0$
⟨*proof*⟩

lemma *even-nat-mod-two-eq-zero*: $even\ (x::nat) ==> x\ mod\ (Suc\ (Suc\ 0)) = 0$
⟨*proof*⟩

lemma *odd-nat-mod-two-eq-one*: $odd\ (x::nat) ==> x\ mod\ (Suc\ (Suc\ 0)) = Suc\ 0$
⟨*proof*⟩

lemma *even-nat-equiv-def*: $even\ (x::nat) = (x\ mod\ Suc\ (Suc\ 0) = 0)$
⟨*proof*⟩

lemma *odd-nat-equiv-def*: $odd\ (x::nat) = (x\ mod\ Suc\ (Suc\ 0) = Suc\ 0)$
⟨*proof*⟩

lemma *even-nat-div-two-times-two*: $even\ (x::nat) ==>$
$Suc\ (Suc\ 0) * (x\ div\ Suc\ (Suc\ 0)) = x$
⟨*proof*⟩

lemma *odd-nat-div-two-times-two-plus-one*: $odd\ (x::nat) ==>$
$Suc(\ Suc\ (Suc\ 0) * (x\ div\ Suc\ (Suc\ 0))) = x$
⟨*proof*⟩

lemma *even-nat-equiv-def2*: $even\ (x::nat) = (EX\ y.\ x = Suc\ (Suc\ 0) * y)$
⟨*proof*⟩

lemma *odd-nat-equiv-def2*: odd (x::nat) = (EX y. x = Suc(Suc (Suc 0) * y))
⟨proof⟩

34.6 Parity and powers

lemma *minus-one-even-odd-power*:
(even x --> (- 1::'a::{comm-ring-1,recpower}) ^x = 1) &
(odd x --> (- 1::'a) ^x = - 1)
⟨proof⟩

lemma *minus-one-even-power* [simp]:
even x ==> (- 1::'a::{comm-ring-1,recpower}) ^x = 1
⟨proof⟩

lemma *minus-one-odd-power* [simp]:
odd x ==> (- 1::'a::{comm-ring-1,recpower}) ^x = - 1
⟨proof⟩

lemma *neg-one-even-odd-power*:
(even x --> (-1::'a::{number-ring,recpower}) ^x = 1) &
(odd x --> (-1::'a) ^x = -1)
⟨proof⟩

lemma *neg-one-even-power* [simp]:
even x ==> (-1::'a::{number-ring,recpower}) ^x = 1
⟨proof⟩

lemma *neg-one-odd-power* [simp]:
odd x ==> (-1::'a::{number-ring,recpower}) ^x = -1
⟨proof⟩

lemma *neg-power-if*:
(-x::'a::{comm-ring-1,recpower}) ^ n =
(if even n then (x ^ n) else -(x ^ n))
⟨proof⟩

lemma *zero-le-even-power*: even n ==>
0 <= (x::'a::{recpower,ordered-ring-strict}) ^ n
⟨proof⟩

lemma *zero-le-odd-power*: odd n ==>
(0 <= (x::'a::{recpower,ordered-idom}) ^ n) = (0 <= x)
⟨proof⟩

lemma *zero-le-power-eq*: (0 <= (x::'a::{recpower,ordered-idom}) ^ n) =
(even n | (odd n & 0 <= x))
⟨proof⟩

lemma *zero-less-power-eq*: $(0 < (x::'a::\{recpower,ordered\text{-}idom\})\ \hat{}\ n) =$
$(n = 0 \mid (even\ n\ \&\ x\ \tilde{}= 0) \mid (odd\ n\ \&\ 0 < x))$
⟨*proof*⟩

lemma *power-less-zero-eq*: $((x::'a::\{recpower,ordered\text{-}idom\})\ \hat{}\ n < 0) =$
$(odd\ n\ \&\ x < 0)$
⟨*proof*⟩

lemma *power-le-zero-eq*: $((x::'a::\{recpower,ordered\text{-}idom\})\ \hat{}\ n <= 0) =$
$(n\ \tilde{}= 0\ \&\ ((odd\ n\ \&\ x <= 0) \mid (even\ n\ \&\ x = 0)))$
⟨*proof*⟩

lemma *power-even-abs*: $even\ n ==>$
$(abs\ (x::'a::\{recpower,ordered\text{-}idom\}))\hat{}n = x\hat{}n$
⟨*proof*⟩

lemma *zero-less-power-nat-eq*: $(0 < (x::nat)\ \hat{}\ n) = (n = 0 \mid 0 < x)$
⟨*proof*⟩

lemma *power-minus-even* [*simp*]: $even\ n ==>$
$(-\ x)\hat{}n = (x\hat{}n::'a::\{recpower,comm\text{-}ring\text{-}1\})$
⟨*proof*⟩

lemma *power-minus-odd* [*simp*]: $odd\ n ==>$
$(-\ x)\hat{}n = -\ (x\hat{}n::'a::\{recpower,comm\text{-}ring\text{-}1\})$
⟨*proof*⟩

lemmas *power-0-left-number-of* = *power-0-left* [*of number-of w*, *standard*]
declare *power-0-left-number-of* [*simp*]

lemmas *zero-le-power-eq-number-of* =
zero-le-power-eq [*of - number-of w*, *standard*]
declare *zero-le-power-eq-number-of* [*simp*]

lemmas *zero-less-power-eq-number-of* =
zero-less-power-eq [*of - number-of w*, *standard*]
declare *zero-less-power-eq-number-of* [*simp*]

lemmas *power-le-zero-eq-number-of* =
power-le-zero-eq [*of - number-of w*, *standard*]
declare *power-le-zero-eq-number-of* [*simp*]

lemmas *power-less-zero-eq-number-of* =
power-less-zero-eq [*of - number-of w*, *standard*]
declare *power-less-zero-eq-number-of* [*simp*]

lemmas *zero-less-power-nat-eq-number-of* =

zero-less-power-nat-eq [*of* - *number-of w*, *standard*]
declare *zero-less-power-nat-eq-number-of* [*simp*]

lemmas *power-eq-0-iff-number-of* = *power-eq-0-iff* [*of* - *number-of w*, *standard*]
declare *power-eq-0-iff-number-of* [*simp*]

lemmas *power-even-abs-number-of* = *power-even-abs* [*of number-of w* -, *standard*]
declare *power-even-abs-number-of* [*simp*]

34.7 An Equivalence for $0 \leq a\hat{\ }n$

lemma *even-power-le-0-imp-0*:
$a\ \hat{}\ (2*k) \leq (0::'a::\{ordered\text{-}idom,recpower\})$ ==> $a=0$
⟨*proof*⟩

lemma *zero-le-power-iff*:
$(0 \leq a\hat{\ }n) = (0 \leq (a::'a::\{ordered\text{-}idom,recpower\}) \mid even\ n)$
(**is** *?P n*)
⟨*proof*⟩

34.8 Miscellaneous

lemma *even-plus-one-div-two*: *even* (*x*::*int*) ==> (*x* + *1*) *div 2* = *x div 2*
⟨*proof*⟩

lemma *odd-plus-one-div-two*: *odd* (*x*::*int*) ==> (*x* + *1*) *div 2* = *x div 2* + *1*
⟨*proof*⟩

lemma *div-Suc*: *Suc a div c* = *a div c* + *Suc 0 div c* +
(*a mod c* + *Suc 0 mod c*) *div c*
⟨*proof*⟩

lemma *even-nat-plus-one-div-two*: *even* (*x*::*nat*) ==>
(*Suc x*) *div Suc* (*Suc 0*) = *x div Suc* (*Suc 0*)
⟨*proof*⟩

lemma *odd-nat-plus-one-div-two*: *odd* (*x*::*nat*) ==>
(*Suc x*) *div Suc* (*Suc 0*) = *Suc* (*x div Suc* (*Suc 0*))
⟨*proof*⟩

end

35 GCD: The Greatest Common Divisor

theory *GCD*
imports *Parity*
begin

See [1].

consts
  gcd :: *nat* × *nat* => *nat* — Euclid's algorithm

recdef *gcd* *measure* $((\lambda(m, n). n)$ :: *nat* × *nat* => *nat*)
  gcd (*m*, *n*) = (*if n* = *0 then m else gcd* (*n*, *m mod n*))

constdefs
  is-gcd :: *nat* => *nat* => *nat* => *bool* — *gcd* as a relation
  is-gcd p m n == *p dvd m* ∧ *p dvd n* ∧
    (∀ *d*. *d dvd m* ∧ *d dvd n* −−> *d dvd p*)

lemma *gcd-induct*:
  (!!*m*. *P m 0*) ==>
    (!!*m n*. *0* < *n* ==> *P n* (*m mod n*) ==> *P m n*)
  ==> *P* (*m*::*nat*) (*n*::*nat*)
  ⟨*proof*⟩

lemma *gcd-0* [*simp*]: *gcd* (*m*, *0*) = *m*
  ⟨*proof*⟩

lemma *gcd-non-0*: *0* < *n* ==> *gcd* (*m*, *n*) = *gcd* (*n*, *m mod n*)
  ⟨*proof*⟩

declare *gcd.simps* [*simp del*]

lemma *gcd-1* [*simp*]: *gcd* (*m*, *Suc 0*) = *1*
  ⟨*proof*⟩

gcd (*m*, *n*) divides *m* and *n*. The conjunctions don't seem provable separately.

lemma *gcd-dvd1* [*iff*]: *gcd* (*m*, *n*) *dvd m*
  and *gcd-dvd2* [*iff*]: *gcd* (*m*, *n*) *dvd n*
  ⟨*proof*⟩

Maximality: for all *m*, *n*, *k* naturals, if *k* divides *m* and *k* divides *n* then *k* divides *gcd* (*m*, *n*).

lemma *gcd-greatest*: *k dvd m* ==> *k dvd n* ==> *k dvd gcd* (*m*, *n*)
  ⟨*proof*⟩

lemma *gcd-greatest-iff* [*iff*]: (*k dvd gcd* (*m*, *n*)) = (*k dvd m* ∧ *k dvd n*)
  ⟨*proof*⟩

lemma *gcd-zero*: (*gcd* (*m*, *n*) = *0*) = (*m* = *0* ∧ *n* = *0*)

⟨*proof*⟩

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: *is-gcd (gcd (m, n)) m n*
⟨*proof*⟩

Uniqueness of GCDs.

lemma *is-gcd-unique*: *is-gcd m a b ==> is-gcd n a b ==> m = n*
⟨*proof*⟩

lemma *is-gcd-dvd*: *is-gcd m a b ==> k dvd a ==> k dvd b ==> k dvd m*
⟨*proof*⟩

Commutativity

lemma *is-gcd-commute*: *is-gcd k m n = is-gcd k n m*
⟨*proof*⟩

lemma *gcd-commute*: *gcd (m, n) = gcd (n, m)*
⟨*proof*⟩

lemma *gcd-assoc*: *gcd (gcd (k, m), n) = gcd (k, gcd (m, n))*
⟨*proof*⟩

lemma *gcd-0-left* [*simp*]: *gcd (0, m) = m*
⟨*proof*⟩

lemma *gcd-1-left* [*simp*]: *gcd (Suc 0, m) = 1*
⟨*proof*⟩

Multiplication laws

lemma *gcd-mult-distrib2*: *k * gcd (m, n) = gcd (k * m, k * n)*
— [1, page 27]
⟨*proof*⟩

lemma *gcd-mult* [*simp*]: *gcd (k, k * n) = k*
⟨*proof*⟩

lemma *gcd-self* [*simp*]: *gcd (k, k) = k*
⟨*proof*⟩

lemma *relprime-dvd-mult*: *gcd (k, n) = 1 ==> k dvd m * n ==> k dvd m*
⟨*proof*⟩

lemma *relprime-dvd-mult-iff*: *gcd (k, n) = 1 ==> (k dvd m * n) = (k dvd m)*
⟨*proof*⟩

lemma *gcd-mult-cancel*: *gcd* $(k, n) = 1 ==> gcd\ (k * m, n) = gcd\ (m, n)$
⟨*proof*⟩

Addition laws

lemma *gcd-add1* [*simp*]: *gcd* $(m + n, n) = gcd\ (m, n)$
⟨*proof*⟩

lemma *gcd-add2* [*simp*]: *gcd* $(m, m + n) = gcd\ (m, n)$
⟨*proof*⟩

lemma *gcd-add2'* [*simp*]: *gcd* $(m, n + m) = gcd\ (m, n)$
⟨*proof*⟩

lemma *gcd-add-mult*: *gcd* $(m, k * m + n) = gcd\ (m, n)$
⟨*proof*⟩

end

36 Binomial: Binomial Coefficients

theory *Binomial*
imports *GCD*
begin

This development is based on the work of Andy Gordon and Florian Kammueller

consts
binomial :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *choose 65*)

primrec
binomial-0: (*0* *choose* k) = (*if* $k = 0$ *then 1 else 0*)

binomial-Suc: (*Suc* n *choose* k) =
(*if* $k = 0$ *then 1 else* (n *choose* $(k - 1)$) + (n *choose* k))

lemma *binomial-n-0* [*simp*]: (n *choose 0*) = *1*
⟨*proof*⟩

lemma *binomial-0-Suc* [*simp*]: (*0 choose Suc* k) = *0*
⟨*proof*⟩

lemma *binomial-Suc-Suc* [*simp*]:
(*Suc* n *choose Suc* k) = (n *choose* k) + (n *choose Suc* k)
⟨*proof*⟩

lemma *binomial-eq-0* [*rule-format*]: $\forall k.\ n < k$ --> (n *choose* k) = *0*
⟨*proof*⟩

declare *binomial-0* [*simp del*] *binomial-Suc* [*simp del*]

lemma *binomial-n-n* [*simp*]: $(n\ choose\ n) = 1$
⟨*proof*⟩

lemma *binomial-Suc-n* [*simp*]: $(Suc\ n\ choose\ n) = Suc\ n$
⟨*proof*⟩

lemma *binomial-1* [*simp*]: $(n\ choose\ Suc\ 0) = n$
⟨*proof*⟩

lemma *zero-less-binomial* [*rule-format*]: $k \leq n\ {-}{-}{>}\ 0 < (n\ choose\ k)$
⟨*proof*⟩

lemma *binomial-eq-0-iff*: $(n\ choose\ k = 0) = (n<k)$
⟨*proof*⟩

lemma *zero-less-binomial-iff*: $(0 < n\ choose\ k) = (k \leq n)$
⟨*proof*⟩

lemma *Suc-times-binomial-eq* [*rule-format*]:
$\forall k.\ k \leq n\ {-}{-}{>}\ Suc\ n * (n\ choose\ k) = (Suc\ n\ choose\ Suc\ k) * Suc\ k$
⟨*proof*⟩

This is the well-known version, but it's harder to use because of the need to reason about division.

lemma *binomial-Suc-Suc-eq-times*:
$k \leq n\ {=}{=}{>}\ (Suc\ n\ choose\ Suc\ k) = (Suc\ n * (n\ choose\ k))\ div\ Suc\ k$
⟨*proof*⟩

Another version, with -1 instead of Suc.

lemma *times-binomial-minus1-eq*:
$[|k \leq n;\ \ 0<k|]\ {=}{=}{>}\ (n\ choose\ k) * k = n * ((n - 1)\ choose\ (k - 1))$
⟨*proof*⟩

36.0.1 Theorems about *choose*

Basic theorem about *choose*. By Florian Kammüller, tidied by LCP.

lemma *card-s-0-eq-empty*:
$finite\ A\ {=}{=}{>}\ card\ \{B.\ B \subseteq A\ \&\ card\ B = 0\} = 1$
⟨*proof*⟩

lemma *choose-deconstruct*: $finite\ M\ {=}{=}{>}\ x \notin M$
$\ {=}{=}{>}\ \{s.\ s <= insert\ x\ M\ \&\ card(s) = Suc\ k\}$
$= \{s.\ s <= M\ \&\ card(s) = Suc\ k\}\ Un$
$\{s.\ EX\ t.\ t <= M\ \&\ card(t) = k\ \&\ s = insert\ x\ t\}$

⟨*proof*⟩

There are as many subsets of A having cardinality k as there are sets obtained from the former by inserting a fixed element x into each.

lemma *constr-bij*:
$[|finite\ A;\ x \notin A|] ==>$
$card\ \{B.\ EX\ C.\ C <= A\ \&\ card(C) = k\ \&\ B = insert\ x\ C\} =$
$card\ \{B.\ B <= A\ \&\ card(B) = k\}$
⟨*proof*⟩

Main theorem: combinatorial statement about number of subsets of a set.

lemma *n-sub-lemma*:
$!!A.\ finite\ A ==> card\ \{B.\ B <= A\ \&\ card\ B = k\} = (card\ A\ choose\ k)$
⟨*proof*⟩

theorem *n-subsets*:
$finite\ A ==> card\ \{B.\ B <= A\ \&\ card\ B = k\} = (card\ A\ choose\ k)$
⟨*proof*⟩

The binomial theorem (courtesy of Tobias Nipkow):

theorem *binomial*: $(a+b::nat)\hat{\ }n = (\sum k{=}0..n.\ (n\ choose\ k) * a\hat{\ }k * b\hat{\ }(n-k))$
⟨*proof*⟩

end

37 PreList: A Basis for Building the Theory of Lists

theory *PreList*
imports *Wellfounded-Relations Presburger Relation-Power Binomial*
begin

Is defined separately to serve as a basis for theory ToyList in the documentation.

end

38 List: The datatype of finite lists

theory *List*
imports *PreList*
begin

datatype *'a list* =
Nil ([])
| *Cons 'a 'a list* (**infixr** # *65*)

38.1 Basic list processing functions

consts

```
@ :: 'a list => 'a list => 'a list    (infixr 65)
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
list-all2 :: ('a => 'b => bool) => 'a list => 'b list => bool
map :: ('a=>'b) => ('a list => 'b list)
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
null:: 'a list => bool
distinct:: 'a list => bool
replicate :: nat => 'a => 'a list
rotate1 :: 'a list ⇒ 'a list
rotate :: nat ⇒ 'a list ⇒ 'a list
sublist :: 'a list => nat set => 'a list

mem :: 'a => 'a list => bool    (infixl 55)
list-inter :: 'a list ⇒ 'a list ⇒ 'a list
list-ex :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
list-all:: ('a => bool) => ('a list => bool)
itrev :: 'a list ⇒ 'a list ⇒ 'a list
filtermap :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list
map-filter :: ('a => 'b) => ('a => bool) => 'a list => 'b 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)

upto:: *nat* => *nat* => *nat list* ((1[-../-]))

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*

[*i*..*j*] == [*i*..<(*Suc j*)]

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*.

syntax *length* :: 'a *list* => *nat*
translations *length* => *size* :: - *list* => *nat*

⟨*ML*⟩

primrec
hd(*x*#*xs*) = *x*

primrec
tl([]) = []
tl(*x*#*xs*) = *xs*

primrec
null([]) = *True*
null(*x*#*xs*) = *False*

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
$append\text{-}Nil{:}[]@ys = ys$
$append\text{-}Cons{:}\ (x\#xs)@ys = x\#(xs@ys)$

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
$foldl\text{-}Nil{:}foldl\ f\ a\ [] = a$
$foldl\text{-}Cons{:}\ foldl\ f\ a\ (x\#xs) = foldl\ f\ (f\ a\ x)\ xs$

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
$drop\text{-}Nil{:}drop\ n\ [] = []$
$drop\text{-}Cons{:}\ drop\ n\ (x\#xs) = (case\ n\ of\ 0 => x\#xs\ |\ Suc(m) => drop\ m\ xs)$
— Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = Suc\ k$

primrec
$take\text{-}Nil{:}take\ n\ [] = []$
$take\text{-}Cons{:}\ take\ n\ (x\#xs) = (case\ n\ of\ 0 => []\ |\ Suc(m) => x\ \#\ take\ m\ xs)$
— Warning: simpset does not contain this definition, but separate theorems for $n = 0$ and $n = Suc\ k$

primrec
$nth\text{-}Cons{:}(x\#xs)!n = (case\ n\ of\ 0 => x\ |\ (Suc\ k) => xs!k)$
— Warning: simpset does not contain this definition, but separate theorems for

$n = 0$ and $n = Suc\ k$

primrec
$[][i{:=}v] = []$
$(x\#xs)[i{:=}v] = (case\ i\ of\ 0\ => v\ \#\ xs \mid Suc\ j\ => x\ \#\ xs[j{:=}v])$

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
$zip\ xs\ []\ = []$
$zip\text{-}Cons$: $zip\ xs\ (y\#ys) = (case\ xs\ of\ []\ => []\ \mid z\#zs\ => (z,y)\#zip\ zs\ ys)$
— Warning: simpset does not contain this definition, but separate theorems for $xs = []$ and $xs = z\ \#\ zs$

primrec
$upt\text{-}0$: $[i..{<}0] = []$
$upt\text{-}Suc$: $[i..{<}(Suc\ j)] = (if\ i <= j\ then\ [i..{<}j]\ @\ [j]\ else\ [])$

primrec
$distinct\ []\ = True$
$distinct\ (x\#xs) = (x\ {\sim}{:}\ set\ xs \wedge distinct\ xs)$

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
$replicate\text{-}0$: $replicate\ 0\ x = []$
$replicate\text{-}Suc$: $replicate\ (Suc\ n)\ x = x\ \#\ replicate\ n\ x$

defs
$rotate1\text{-}def$: $rotate1\ xs == (case\ xs\ of\ []\ \Rightarrow []\ \mid x\#xs \Rightarrow xs\ @\ [x])$
$rotate\text{-}def$: $rotate\ n == rotate1\ \hat{}\ n$

$list\text{-}all2\text{-}def$:
$list\text{-}all2\ P\ xs\ ys ==$
$length\ xs = length\ ys \wedge (\forall (x,\ y) \in set\ (zip\ xs\ ys).\ P\ x\ y)$

$sublist\text{-}def$:

sublist xs A == map fst (filter (%p. snd p : A) (zip xs [0..<size xs]))

primrec
x mem [] = False
x mem (y#ys) = (if y=x then True else x mem ys)

primrec
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 P [] = True
list-all P (x#xs) = (P(x) ∧ list-all P xs)

primrec
list-ex P [] = False
list-ex P (x#xs) = (P x ∨ list-ex P xs)

primrec
filtermap f [] = []
filtermap f (x#xs) =
(case f x of None ⇒ filtermap f xs
| Some y ⇒ y # (filtermap f xs))

primrec
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)

primrec
itrev [] ys = ys
itrev (x#xs) ys = itrev xs (x#ys)

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*⟩

38.1.1 *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-Suc-conv*:
(*length xs* = *Suc n*) = (∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n*)
⟨*proof*⟩

lemma *Suc-length-conv*:
(*Suc n* = *length xs*) = (∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n*)
⟨*proof*⟩

lemma *impossible-Cons* [*rule-format*]:
length xs <= *length ys* −−> *xs* = *x* # *ys* = *False*
⟨*proof*⟩

lemma *list-induct2*[*consumes 1*]: ⋀*ys*.
⟦ *length xs* = *length ys*;
P [] [];
⋀*x xs y ys*. ⟦ *length xs* = *length ys*; *P xs ys* ⟧ ⟹ *P* (*x*#*xs*) (*y*#*ys*) ⟧
⟹ *P xs ys*
⟨*proof*⟩

38.1.2 @ − append

lemma *append-assoc* [*simp*]: (*xs* @ *ys*) @ *zs* = *xs* @ (*ys* @ *zs*)
⟨*proof*⟩

lemma *append-Nil2* [*simp*]: *xs* @ [] = *xs*
⟨*proof*⟩

lemma *append-is-Nil-conv* [*iff*]: (*xs* @ *ys* = []) = (*xs* = [] ∧ *ys* = [])

⟨*proof*⟩

lemma *Nil-is-append-conv* [*iff*]: $([] = xs\ @\ ys) = (xs = [] \wedge ys = [])$
⟨*proof*⟩

lemma *append-self-conv* [*iff*]: $(xs\ @\ ys = xs) = (ys = [])$
⟨*proof*⟩

lemma *self-append-conv* [*iff*]: $(xs = xs\ @\ ys) = (ys = [])$
⟨*proof*⟩

lemma *append-eq-append-conv* [*simp*]:
$!!ys.\ length\ xs = length\ ys \vee length\ us = length\ vs$
$==> (xs@us = ys@vs) = (xs{=}ys \wedge us{=}vs)$
⟨*proof*⟩

lemma *append-eq-append-conv2*: $!!ys\ zs\ ts.$
$(xs\ @\ ys = zs\ @\ ts) =$
$(EX\ us.\ xs = zs\ @\ us\ \&\ us\ @\ ys = ts \mid xs\ @\ us = zs\ \&\ ys = us@\ ts)$
⟨*proof*⟩

lemma *same-append-eq* [*iff*]: $(xs\ @\ ys = xs\ @\ zs) = (ys = zs)$
⟨*proof*⟩

lemma *append1-eq-conv* [*iff*]: $(xs\ @\ [x] = ys\ @\ [y]) = (xs = ys \wedge x = y)$
⟨*proof*⟩

lemma *append-same-eq* [*iff*]: $(ys\ @\ xs = zs\ @\ xs) = (ys = zs)$
⟨*proof*⟩

lemma *append-self-conv2* [*iff*]: $(xs\ @\ ys = ys) = (xs = [])$
⟨*proof*⟩

lemma *self-append-conv2* [*iff*]: $(ys = xs\ @\ ys) = (xs = [])$
⟨*proof*⟩

lemma *hd-Cons-tl* [*simp*]: $xs \neq [] ==> hd\ xs\ \#\ tl\ xs = xs$
⟨*proof*⟩

lemma *hd-append*: $hd\ (xs\ @\ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
⟨*proof*⟩

lemma *hd-append2* [*simp*]: $xs \neq [] ==> hd\ (xs\ @\ ys) = hd\ xs$
⟨*proof*⟩

lemma *tl-append*: $tl\ (xs\ @\ ys) = (case\ xs\ of\ [] => tl\ ys \mid z\#zs => zs\ @\ ys)$
⟨*proof*⟩

lemma *tl-append2* [*simp*]: $xs \neq [] ==> tl\ (xs\ @\ ys) = tl\ xs\ @\ ys$

⟨*proof*⟩

lemma *Cons-eq-append-conv*: $x\#xs = ys@zs =$
$(ys = [] \ \& \ x\#xs = zs \mid (EX\ ys'.\ x\#ys' = ys \ \& \ xs = ys'@zs))$
⟨*proof*⟩

lemma *append-eq-Cons-conv*: $(ys@zs = x\#xs) =$
$(ys = [] \ \& \ zs = x\#xs \mid (EX\ ys'.\ ys = x\#ys' \ \& \ ys'@zs = xs))$
⟨*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*⟩

38.1.3 *map*

lemma *map-ext*: $(!!x.\ x : set\ xs \ --> f\ x = g\ x) ==> map\ f\ xs = map\ g\ xs$
⟨*proof*⟩

lemma *map-ident* [*simp*]: $map\ (\lambda x.\ x) = (\lambda xs.\ xs)$
⟨*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* [*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*[*iff*]:
$(map\ f\ xs = y\#ys) = (\exists\, z\ zs.\ xs = z\#zs \wedge f\ z = y \wedge map\ f\ zs = ys)$
⟨*proof*⟩

lemma *Cons-eq-map-conv*[*iff*]:
$(x\#xs = map\ f\ ys) = (\exists\, z\ zs.\ ys = z\#zs \wedge x = f\ z \wedge xs = map\ f\ zs)$
⟨*proof*⟩

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*:
$!!xs.\ map\ f\ xs = map\ f\ ys ==> length\ xs = length\ ys$
⟨*proof*⟩

lemma *map-inj-on*:
$[|\ map\ f\ xs = map\ f\ ys;\ inj\text{-}on\ f\ (set\ xs\ Un\ set\ ys)\ |]$
$==> xs = ys$
⟨*proof*⟩

lemma *inj-on-map-eq-map*:
$inj\text{-}on\ f\ (set\ xs\ Un\ set\ ys) \Longrightarrow (map\ f\ xs = map\ f\ ys) = (xs = ys)$
⟨*proof*⟩

lemma *map-injective*:
$!!xs.\ map\ f\ xs = map\ f\ ys ==> inj\ f ==> xs = ys$
⟨*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*: $inj\text{-}on\ f\ (\bigcup(set\ `\ A)) \Longrightarrow inj\text{-}on\ (map\ f)\ A$
⟨proof⟩

lemma *map-idI*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = x) \Longrightarrow map\ f\ xs = xs$
⟨proof⟩

lemma *map-fun-upd* [*simp*]: $y \notin set\ xs \Longrightarrow map\ (f(y{:=}v))\ xs = map\ f\ xs$
⟨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⟩

38.1.4 *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*]: $!!ys.\ (rev\ xs = rev\ ys) = (xs = ys)$
⟨proof⟩

lemma *inj-on-rev*[*iff*]: $inj\text{-}on\ rev\ A$
⟨proof⟩

lemma *rev-induct* [*case-names Nil snoc*]:

[| P []; !!x xs. P xs ==> P (xs @ [x]) |] ==> P xs
⟨*proof*⟩

⟨*ML*⟩

lemma *rev-exhaust* [*case-names Nil snoc*]:
(xs = [] ==> P) ==>(!!ys y. xs = ys @ [y] ==> P) ==> P
⟨*proof*⟩

lemmas *rev-cases* = *rev-exhaust*

38.1.5 *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*: $l = x\#xs \Longrightarrow x \in set\ l$
⟨*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* $\wedge$ P x}
⟨*proof*⟩

lemma *set-upt* [*simp*]: *set*[i..<j] = {k. $i \leq k \wedge k < j$}
⟨*proof*⟩

lemma *in-set-conv-decomp*: (x : *set xs*) = ($\exists$ *ys zs*. *xs* = *ys* @ x # *zs*)
⟨*proof*⟩

lemma *finite-list*: *finite A ==> EX l. set l = A*
⟨*proof*⟩

lemma *card-length*: *card (set xs)* $\leq$ *length xs*
⟨*proof*⟩

38.1.6 *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 *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 ==> 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)}

$\langle proof \rangle$

lemma *Cons-eq-filterD*:
$x\#xs = filter\ P\ ys \Longrightarrow$
$\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
(**concl is** $\exists\, us\ vs.\ ?P\ ys\ us\ vs$)
$\langle proof \rangle$

lemma *filter-eq-ConsD*:
$filter\ P\ ys = x\#xs \Longrightarrow$
$\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
$\langle proof \rangle$

lemma *filter-eq-Cons-iff*:
$(filter\ P\ ys = x\#xs) =$
$(\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
$\langle proof \rangle$

lemma *Cons-eq-filter-iff*:
$(x\#xs = filter\ P\ ys) =$
$(\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
$\langle proof \rangle$

lemma *filter-cong*:
$xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ ys \Longrightarrow P\ x = Q\ x) \Longrightarrow filter\ P\ xs = filter\ Q\ ys$
$\langle proof \rangle$

38.1.7 *concat*

lemma *concat-append* [*simp*]: $concat\ (xs\ @\ ys) = concat\ xs\ @\ concat\ ys$
$\langle proof \rangle$

lemma *concat-eq-Nil-conv* [*iff*]: $(concat\ xss = []) = (\forall\, xs \in set\ xss.\ xs = [])$
$\langle proof \rangle$

lemma *Nil-eq-concat-conv* [*iff*]: $([] = concat\ xss) = (\forall\, xs \in set\ xss.\ xs = [])$
$\langle proof \rangle$

lemma *set-concat* [*simp*]: $set\ (concat\ xs) = \bigcup (set\ `\ set\ xs)$
$\langle proof \rangle$

lemma *map-concat*: $map\ f\ (concat\ xs) = concat\ (map\ (map\ f)\ xs)$
$\langle proof \rangle$

lemma *filter-concat*: $filter\ p\ (concat\ xs) = concat\ (map\ (filter\ p)\ xs)$
$\langle proof \rangle$

lemma *rev-concat*: $rev\ (concat\ xs) = concat\ (map\ rev\ (rev\ xs))$
$\langle proof \rangle$

38.1.8 *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*:
!!*n*. (*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*. *n* < *length xs* ==> (*map f xs*)!*n* = *f*(*xs*!*n*)
⟨*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*⟩

38.1.9 *list-update*

lemma *length-list-update* [*simp*]: !!*i*. *length*(*xs*[*i*:=*x*]) = *length xs*
⟨*proof*⟩

lemma *nth-list-update*:
!!*i j*. *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*]: $!!i\ j.\ i \neq j ==> xs[i:=x]!j = xs!j$
⟨*proof*⟩

lemma *list-update-overwrite* [*simp*]:
$!!i.\ i < size\ xs ==> xs[i:=x,\ i:=y] = xs[i:=y]$
⟨*proof*⟩

lemma *list-update-id*[*simp*]: $!!i.\ i < length\ xs ==> xs[i := xs!i] = xs$
⟨*proof*⟩

lemma *list-update-beyond*[*simp*]: $\bigwedge i.\ length\ xs \leq i \Longrightarrow xs[i:=x] = xs$
⟨*proof*⟩

lemma *list-update-same-conv*:
$!!i.\ i < length\ xs ==> (xs[i := x] = xs) = (xs!i = x)$
⟨*proof*⟩

lemma *list-update-append1*:
$!!i.\ i < size\ xs \Longrightarrow (xs\ @\ ys)[i:=x] = xs[i:=x]\ @\ ys$
⟨*proof*⟩

lemma *list-update-append*:
$!!n.\ (xs\ @\ ys)\ [n:= x] =$
$(if\ n < length\ xs\ then\ xs[n:= x]\ @\ ys\ else\ xs\ @\ (ys\ [n-length\ xs:= x]))$
⟨*proof*⟩

lemma *list-update-length* [*simp*]:
$(xs\ @\ x\ \#\ ys)[length\ xs := y] = (xs\ @\ y\ \#\ ys)$
⟨*proof*⟩

lemma *update-zip*:
$!!i\ xy\ xs.\ length\ xs = length\ ys ==>$
$(zip\ xs\ ys)[i:=xy] = zip\ (xs[i:=fst\ xy])\ (ys[i:=snd\ xy])$
⟨*proof*⟩

lemma *set-update-subset-insert*: $!!i.\ set(xs[i:=x]) <= insert\ x\ (set\ xs)$
⟨*proof*⟩

lemma *set-update-subsetI*: $[|\ set\ xs <= A;\ x{:}A\ |] ==> set(xs[i := x]) <= A$
⟨*proof*⟩

lemma *set-update-memI*: $!!n.\ n < length\ xs \Longrightarrow x \in set\ (xs[n := x])$
⟨*proof*⟩

38.1.10 *last* and *butlast*

lemma *last-snoc* [*simp*]: $last\ (xs\ @\ [x]) = x$
⟨*proof*⟩

lemma *butlast-snoc* [*simp*]: $butlast\ (xs\ @\ [x]) = xs$
⟨*proof*⟩

lemma *last-ConsL*: $xs = [] \Longrightarrow last(x\#xs) = x$
⟨*proof*⟩

lemma *last-ConsR*: $xs \neq [] \Longrightarrow last(x\#xs) = last\ xs$
⟨*proof*⟩

lemma *last-append*: $last(xs\ @\ ys) = (if\ ys = []\ then\ last\ xs\ else\ last\ ys)$
⟨*proof*⟩

lemma *last-appendL*[*simp*]: $ys = [] \Longrightarrow last(xs\ @\ ys) = last\ xs$
⟨*proof*⟩

lemma *last-appendR*[*simp*]: $ys \neq [] \Longrightarrow last(xs\ @\ ys) = last\ ys$
⟨*proof*⟩

lemma *length-butlast* [*simp*]: $length\ (butlast\ xs) = length\ xs - 1$
⟨*proof*⟩

lemma *butlast-append*:
$!!ys.\ butlast\ (xs\ @\ ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs\ @\ butlast\ ys)$
⟨*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*:
$x : set\ (butlast\ xs)\ |\ x : set\ (butlast\ ys) ==> x : set\ (butlast\ (xs\ @\ ys))$
⟨*proof*⟩

lemma *last-drop*[*simp*]: $!!n.\ n < length\ xs \Longrightarrow last\ (drop\ n\ xs) = last\ xs$
⟨*proof*⟩

lemma *last-conv-nth*: $xs{\neq}[] \Longrightarrow last\ xs = xs!(length\ xs - 1)$
⟨*proof*⟩

38.1.11 *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 *drop-tl*: !!*n*. *drop n* (*tl xs*) = *tl*(*drop n xs*)
⟨*proof*⟩

lemma *nth-via-drop*: !!*n*. *drop n xs* = *y*#*ys* ⟹ *xs*!*n* = *y*
⟨*proof*⟩

lemma *take-Suc-conv-app-nth*:
!!*i*. *i* < *length xs* ⟹ *take* (*Suc i*) *xs* = *take i xs* @ [*xs*!*i*]
⟨*proof*⟩

lemma *drop-Suc-conv-tl*:
!!*i*. *i* < *length xs* ⟹ (*xs*!*i*) # (*drop* (*Suc i*) *xs*) = *drop i xs*
⟨*proof*⟩

lemma *length-take* [*simp*]: !!*xs*. *length* (*take n xs*) = *min* (*length xs*) *n*
⟨*proof*⟩

lemma *length-drop* [*simp*]: !!*xs*. *length* (*drop n xs*) = (*length xs* − *n*)
⟨*proof*⟩

lemma *take-all* [*simp*]: !!*xs*. *length xs* <= *n* ==> *take n xs* = *xs*
⟨*proof*⟩

lemma *drop-all* [*simp*]: !!*xs*. *length xs* <= *n* ==> *drop n xs* = []
⟨*proof*⟩

lemma *take-append* [*simp*]:
!!*xs*. *take n* (*xs* @ *ys*) = (*take n xs* @ *take* (*n* − *length xs*) *ys*)

⟨*proof*⟩

lemma *drop-append* [*simp*]:
$!!xs.\ drop\ n\ (xs\ @\ ys) = drop\ n\ xs\ @\ drop\ (n - length\ xs)\ ys$
⟨*proof*⟩

lemma *take-take* [*simp*]: $!!xs\ n.\ take\ n\ (take\ m\ xs) = take\ (min\ n\ m)\ xs$
⟨*proof*⟩

lemma *drop-drop* [*simp*]: $!!xs.\ drop\ n\ (drop\ m\ xs) = drop\ (n + m)\ xs$
⟨*proof*⟩

lemma *take-drop*: $!!xs\ n.\ take\ n\ (drop\ m\ xs) = drop\ m\ (take\ (n + m)\ xs)$
⟨*proof*⟩

lemma *drop-take*: $!!m\ n.\ drop\ n\ (take\ m\ xs) = take\ (m{-}n)\ (drop\ n\ xs)$
⟨*proof*⟩

lemma *append-take-drop-id* [*simp*]: $!!xs.\ take\ n\ xs\ @\ drop\ n\ xs = xs$
⟨*proof*⟩

lemma *take-eq-Nil*[*simp*]: $!!n.\ (take\ n\ xs = []) = (n = 0 \vee xs = [])$
⟨*proof*⟩

lemma *drop-eq-Nil*[*simp*]: $!!n.\ (drop\ n\ xs = []) = (length\ xs <= n)$
⟨*proof*⟩

lemma *take-map*: $!!xs.\ take\ n\ (map\ f\ xs) = map\ f\ (take\ n\ xs)$
⟨*proof*⟩

lemma *drop-map*: $!!xs.\ drop\ n\ (map\ f\ xs) = map\ f\ (drop\ n\ xs)$
⟨*proof*⟩

lemma *rev-take*: $!!i.\ rev\ (take\ i\ xs) = drop\ (length\ xs - i)\ (rev\ xs)$
⟨*proof*⟩

lemma *rev-drop*: $!!i.\ rev\ (drop\ i\ xs) = take\ (length\ xs - i)\ (rev\ xs)$
⟨*proof*⟩

lemma *nth-take* [*simp*]: $!!n\ i.\ i < n ==> (take\ n\ xs)!i = xs!i$
⟨*proof*⟩

lemma *nth-drop* [*simp*]:
$!!xs\ i.\ n + i <= length\ xs ==> (drop\ n\ xs)!i = xs!(n + i)$
⟨*proof*⟩

lemma *set-take-subset*: $\bigwedge n.\ set(take\ n\ xs) \subseteq set\ xs$
⟨*proof*⟩

lemma *set-drop-subset*: $\bigwedge n.\ set(drop\ n\ xs) \subseteq set\ xs$
⟨*proof*⟩

lemma *in-set-takeD*: $x : set(take\ n\ xs) \Longrightarrow x : set\ xs$
⟨*proof*⟩

lemma *in-set-dropD*: $x : set(drop\ n\ xs) \Longrightarrow x : set\ xs$
⟨*proof*⟩

lemma *append-eq-conv-conj*:
$!!zs.\ (xs\ @\ ys = zs) = (xs = take\ (length\ xs)\ zs \wedge ys = drop\ (length\ xs)\ zs)$
⟨*proof*⟩

lemma *take-add* [*rule-format*]:
$\forall i.\ i{+}j \leq length(xs)\ {-}{-}{>}\ take\ (i{+}j)\ xs = take\ i\ xs\ @\ take\ j\ (drop\ i\ xs)$
⟨*proof*⟩

lemma *append-eq-append-conv-if*:
$!!\ ys_1.\ (xs_1\ @\ xs_2 = ys_1\ @\ ys_2) =$
$(if\ size\ xs_1 \leq size\ ys_1$
$then\ xs_1 = take\ (size\ xs_1)\ ys_1 \wedge xs_2 = drop\ (size\ xs_1)\ ys_1\ @\ ys_2$
$else\ take\ (size\ ys_1)\ xs_1 = ys_1 \wedge drop\ (size\ ys_1)\ xs_1\ @\ xs_2 = ys_2)$
⟨*proof*⟩

lemma *take-hd-drop*:
$!!n.\ n < length\ xs \Longrightarrow take\ n\ xs\ @\ [hd\ (drop\ n\ xs)] = take\ (n{+}1)\ xs$
⟨*proof*⟩

lemma *id-take-nth-drop*:
$i < length\ xs \Longrightarrow xs = take\ i\ xs\ @\ xs!i\ \#\ drop\ (Suc\ i)\ xs$
⟨*proof*⟩

lemma *upd-conv-take-nth-drop*:
$i < length\ xs \Longrightarrow xs[i{:=}a] = take\ i\ xs\ @\ a\ \#\ drop\ (Suc\ i)\ xs$
⟨*proof*⟩

38.1.12 *takeWhile* and *dropWhile*

lemma *takeWhile-dropWhile-id* [*simp*]: $takeWhile\ P\ xs\ @\ dropWhile\ P\ xs = xs$
⟨*proof*⟩

lemma *takeWhile-append1* [*simp*]:
$[|\ x{:}set\ xs;\ {}^{\sim}P(x)|]\ {=}{=}{>}\ takeWhile\ P\ (xs\ @\ ys) = takeWhile\ P\ xs$
⟨*proof*⟩

lemma *takeWhile-append2* [*simp*]:
$(!!x.\ x : set\ xs\ {=}{=}{>}\ P\ x)\ {=}{=}{>}\ takeWhile\ P\ (xs\ @\ ys) = xs\ @\ takeWhile\ P\ ys$
⟨*proof*⟩

lemma *takeWhile-tail*: $\neg$ 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-take-whileD*: x : set $(takeWhile$ P $xs)$ ==> x : set xs $\wedge$ P x
⟨*proof*⟩

lemma *takeWhile-eq-all-conv*[*simp*]:
$(takeWhile\ P\ xs = xs) = (\forall x \in set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *dropWhile-eq-Nil-conv*[*simp*]:
$(dropWhile\ P\ xs = []) = (\forall x \in set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *dropWhile-eq-Cons-conv*:
$(dropWhile\ P\ xs = y\#ys) = (xs = takeWhile\ P\ xs$ @ $y\ \#\ ys\ \&\ \neg\ P\ y)$
⟨*proof*⟩

The following two lemmmas could be generalized to an arbitrary property.

lemma *takeWhile-neq-rev*: $[\![distinct\ xs;\ x \in set\ xs]\!] \Longrightarrow$
$takeWhile\ (\lambda y.\ y \neq x)\ (rev\ xs) = rev\ (tl\ (dropWhile\ (\lambda y.\ y \neq x)\ xs))$
⟨*proof*⟩

lemma *dropWhile-neq-rev*: $[\![distinct\ xs;\ x \in set\ xs]\!] \Longrightarrow$
$dropWhile\ (\lambda y.\ y \neq x)\ (rev\ xs) = x\ \#\ rev\ (takeWhile\ (\lambda y.\ y \neq x)\ xs)$
⟨*proof*⟩

38.1.13 *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*:
$zip\ (x\#xs)\ ys = (case\ ys\ of\ [] \Rightarrow []\ |\ y\#ys \Rightarrow (x,y)\#zip\ xs\ ys)$
⟨*proof*⟩

lemma *length-zip* [*simp*]:
!!*xs*. *length* (*zip xs ys*) = *min* (*length xs*) (*length ys*)
⟨*proof*⟩

lemma *zip-append1*:
!!*xs*. *zip* (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
⟨*proof*⟩

lemma *zip-append2*:
!!*ys*. *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 *nth-zip* [*simp*]:
!!*i xs*. [| *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*]:
!!*j*. *zip* (*replicate i x*) (*replicate j y*) = *replicate* (*min i j*) (*x*,*y*)
⟨*proof*⟩

38.1.14 *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*]: *list-all2 P xs* [] = (*xs* = [])

⟨*proof*⟩

lemma *list-all2-Cons* [*iff*,*code*]:
$list\text{-}all2\ P\ (x \# xs)\ (y \# ys) = (P\ x\ y \wedge list\text{-}all2\ P\ xs\ ys)$
⟨*proof*⟩

lemma *list-all2-Cons1*:
$list\text{-}all2\ P\ (x \# xs)\ ys = (\exists z\ zs.\ ys = z \# zs \wedge P\ x\ z \wedge list\text{-}all2\ P\ xs\ zs)$
⟨*proof*⟩

lemma *list-all2-Cons2*:
$list\text{-}all2\ P\ xs\ (y \# ys) = (\exists z\ zs.\ xs = z \# zs \wedge P\ z\ y \wedge list\text{-}all2\ P\ zs\ ys)$
⟨*proof*⟩

lemma *list-all2-rev* [*iff*]:
$list\text{-}all2\ P\ (rev\ xs)\ (rev\ ys) = list\text{-}all2\ P\ xs\ ys$
⟨*proof*⟩

lemma *list-all2-rev1*:
$list\text{-}all2\ P\ (rev\ xs)\ ys = list\text{-}all2\ P\ xs\ (rev\ ys)$
⟨*proof*⟩

lemma *list-all2-append1*:
$list\text{-}all2\ P\ (xs\ @\ ys)\ zs =$
$(EX\ us\ vs.\ zs = us\ @\ vs \wedge length\ us = length\ xs \wedge length\ vs = length\ ys \wedge$
$list\text{-}all2\ P\ xs\ us \wedge list\text{-}all2\ P\ ys\ vs)$
⟨*proof*⟩

lemma *list-all2-append2*:
$list\text{-}all2\ P\ xs\ (ys\ @\ zs) =$
$(EX\ us\ vs.\ xs = us\ @\ vs \wedge length\ us = length\ ys \wedge length\ vs = length\ zs \wedge$
$list\text{-}all2\ P\ us\ ys \wedge list\text{-}all2\ P\ vs\ zs)$
⟨*proof*⟩

lemma *list-all2-append*:
$length\ xs = length\ ys \Longrightarrow$
$list\text{-}all2\ P\ (xs@us)\ (ys@vs) = (list\text{-}all2\ P\ xs\ ys \wedge list\text{-}all2\ P\ us\ vs)$
⟨*proof*⟩

lemma *list-all2-appendI* [*intro?*, *trans*]:
$[\![\ list\text{-}all2\ P\ a\ b;\ list\text{-}all2\ P\ c\ d\ ]\!] \Longrightarrow list\text{-}all2\ P\ (a@c)\ (b@d)$
⟨*proof*⟩

lemma *list-all2-conv-all-nth*:
$list\text{-}all2\ P\ xs\ ys =$
$(length\ xs = length\ ys \wedge (\forall i < length\ xs.\ P\ (xs!i)\ (ys!i)))$
⟨*proof*⟩

lemma *list-all2-trans*:

assumes *tr*: $!!a\ b\ c.\ P1\ a\ b\ {=}{=}{>}\ P2\ b\ c\ {=}{=}{>}\ P3\ a\ c$
shows $!!bs\ cs.\ list\text{-}all2\ P1\ as\ bs\ {=}{=}{>}\ list\text{-}all2\ P2\ bs\ cs\ {=}{=}{>}\ list\text{-}all2\ P3\ as\ cs$
(**is** $!!bs\ cs.\ PROP\ ?Q\ as\ bs\ cs$)
⟨*proof*⟩

lemma *list-all2-all-nthI* [*intro?*]:
$length\ a = length\ b \Longrightarrow (\bigwedge n.\ n < length\ a \Longrightarrow P\ (a!n)\ (b!n)) \Longrightarrow list\text{-}all2\ P\ a\ b$
⟨*proof*⟩

lemma *list-all2I*:
$\forall x \in set\ (zip\ a\ b).\ split\ P\ x \Longrightarrow length\ a = length\ b \Longrightarrow list\text{-}all2\ P\ a\ b$
⟨*proof*⟩

lemma *list-all2-nthD*:
$[\![\ list\text{-}all2\ P\ xs\ ys;\ p < size\ xs\ ]\!] \Longrightarrow P\ (xs!p)\ (ys!p)$
⟨*proof*⟩

lemma *list-all2-nthD2*:
$[\![list\text{-}all2\ P\ xs\ ys;\ p < size\ ys]\!] \Longrightarrow P\ (xs!p)\ (ys!p)$
⟨*proof*⟩

lemma *list-all2-map1*:
$list\text{-}all2\ P\ (map\ f\ as)\ bs = list\text{-}all2\ (\lambda x\ y.\ P\ (f\ x)\ y)\ as\ bs$
⟨*proof*⟩

lemma *list-all2-map2*:
$list\text{-}all2\ P\ as\ (map\ f\ bs) = list\text{-}all2\ (\lambda x\ y.\ P\ x\ (f\ y))\ as\ bs$
⟨*proof*⟩

lemma *list-all2-refl* [*intro?*]:
$(\bigwedge x.\ P\ x\ x) \Longrightarrow list\text{-}all2\ P\ xs\ xs$
⟨*proof*⟩

lemma *list-all2-update-cong*:
$[\![\ i{<}size\ xs;\ list\text{-}all2\ P\ xs\ ys;\ P\ x\ y\ ]\!] \Longrightarrow list\text{-}all2\ P\ (xs[i{:=}x])\ (ys[i{:=}y])$
⟨*proof*⟩

lemma *list-all2-update-cong2*:
$[\![list\text{-}all2\ P\ xs\ ys;\ P\ x\ y;\ i < length\ ys]\!] \Longrightarrow list\text{-}all2\ P\ (xs[i{:=}x])\ (ys[i{:=}y])$
⟨*proof*⟩

lemma *list-all2-takeI* [*simp*,*intro?*]:
$\bigwedge n\ ys.\ list\text{-}all2\ P\ xs\ ys \Longrightarrow list\text{-}all2\ P\ (take\ n\ xs)\ (take\ n\ ys)$
⟨*proof*⟩

lemma *list-all2-dropI* [*simp*,*intro?*]:
$\bigwedge n\ bs.\ list\text{-}all2\ P\ as\ bs \Longrightarrow list\text{-}all2\ P\ (drop\ n\ as)\ (drop\ n\ bs)$
⟨*proof*⟩

lemma *list-all2-mono* [*intro?*]:
$\bigwedge y.\ list\text{-}all2\ P\ x\ y \Longrightarrow (\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow list\text{-}all2\ Q\ x\ y$
⟨*proof*⟩

38.1.15 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:
$!!a.\ foldl\ f\ a\ (xs\ @\ ys) = foldl\ f\ (foldl\ f\ a\ xs)\ ys$
⟨*proof*⟩

lemma *foldr-append*[*simp*]: $foldr\ f\ (xs\ @\ ys)\ a = foldr\ f\ xs\ (foldr\ f\ ys\ a)$
⟨*proof*⟩

lemma *foldr-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*⟩

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*: $!!n::nat.\ m <= n ==> m <= foldl\ (op\ +)\ n\ ns$
⟨*proof*⟩

lemma *elem-le-sum*: $!!n::nat.\ n : set\ ns ==> n <= foldl\ (op\ +)\ 0\ ns$
⟨*proof*⟩

lemma *sum-eq-0-conv* [*iff*]:
$!!m::nat.\ (foldl\ (op\ +)\ m\ ns = 0) = (m = 0 \wedge (\forall n \in set\ ns.\ n = 0))$
⟨*proof*⟩

38.1.16 *upto*

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*:
$!!x\ xs.\ ([i..<j] = x\#xs) = (i < j\ \&\ i = x\ \&\ [i+1..<j] = xs)$
⟨*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*: $i < j \text{ ==> } [i..<j] = i \ \# \ [Suc \ i..<j]$
⟨*proof*⟩

lemma *upt-add-eq-append*: $i<=j \text{ ==> } [i..<j+k] = [i..<j]@[j..<j+k]$
— LOOPS as a simprule, since $j <= j$.
⟨*proof*⟩

lemma *length-upt* [*simp*]: $length \ [i..<j] = j - i$
⟨*proof*⟩

lemma *nth-upt* [*simp*]: $i + k < j \text{ ==> } [i..<j] \ ! \ k = i + k$
⟨*proof*⟩

lemma *take-upt* [*simp*]: $!!i. \ i+m <= n \text{ ==> } take \ m \ [i..<n] = [i..<i+m]$
⟨*proof*⟩

lemma *drop-upt*[*simp*]: $drop \ m \ [i..<j] = [i+m..<j]$
⟨*proof*⟩

lemma *map-Suc-upt*: $map \ Suc \ [m..<n] = [Suc \ m..n]$
⟨*proof*⟩

lemma *nth-map-upt*: $!!i. \ i < n-m \text{ ==> } (map \ f \ [m..<n]) \ ! \ i = f(m+i)$
⟨*proof*⟩

lemma *nth-take-lemma*:
$!!xs \ ys. \ k <= length \ xs \text{ ==> } k <= length \ ys \text{ ==> }$
$(!!i. \ i < k \text{ --> } xs!i = ys!i) \text{ ==> } take \ k \ xs = take \ k \ ys$
⟨*proof*⟩

lemma *nth-equalityI*:
$[| \ length \ xs = length \ ys; \ ALL \ i < length \ xs. \ xs!i = ys!i \ |] \text{ ==> } xs = ys$
⟨*proof*⟩

lemma *list-all2-antisym*:
$[\![\ (\bigwedge x \ y. \ [\![P \ x \ y; \ Q \ y \ x]\!] \Longrightarrow x = y); \ list\text{-}all2 \ P \ xs \ ys; \ list\text{-}all2 \ Q \ ys \ xs \ ]\!]$
$\Longrightarrow xs = ys$
⟨*proof*⟩

lemma *take-equalityI*: $(\forall i. \ take \ i \ xs = take \ i \ ys) \text{ ==> } xs = ys$
— 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 [*simp*] = *take-Cons'*[*of number-of v*,*standard*]
drop-Cons'[*of number-of v*,*standard*]
nth-Cons'[*of - - number-of v*,*standard*]

38.1.17 *distinct* and *remdups*

lemma *distinct-append* [*simp*]:
$distinct\ (xs\ @\ ys) = (distinct\ xs \wedge distinct\ ys \wedge set\ xs \cap set\ ys = \{\})$
⟨*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 *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-filter* [*simp*]: $distinct\ xs ==> distinct\ (filter\ P\ xs)$
⟨*proof*⟩

lemma *distinct-map-filterI*:
$distinct(map\ f\ xs) \Longrightarrow distinct(map\ f\ (filter\ P\ xs))$
⟨*proof*⟩

lemma *distinct-upt*[*simp*]: $distinct[i..<j]$

⟨*proof*⟩

lemma *distinct-take*[*simp*]: $\bigwedge i.$ *distinct xs* $\Longrightarrow$ *distinct* (*take i xs*)
⟨*proof*⟩

lemma *distinct-drop*[*simp*]: $\bigwedge i.$ *distinct xs* $\Longrightarrow$ *distinct* (*drop i xs*)
⟨*proof*⟩

lemma *distinct-list-update*:
assumes *d*: *distinct xs* **and** *a*: $a \notin$ *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* --> *xs*!*i* $\neq$ *xs*!*j*)
⟨*proof*⟩

lemma *distinct-card*: *distinct xs* ==> *card* (*set xs*) = *size xs*
⟨*proof*⟩

lemma *card-distinct*: *card* (*set xs*) = *size xs* ==> *distinct xs*
⟨*proof*⟩

lemma *inj-on-setI*: *distinct*(*map f xs*) ==> *inj-on f* (*set xs*)
⟨*proof*⟩

lemma *inj-on-set-conv*:
distinct xs $\Longrightarrow$ *inj-on f* (*set xs*) = *distinct*(*map f xs*)
⟨*proof*⟩

38.1.18 *remove1*

lemma *set-remove1-subset*: *set*(*remove1 x xs*) <= *set xs*
⟨*proof*⟩

lemma *set-remove1-eq* [*simp*]: *distinct xs* ==> *set*(*remove1 x xs*) = *set xs* $-$ $\{x\}$
⟨*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*⟩

38.1.19 *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*]: $n \neq 0$ ==> *hd* (*replicate n x*) = *x*
⟨*proof*⟩

lemma *tl-replicate* [*simp*]: $n \neq 0$ ==> *tl* (*replicate n x*) = *replicate* (*n* − *1*) *x*
⟨*proof*⟩

lemma *last-replicate* [*simp*]: $n \neq 0$ ==> *last* (*replicate n x*) = *x*
⟨*proof*⟩

lemma *nth-replicate*[*simp*]: !!*i*. *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*]: !!*i*. *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*]: $n \neq 0$ ==> *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*⟩

38.1.20 *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-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*]: !!*xs*. *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*⟩

38.1.21 *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*:
!!*is*. *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*: !!*I*. *set*(*sublist xs I*) = {*xs*!*i*|*i*. *i*<*size xs* ∧ *i* ∈ *I*}

⟨*proof*⟩

lemma *set-sublist-subset*: $set(sublist\ xs\ I) \subseteq set\ xs$
⟨*proof*⟩

lemma *notin-set-sublistI*[*simp*]: $x \notin set\ xs \Longrightarrow x \notin set(sublist\ xs\ I)$
⟨*proof*⟩

lemma *in-set-sublistD*: $x \in set(sublist\ xs\ I) \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *sublist-singleton* [*simp*]: $sublist\ [x]\ A = (if\ 0 : A\ then\ [x]\ else\ [])$
⟨*proof*⟩

lemma *distinct-sublistI*[*simp*]: $!!I.\ distinct\ xs \Longrightarrow distinct(sublist\ xs\ I)$
⟨*proof*⟩

lemma *sublist-upt-eq-take* [*simp*]: $sublist\ l\ \{..<n\} = take\ n\ l$
⟨*proof*⟩

lemma *filter-in-sublist*: $\bigwedge s.\ distinct\ xs \Longrightarrow$
$filter\ (\%x.\ x \in set(sublist\ xs\ s))\ xs = sublist\ xs\ s$
⟨*proof*⟩

38.1.22 Sets of Lists

38.1.23 *lists*: the list-forming operator over sets

consts *lists* :: *'a set => 'a list set*
inductive *lists A*
intros
Nil [*intro*!]: []: *lists A*
Cons [*intro*!]: [| *a*: *A*;*l*: *lists A*|] ==> *a*#*l* : *lists A*

inductive-cases *listsE* [*elim*!]: *x*#*l* : *lists A*

lemma *lists-mono* [*mono*]: $A \subseteq B$ ==> $lists\ A \subseteq lists\ B$
⟨*proof*⟩

lemma *lists-IntI*:
assumes *l*: *l*: *lists A* **shows** *l*: *lists B* ==> *l*: *lists* (*A Int B*) ⟨*proof*⟩

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

lemma *append-in-lists-conv* [*iff*]:
$(xs\ @\ ys : lists\ A) = (xs : lists\ A \wedge ys : lists\ A)$
⟨*proof*⟩

lemma *in-lists-conv-set*: $(xs : lists\ A) = (\forall x \in set\ xs.\ x : A)$
— eliminate *lists* in favour of *set*
⟨*proof*⟩

lemma *in-listsD* [*dest*!]: $xs \in lists\ A$ ==> $\forall x{\in}set\ xs.\ x \in A$
⟨*proof*⟩

lemma *in-listsI* [*intro*!]: $\forall x{\in}set\ xs.\ x \in A$ ==> $xs \in lists\ A$
⟨*proof*⟩

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

38.1.24 For efficiency

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.

The functions *itrev*, *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

lemma *mem-iff*: $(x\ mem\ xs) = (x : set\ xs)$
⟨*proof*⟩

lemma *list-inter-conv*: $set(list\text{-}inter\ xs\ ys) = set\ xs \cap set\ ys$
⟨*proof*⟩

lemma *list-all-iff*: $list\text{-}all\ P\ xs = (\forall x \in set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *list-all-append* [*simp*]:
$list\text{-}all\ P\ (xs\ @\ ys) = (list\text{-}all\ P\ xs \wedge list\text{-}all\ P\ ys)$
⟨*proof*⟩

lemma *list-all-rev* [*simp*]: $list\text{-}all\ P\ (rev\ xs) = list\text{-}all\ P\ xs$
⟨*proof*⟩

lemma *list-ex-iff*: $list\text{-}ex\ P\ xs = (\exists x \in set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *itrev*[*simp*]: $ALL\ ys.\ itrev\ xs\ ys = rev\ xs\ @\ ys$
⟨*proof*⟩

lemma *filtermap-conv*:
$filtermap\ f\ xs = map\ (\%x.\ the(f\ x))\ (filter\ (\%x.\ f\ x \neq None)\ xs)$

⟨*proof*⟩

lemma *map-filter-conv*[*simp*]: *map-filter f P xs* = *map f* (*filter P xs*)
⟨*proof*⟩

38.1.25 Code generation

Defaults for generating efficient code for some standard functions.

lemmas *in-set-code*[*code unfold*] = *mem-iff* [*symmetric*, *THEN eq-reflection*]

lemma *rev-code*[*code unfold*]: *rev xs* == *itrev xs* []
⟨*proof*⟩

lemma *distinct-Cons-mem*[*code*]: *distinct* (*x*#*xs*) = (~(*x mem xs*) ∧ *distinct xs*)
⟨*proof*⟩

lemma *remdups-Cons-mem*[*code*]:
remdups (*x*#*xs*) = (*if x mem xs then remdups xs else x* # *remdups xs*)
⟨*proof*⟩

lemma *list-inter-Cons-mem*[*code*]: *list-inter* (*a*#*as*) *bs* =
(*if a mem bs then a*#(*list-inter as bs*) *else list-inter as bs*)
⟨*proof*⟩

For implementing bounded quantifiers over lists by *list-ex*/*list-all*:

lemmas *list-bex-code*[*code unfold*] = *list-ex-iff* [*symmetric*, *THEN eq-reflection*]
lemmas *list-ball-code*[*code unfold*] = *list-all-iff* [*symmetric*, *THEN eq-reflection*]

38.1.26 Inductive definition for membership

consts *ListMem* :: ('*a* × '*a list*)*set*
inductive *ListMem*
intros
elem: (*x*,*x*#*xs*) ∈ *ListMem*
insert: (*x*,*xs*) ∈ *ListMem* ⟹ (*x*,*y*#*xs*) ∈ *ListMem*

lemma *ListMem-iff*: ((*x*,*xs*) ∈ *ListMem*) = (*x* ∈ *set xs*)
⟨*proof*⟩

38.1.27 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*)

38.2 Relations on Lists

38.2.1 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*. (*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*:
$lex\ r =$
$\{(xs,ys).\ length\ xs = length\ ys \wedge$
$(\exists xys\ x\ y\ xs'\ ys'.\ xs = xys\ @\ x\ \#\ xs' \wedge ys = xys\ @\ y\ \#\ ys' \wedge (x,\ y){:}r)\}$
$\langle proof \rangle$

lemma *wf-lenlex* [*intro*!]: $wf\ r \Longrightarrow wf\ (lenlex\ r)$
$\langle proof \rangle$

lemma *lenlex-conv*:
$lenlex\ r = \{(xs,ys).\ length\ xs < length\ ys\ |$
$length\ xs = length\ ys \wedge (xs,\ ys) : lex\ r\}$
$\langle proof \rangle$

lemma *Nil-notin-lex* [*iff*]: $([],\ ys) \notin lex\ r$
$\langle proof \rangle$

lemma *Nil2-notin-lex* [*iff*]: $(xs,\ []) \notin lex\ r$
$\langle proof \rangle$

lemma *Cons-in-lex* [*iff*]:
$((x\ \#\ xs,\ y\ \#\ ys) : lex\ r) =$
$((x,\ y) : r \wedge length\ xs = length\ ys\ |\ x = y \wedge (xs,\ ys) : lex\ r)$
$\langle proof \rangle$

38.2.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" ¡ "ab" ¡ "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

constdefs
$lexord :: ('a * 'a)set \Rightarrow ('a\ list * 'a\ list)\ set$
$lexord\ \ r == \{(x,y).\ \exists\ a\ v.\ y = x\ @\ a\ \#\ v \vee$
$(\exists\ u\ a\ b\ v\ w.\ (a,b) \in r \wedge x = u\ @\ (a\ \#\ v) \wedge y = u\ @\ (b\ \#\ w))\}$

lemma *lexord-Nil-left*[*simp*]: $([],y) \in lexord\ r = (\exists\ a\ x.\ y = a\ \#\ x)$
$\langle proof \rangle$

lemma *lexord-Nil-right*[*simp*]: $(x,[]) \notin lexord\ r$
$\langle proof \rangle$

lemma *lexord-cons-cons*[*simp*]:
$((a\ \#\ x,\ b\ \#\ y) \in lexord\ r) = ((a,b)\in r\ |\ (a = b\ \&\ (x,y)\in lexord\ r))$
$\langle proof \rangle$

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-append-rightI*: $\exists\ b\ z.\ y = b\ \#\ z \Longrightarrow (x,\ x\ @\ y) \in lexord\ r$
$\langle proof \rangle$

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*:
$[\![(x, y) \in \mathit{lexord}\ r; (y, z) \in \mathit{lexord}\ r; \mathit{trans}\ r]\!] \Longrightarrow (x, z) \in \mathit{lexord}\ r$
⟨*proof*⟩

lemma *lexord-transI*: $\mathit{trans}\ r \Longrightarrow \mathit{trans}\ (\mathit{lexord}\ r)$
⟨*proof*⟩

lemma *lexord-linear*: $(!\ a\ b.\ (a,b)\in r \mid a = b \mid (b,a) \in r) \Longrightarrow (x,y) : \mathit{lexord}\ r \mid x = y \mid (y,x) : \mathit{lexord}\ r$
⟨*proof*⟩

38.2.3 Lifting a Relation on List Elements to the Lists

consts *listrel* :: $('a * 'a)\mathit{set} => ('a\ \mathit{list} * 'a\ \mathit{list})\mathit{set}$

inductive *listrel*(r)
intros
Nil: $([],[]) \in \mathit{listrel}\ r$
Cons: $[|\ (x,y) \in r; (xs,ys) \in \mathit{listrel}\ r\ |] ==> (x\#xs, y\#ys) \in \mathit{listrel}\ r$

inductive-cases *listrel-Nil1* [*elim*!]: $([],xs) \in \mathit{listrel}\ r$
inductive-cases *listrel-Nil2* [*elim*!]: $(xs,[]) \in \mathit{listrel}\ r$
inductive-cases *listrel-Cons1* [*elim*!]: $(y\#ys,xs) \in \mathit{listrel}\ r$
inductive-cases *listrel-Cons2* [*elim*!]: $(xs,y\#ys) \in \mathit{listrel}\ r$

lemma *listrel-mono*: $r \subseteq s \Longrightarrow \mathit{listrel}\ r \subseteq \mathit{listrel}\ s$

⟨*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*:
$listrel\ r\ ``\ \{x\#xs\} = set\text{-}Cons\ (r``\{x\})\ (listrel\ r\ ``\ \{xs\})$
⟨*proof*⟩

38.3 Miscellany

38.3.1 Characters and strings

datatype *nibble* =
Nibble0 | *Nibble1* | *Nibble2* | *Nibble3* | *Nibble4* | *Nibble5* | *Nibble6* | *Nibble7*
| *Nibble8* | *Nibble9* | *NibbleA* | *NibbleB* | *NibbleC* | *NibbleD* | *NibbleE* | *NibbleF*

datatype *char* = *Char nibble nibble*
— Note: canonical order of character encoding coincides with standard term ordering

types *string* = *char list*

syntax
-*Char* :: *xstr* => *char* (*CHR* -)
-*String* :: *xstr* => *string* (-)

⟨*ML*⟩

38.3.2 Code generator setup

⟨*ML*⟩

types-code
list (- *list*)

```
attach (term-of) ⟪
val term-of-list = HOLogic.mk-list;
⟫
attach (test) ⟪
fun gen-list' aG i j = frequency
  [(i, fn () => aG j :: gen-list' aG (i−1) j), (1, fn () => [])] ()
and gen-list aG i = gen-list' aG i i;
⟫
  char (string)
attach (term-of) ⟪
val nibbleT = Type (List.nibble, []);

fun term-of-char c =
  Const (List.char.Char, nibbleT −−> nibbleT −−> Type (List.char, [])) $
    Const (List.nibble.Nibble ^ nibble-of-int (ord c div 16), nibbleT) $
    Const (List.nibble.Nibble ^ nibble-of-int (ord c mod 16), nibbleT);
⟫
attach (test) ⟪
fun gen-char i = chr (random-range (ord a) (Int.min (ord a + i, ord z)));
⟫

consts-code Cons ((- ::/ -))

⟨ML⟩

end
```

39 Map: Maps

```
theory Map
imports List
begin

types ('a,'b) ~=> = 'a => 'b option (infixr 0)
translations (type) a ~=> b <= (type) a => b option

consts
chg-map :: ('b => 'b) => 'a => ('a ~=> 'b) => ('a ~=> 'b)
map-add :: ('a ~=> 'b) => ('a ~=> 'b) => ('a ~=> 'b) (infixl ++ 100)
restrict-map :: ('a ~=> 'b) => 'a set => ('a ~=> 'b) (infixl |` 110)
dom     :: ('a ~=> 'b) => 'a set
ran     :: ('a ~=> 'b) => 'b set
map-of  :: ('a ∗ 'b)list => 'a ~=> 'b
map-upds:: ('a ~=> 'b) => 'a list => 'b list =>
          ('a ~=> 'b)
map-upd-s::('a ~=> 'b) => 'a set => 'b =>
          ('a ~=> 'b)                    (-/'(-{|−>}-/') [900,0,0]900)
map-subst::('a ~=> 'b) => 'b => 'b =>
```

```
           ('a ~=> 'b)                        (-/'(-~>-/')    [900,0,0]900)
map-le :: ('a ~=> 'b) => ('a ~=> 'b) => bool (infix ⊆m 50)
```

constdefs

```
  map-comp :: ('b ~=> 'c) => ('a ~=> 'b) => ('a ~=> 'c) (infixl o'-m 55)
  f o-m g == (λk. case g k of None ⇒ None | Some v ⇒ f v)
```

nonterminals

```
  maplets maplet
```

syntax

```
  empty     :: 'a ~=> 'b
  -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*)

```
  ~=>      :: [type, type] => type    (infixr ⇀ 0)

  map-comp :: ('b ~=> 'c) => ('a ~=> 'b) => ('a ~=> 'c) (infixl ∘m 55)

  -maplet  :: ['a, 'a] => maplet             (- /↦/ -)
  -maplets :: ['a, 'a] => maplet             (- /[↦]/ -)

  map-upd-s  :: ('a ~=> 'b) => 'a set => 'b => ('a ~=> 'b)
                                            (-/'(-/{↦}/-') [900,0,0]900)
  map-subst :: ('a ~=> 'b) => 'b => 'b =>
               ('a ~=> 'b)                  (-/'(⤳-/')    [900,0,0]900)
 @chg-map :: ('a ~=> 'b) => 'a => ('b => 'b) => ('a ~=> 'b)
                                    (-/'(-/↦λ-. -')  [900,0,0,0] 900)
```

syntax (*latex* **output**)

```
  restrict-map :: ('a ~=> 'b) => 'a set => ('a ~=> 'b) (-↾- [111,110] 110)
  — requires amssymb!
```

translations

```
  empty    => -K None
  empty    <= %x. None

  m(x↦λy. f) == chg-map (λy. f) x m

  -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 empty ms
  -Map (-Maplets ms1 ms2)      <= -MapUpd (-Map ms1) ms2
```

-Maplets ms1 (-Maplets ms2 ms3) <= -Maplets (-Maplets ms1 ms2) ms3

defs
chg-map-def: *chg-map f a m == case m a of None => m | Some b => m(a|->f b)*

map-add-def: *m1++m2 == %x. case m2 x of None => m1 x | Some y => Some y*
restrict-map-def: *m|'A == %x. if x : A then m x else None*

map-upds-def: *m(xs [|->] ys) == m ++ map-of (rev(zip xs ys))*
map-upd-s-def: *m(as{|->}b) == %x. if x : as then Some b else m x*
map-subst-def: *m(a~>b) == %x. if m x = Some a then Some b else m x*

dom-def: *dom(m) == {a. m a ~= None}*
ran-def: *ran(m) == {b. EX a. m a = Some b}*

map-le-def: $m_1 \subseteq_m m_2$ == *ALL a : dom* m_1. m_1 *a* = m_2 *a*

primrec
map-of [] = empty
map-of (p#ps) = (map-of ps)(fst p |-> snd p)

39.1 *empty*

lemma *empty-upd-none[simp]*: *empty(x := None) = empty*
⟨*proof*⟩

lemma *sum-case-empty-empty[simp]*: *sum-case empty empty = empty*
⟨*proof*⟩

39.2 *map-upd*

lemma *map-upd-triv*: *t k = Some x ==> t(k|->x) = t*
⟨*proof*⟩

lemma *map-upd-nonempty[simp]*: *t(k|->x) ~= empty*
⟨*proof*⟩

lemma *map-upd-eqD1*: $m(a \mapsto x) = n(a \mapsto y) \Longrightarrow x = y$
⟨*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) ==> *finite* (*range* ($f(a|->b)$))
⟨*proof*⟩

39.3 *sum-case* and *empty*/*map-upd*

lemma *sum-case-map-upd-empty*[*simp*]:
sum-case ($m(k|->y)$) *empty* = (*sum-case m empty*)(*Inl* $k|->y$)
⟨*proof*⟩

lemma *sum-case-empty-map-upd*[*simp*]:
sum-case empty ($m(k|->y)$) = (*sum-case empty m*)(*Inr* $k|->y$)
⟨*proof*⟩

lemma *sum-case-map-upd-map-upd*[*simp*]:
sum-case ($m1(k1|->y1)$) ($m2(k2|->y2)$) = (*sum-case* ($m1(k1|->y1)$) $m2$)(*Inr* $k2|->y2$)
⟨*proof*⟩

39.4 *chg-map*

lemma *chg-map-new*[*simp*]: $m\ a$ = *None* ==> *chg-map* $f\ a\ m = m$
⟨*proof*⟩

lemma *chg-map-upd*[*simp*]: $m\ a$ = *Some* b ==> *chg-map* $f\ a\ m = m(a|->f\ b)$
⟨*proof*⟩

lemma *chg-map-other* [*simp*]: $a \neq b \Longrightarrow$ *chg-map* $f\ a\ m\ b = m\ b$
⟨*proof*⟩

39.5 *map-of*

lemma *map-of-eq-None-iff*:
(*map-of* $xys\ x$ = *None*) = ($x \notin$ *fst* ‘ (*set* xys))
⟨*proof*⟩

lemma *map-of-is-SomeD*:
map-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-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-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-of* $xys\ x$ = *Some* y

⟨*proof*⟩

lemma *map-of-zip-is-None*[*simp*]:
$length\ xs = length\ ys \Longrightarrow (map\text{-}of\ (zip\ xs\ ys)\ x = None) = (x \notin set\ xs)$
⟨*proof*⟩

lemma *finite-range-map-of*: $finite\ (range\ (map\text{-}of\ xys))$
⟨*proof*⟩

lemma *map-of-SomeD* [*rule-format*]: $map\text{-}of\ xs\ k = Some\ y\ {-}{-}{>}\ (k,y){:}set\ xs$
⟨*proof*⟩

lemma *map-of-mapk-SomeI* [*rule-format*]:
$inj\ f\ {=}{=}{>}\ map\text{-}of\ t\ k = Some\ x\ {-}{-}{>}$
$map\text{-}of\ (map\ (split\ (\%k.\ Pair\ (f\ k)))\ t)\ (f\ k) = Some\ x$
⟨*proof*⟩

lemma *weak-map-of-SomeI* [*rule-format*]:
$(k,\ x) : set\ l\ {-}{-}{>}\ (\exists\, x.\ map\text{-}of\ l\ k = Some\ x)$
⟨*proof*⟩

lemma *map-of-filter-in*:
$[|\ map\text{-}of\ xs\ k = Some\ z;\ P\ k\ z\ |]\ {=}{=}{>}\ map\text{-}of\ (filter\ (split\ P)\ xs)\ k = Some\ z$
⟨*proof*⟩

lemma *map-of-map*: $map\text{-}of\ (map\ (\%(a,b).\ (a,f\ b))\ xs)\ x = option\text{-}map\ f\ (map\text{-}of\ xs\ x)$
⟨*proof*⟩

39.6 *option-map* related

lemma *option-map-o-empty*[*simp*]: $option\text{-}map\ f\ o\ empty = empty$
⟨*proof*⟩

lemma *option-map-o-map-upd*[*simp*]:
$option\text{-}map\ f\ o\ m(a|{-}{>}b) = (option\text{-}map\ f\ o\ m)(a|{-}{>}f\ b)$
⟨*proof*⟩

39.7 *map-comp* related

lemma *map-comp-empty* [*simp*]:
$m \circ_m empty = empty$
$empty \circ_m m = empty$
⟨*proof*⟩

lemma *map-comp-simps* [*simp*]:
$m2\ k = None \Longrightarrow (m1 \circ_m m2)\ k = None$
$m2\ k = Some\ k' \Longrightarrow (m1 \circ_m m2)\ k = m1\ k'$
⟨*proof*⟩

lemma *map-comp-Some-iff*:
$((m1 \circ_m m2)\ k = Some\ v) = (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = Some\ v)$
⟨*proof*⟩

lemma *map-comp-None-iff*:
$((m1 \circ_m m2)\ k = None) = (m2\ k = None \vee (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = None))$
⟨*proof*⟩

39.8 ++

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*⟩

lemmas *map-add-SomeD* = *map-add-Some-iff* [*THEN iffD1*, *standard*]
declare *map-add-SomeD* [*dest*!]

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*⟩

declare *fun-upd-apply* [*simp del*]
lemma *finite-range-map-of-map-add*:
$finite\ (range\ f)\ {=}{=}{>}\ finite\ (range\ (f\ {+}{+}\ map\text{-}of\ l))$
⟨*proof*⟩
declare *fun-upd-apply* [*simp*]

lemma *inj-on-map-add-dom*[*iff*]:
$inj\text{-}on\ (m\ ++\ m')\ (dom\ m') = inj\text{-}on\ m'\ (dom\ m')$
⟨*proof*⟩

39.9 *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*]:
$m(x := y)|`D = (if\ x \in D\ then\ (m|`(D{-}\{x\}))(x := y)\ else\ m|`D)$
⟨*proof*⟩

lemma *fun-upd-None-restrict*[*simp*]:
$(m|`D)(x := None) = (if\ x{:}D\ then\ m|`(D - \{x\})\ else\ m|`D)$
⟨*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*⟩

39.10 *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*]: $\bigwedge ys\ m.\ size\ xs < size\ ys \Longrightarrow$
$m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys)(x \mapsto ys!size\ xs)$
⟨*proof*⟩

lemma *map-upds-list-update2-drop*[*simp*]:
$\bigwedge m\ ys\ i.\ [\![size\ xs \le i;\ i < size\ ys]\!]$
$\Longrightarrow m(xs[\mapsto]ys[i{:=}y]) = m(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *map-upd-upds-conv-if*: $!!x\ y\ ys\ f.$
$(f(x|->y))(xs\ [|->]\ ys) =$
$(if\ x : set(take\ (length\ ys)\ xs)\ then\ f(xs\ [|->]\ ys)$
$else\ (f(xs\ [|->]\ ys))(x|->y))$
⟨*proof*⟩

lemma *map-upds-twist* [*simp*]:
$a\ \tilde{}:\ set\ as\ ==>\ m(a|->b)(as[|->]bs) = m(as[|->]bs)(a|->b)$
⟨*proof*⟩

lemma *map-upds-apply-nontin*[*simp*]:
$!!ys.\ x\ \tilde{}:\ set\ xs\ ==>\ (f(xs[|->]ys))\ x = f\ x$
⟨*proof*⟩

lemma *fun-upds-append-drop*[*simp*]:
$!!m\ ys.\ size\ xs = size\ ys \Longrightarrow m(xs@zs[\mapsto]ys) = m(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *fun-upds-append2-drop*[*simp*]:
$!!m\ ys.\ size\ xs = size\ ys \Longrightarrow m(xs[\mapsto]ys@zs) = m(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *restrict-map-upds*[*simp*]: $!!m\ ys.$
$[\![\ length\ xs = length\ ys;\ set\ xs \subseteq D\ ]\!]$
$\Longrightarrow m(xs\ [\mapsto]\ ys)|`D = (m|`(D - set\ xs))(xs\ [\mapsto]\ ys)$
⟨*proof*⟩

39.11 *map-upd-s*

lemma *map-upd-s-apply* [*simp*]:
$(m(as\{|->\}b))\ x = (if\ x : as\ then\ Some\ b\ else\ m\ x)$

⟨*proof*⟩

lemma *map-subst-apply* [*simp*]:
$(m(a{\sim}{>}b))\ x = (\mathit{if}\ m\ x = \mathit{Some}\ a\ \mathit{then}\ \mathit{Some}\ b\ \mathit{else}\ m\ x)$
⟨*proof*⟩

39.12 *dom*

lemma *domI*: $m\ a = \mathit{Some}\ b \Longrightarrow a : \mathit{dom}\ m$
⟨*proof*⟩

lemma *domD*: $a : \mathit{dom}\ m \Longrightarrow \exists\, b.\ m\ a = \mathit{Some}\ b$
⟨*proof*⟩

lemma *domIff* [*iff*]: $(a : \mathit{dom}\ m) = (m\ a\ {\sim}{=}\ \mathit{None})$
⟨*proof*⟩
declare *domIff* [*simp del*]

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

lemma *dom-fun-upd*[*simp*]:
$\mathit{dom}(f(x := y)) = (\mathit{if}\ y{=}\mathit{None}\ \mathit{then}\ \mathit{dom}\ f - \{x\}\ \mathit{else}\ \mathit{insert}\ x\ (\mathit{dom}\ f))$
⟨*proof*⟩

lemma *dom-map-of*: $\mathit{dom}(\mathit{map\text{-}of}\ \mathit{xys}) = \{x.\ \exists\, y.\ (x,y) : \mathit{set}\ \mathit{xys}\}$
⟨*proof*⟩

lemma *dom-map-of-conv-image-fst*:
$\mathit{dom}(\mathit{map\text{-}of}\ \mathit{xys}) = \mathit{fst}\ `\ (\mathit{set}\ \mathit{xys})$
⟨*proof*⟩

lemma *dom-map-of-zip*[*simp*]: $[|\ \mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys};\ \mathit{distinct}\ \mathit{xs}\ |] \Longrightarrow$
$\mathit{dom}(\mathit{map\text{-}of}(\mathit{zip}\ \mathit{xs}\ \mathit{ys})) = \mathit{set}\ \mathit{xs}$
⟨*proof*⟩

lemma *finite-dom-map-of*: $\mathit{finite}\ (\mathit{dom}\ (\mathit{map\text{-}of}\ l))$
⟨*proof*⟩

lemma *dom-map-upds*[*simp*]:
$!!m\ \mathit{ys}.\ \mathit{dom}(m(\mathit{xs}[|{-}{>}]\mathit{ys})) = \mathit{set}(\mathit{take}\ (\mathit{length}\ \mathit{ys})\ \mathit{xs})\ \mathit{Un}\ \mathit{dom}\ m$
⟨*proof*⟩

lemma *dom-map-add*[*simp*]: $\mathit{dom}(m{+}{+}n) = \mathit{dom}\ n\ \mathit{Un}\ \mathit{dom}\ m$
⟨*proof*⟩

lemma *dom-override-on*[*simp*]:
$\mathit{dom}(\mathit{override\text{-}on}\ f\ g\ A) =$

$(dom\ f\ -\ \{a.\ a\ :\ A\ -\ dom\ g\})\ Un\ \{a.\ a\ :\ A\ Int\ dom\ g\}$
⟨*proof*⟩

lemma *map-add-comm*: $dom\ m1 \cap dom\ m2 = \{\} \Longrightarrow m1{+}{+}m2 = m2{+}{+}m1$
⟨*proof*⟩

39.13 *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*⟩

39.14 *map-le*

lemma *map-le-empty* [*simp*]: $empty \subseteq_m g$
⟨*proof*⟩

lemma *upd-None-map-le* [*simp*]: $f(x := None) \subseteq_m f$
⟨*proof*⟩

lemma *map-le-upd*[*simp*]: $f \subseteq_m g ==> f(a := b) \subseteq_m g(a := b)$
⟨*proof*⟩

lemma *map-le-imp-upd-le* [*simp*]: $m1 \subseteq_m m2 \Longrightarrow m1(x := None) \subseteq_m m2(x \mapsto y)$
⟨*proof*⟩

lemma *map-le-upds*[*simp*]:
$!!f\ g\ bs.\ f \subseteq_m g ==> f(as\ [|->]\ bs) \subseteq_m g(as\ [|->]\ bs)$
⟨*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

40 Refute: Refute

theory *Refute*
imports *Map*
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.  Defining equations for constants are added     *)
(* automatically, as are sort axioms.  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

41 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

42 Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *NatArith*
uses (*Tools/meson.ML*) (*Tools/specification-package.ML*)
begin

42.1 Hilbert's epsilon

consts
  Eps :: ('a => bool) => 'a

syntax (*epsilon*)
  -Eps :: [*pttrn*, *bool*] => 'a ((*3*ϵ -./ -) [*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 == *Eps* (%*x. P*)

⟨*ML*⟩

axioms
  someI: *P* (*x*::'*a*) ==> *P* (*SOME x. P x*)
finalconsts
  Eps

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*

42.2 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*: $P\ (SOME\ x.\ P\ x) = (\exists x.\ P\ x)$
⟨proof⟩

lemma *some-eq-trivial* [*simp*]: $(SOME\ y.\ y{=}x) = x$
⟨proof⟩

lemma *some-sym-eq-trivial* [*simp*]: $(SOME\ y.\ x{=}y) = x$
⟨proof⟩

42.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

lemma *choice*: $\forall x.\ \exists y.\ Q\ x\ y \implies \exists f.\ \forall x.\ Q\ x\ (f\ x)$
⟨proof⟩

lemma *bchoice*: $\forall x{\in}S.\ \exists y.\ Q\ x\ y \implies \exists f.\ \forall x{\in}S.\ Q\ x\ (f\ x)$
⟨proof⟩

42.4 Function Inverse

lemma *inv-id* [*simp*]: $inv\ id = id$
⟨proof⟩

A one-to-one function has an inverse.

lemma *inv-f-f* [*simp*]: $inj\ f \implies inv\ f\ (f\ x) = x$
⟨proof⟩

lemma *inv-f-eq*: $[|\ inj\ f;\ \ f\ x = y\ |] \implies inv\ f\ y = x$
⟨proof⟩

lemma *inj-imp-inv-eq*: $[|\ inj\ f;\ \forall x.\ f(g\ x) = x\ |] \implies inv\ f = g$
⟨proof⟩

But is it useful?

lemma *inj-transfer*:
  assumes *injf*: $inj\ f$ **and** *minor*: $!!y.\ y \in range(f) \implies P(inv\ f\ y)$
  shows $P\ x$
⟨proof⟩

lemma *inj-iff*: $(inj\ f) = (inv\ f\ o\ f = id)$
⟨proof⟩

lemma *inj-imp-surj-inv*: $inj\ f \implies surj\ (inv\ f)$
⟨proof⟩

lemma *f-inv-f*: $y \in range(f) ==> f(inv\ f\ y) = y$
⟨*proof*⟩

lemma *surj-f-inv-f*: $surj\ f ==> f(inv\ f\ y) = y$
⟨*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*: $A <= range(f) ==> inj\text{-}on\ (inv\ f)\ A$
⟨*proof*⟩

lemma *surj-imp-inj-inv*: $surj\ f ==> inj\ (inv\ f)$
⟨*proof*⟩

lemma *surj-iff*: $(surj\ f) = (f\ o\ inv\ f = id)$
⟨*proof*⟩

lemma *surj-imp-inv-eq*: $[|\ surj\ f; \forall x.\ g(f\ x) = x\ |] ==> inv\ f = g$
⟨*proof*⟩

lemma *bij-imp-bij-inv*: $bij\ f ==> bij\ (inv\ f)$
⟨*proof*⟩

lemma *inv-equality*: $[|\ !!x.\ g\ (f\ x) = x;\ \ !!y.\ f\ (g\ y) = y\ |] ==> inv\ f = g$
⟨*proof*⟩

lemma *inv-inv-eq*: $bij\ f ==> inv\ (inv\ f) = f$
⟨*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*⟩

42.5 Inverse of a PI-function (restricted domain)

lemma *Inv-f-f*: [| *inj-on f A*; $x \in A$ |] $==>$ *Inv A f* $(f\ x) = x$
⟨*proof*⟩

lemma *f-Inv-f*: $y \in f$'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 ' $A = B$; $x \in B$ |] $==>$ *Inv A f x* $\in A$
⟨*proof*⟩

lemma *Inv-f-eq*: [| *inj-on f A*; $f\ x = y$; $x \in A$ |] $==>$ *Inv A f y* $= x$
  ⟨*proof*⟩

lemma *Inv-comp*:
  [| *inj-on f* (g ' A); *inj-on g A*; $x \in f$ ' g ' A |] $==>$
  Inv A (f *o* g) x = (*Inv A g o Inv* (g ' A) f) x
  ⟨*proof*⟩

42.6 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*:
$wf\ r = ({\sim}(\exists f.\ \forall i.\ (f(Suc\ i), f\ i) \in r))$
$\langle proof \rangle$

A dynamically-scoped fact for TFL

lemma *tfl-some*: $\forall P\ x.\ P\ x\ {-}{-}{>}\ P\ (Eps\ P)$
$\langle proof \rangle$

42.7 Least value operator

constdefs
$LeastM :: [{'a} => {'b}::ord,\ {'a} => bool] => {'a}$
$LeastM\ m\ P == SOME\ x.\ P\ x\ \&\ (\forall y.\ P\ y\ {-}{-}{>}\ m\ x <= m\ y)$

syntax
$\text{-}LeastM :: [pttrn,\ {'a} => {'b}::ord,\ bool] => {'a}$ $\quad (LEAST\ \text{-}\ WRT\ \text{-}.\ \text{-}\ [0,\ 4,\ 10]\ 10)$
translations
$LEAST\ x\ WRT\ m.\ P == LeastM\ m\ (\%x.\ P)$

lemma *LeastMI2*:
$P\ x ==> (!!y.\ P\ y ==> m\ x <= m\ y)$
$==> (!!x.\ P\ x ==> \forall y.\ P\ y\ {-}{-}{>}\ m\ x \le m\ y ==> Q\ x)$
$==> Q\ (LeastM\ m\ P)$
$\langle proof \rangle$

lemma *LeastM-equality*:
$P\ k ==> (!!x.\ P\ x ==> m\ k <= m\ x)$
$==> m\ (LEAST\ x\ WRT\ m.\ P\ x) = (m\ k::{'a}::order)$
$\langle proof \rangle$

lemma *wf-linord-ex-has-least*:
$wf\ r ==> \forall x\ y.\ ((x,y):r\hat{}{+}) = ((y,x){\sim}:r\hat{}{*}) ==> P\ k$
$==> \exists x.\ P\ x\ \&\ (!y.\ P\ y\ {-}{-}{>}\ (m\ x, m\ y):r\hat{}{*})$
$\langle proof \rangle$

lemma *ex-has-least-nat*:
$P\ k ==> \exists x.\ P\ x\ \&\ (\forall y.\ P\ y\ {-}{-}{>}\ m\ x <= (m\ y::nat))$
$\langle proof \rangle$

lemma *LeastM-nat-lemma*:
$P\ k ==> P\ (LeastM\ m\ P)\ \&\ (\forall y.\ P\ y\ {-}{-}{>}\ m\ (LeastM\ m\ P) <= (m\ y::nat))$
$\langle proof \rangle$

lemmas *LeastM-natI* = *LeastM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *LeastM-nat-le*: $P\ x ==> m\ (LeastM\ m\ P) <= (m\ x::nat)$
$\langle proof \rangle$

42.8 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 ==> ∀ *x. P x* --> (∃ *y. P y* & ~ ((*m y*::*nat*) <= *m x*))
  ==> ∃ *y. P y* & ~ (*m y* < *m k* + *n*)
⟨*proof*⟩

lemma *ex-has-greatest-nat*:
P k ==> ∀ *y. P y* --> *m y* < *b*
  ==> ∃ *x. P x* & (∀ *y. P y* --> (*m y*::*nat*) <= *m x*)
⟨*proof*⟩

lemma *GreatestM-nat-lemma*:
P k ==> ∀ *y. P y* --> *m y* < *b*
  ==> *P* (*GreatestM m P*) & (∀ *y. P y* --> (*m y*::*nat*) <= *m* (*GreatestM m P*))
⟨*proof*⟩

lemmas *GreatestM-natI* = *GreatestM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *GreatestM-nat-le*:

$P\ x\ ==>\ \forall y.\ P\ y\ -->\ m\ y < b$
$==>\ (m\ x::nat) <= m\ (GreatestM\ m\ P)$
⟨*proof*⟩

Specialization to *GREATEST*.

lemma *GreatestI*: $P\ (k::nat)\ ==>\ \forall y.\ P\ y\ -->\ y < b\ ==>\ P\ (GREATEST\ x.\ P\ x)$
⟨*proof*⟩

lemma *Greatest-le*:
$P\ x\ ==>\ \forall y.\ P\ y\ -->\ y < b\ ==>\ (x::nat) <= (GREATEST\ x.\ P\ x)$
⟨*proof*⟩

42.9 The Meson proof procedure

42.9.1 Negation Normal Form

de Morgan laws

lemma *meson-not-conjD*: $\sim(P\&Q)\ ==>\ \sim P\ |\ \sim Q$
and *meson-not-disjD*: $\sim(P|Q)\ ==>\ \sim P\ \&\ \sim Q$
and *meson-not-notD*: $\sim\sim P\ ==>\ P$
and *meson-not-allD*: $!!P.\ \sim(\forall x.\ P(x))\ ==>\ \exists x.\ \sim P(x)$
and *meson-not-exD*: $!!P.\ \sim(\exists x.\ P(x))\ ==>\ \forall x.\ \sim P(x)$
⟨*proof*⟩

Removal of $-->$ and $<->$ (positive and negative occurrences)

lemma *meson-imp-to-disjD*: $P-->Q\ ==>\ \sim P\ |\ Q$
and *meson-not-impD*: $\sim(P-->Q)\ ==>\ P\ \&\ \sim Q$
and *meson-iff-to-disjD*: $P=Q\ ==>\ (\sim P\ |\ Q)\ \&\ (\sim Q\ |\ P)$
and *meson-not-iffD*: $\sim(P=Q)\ ==>\ (P\ |\ Q)\ \&\ (\sim P\ |\ \sim Q)$
— Much more efficient than $P \wedge \neg Q \vee Q \wedge \neg P$ for computing CNF
⟨*proof*⟩

42.9.2 Pulling out the existential quantifiers

Conjunction

lemma *meson-conj-exD1*: $!!P\ Q.\ (\exists x.\ P(x))\ \&\ Q\ ==>\ \exists x.\ P(x)\ \&\ Q$
and *meson-conj-exD2*: $!!P\ Q.\ P\ \&\ (\exists x.\ Q(x))\ ==>\ \exists x.\ P\ \&\ Q(x)$
⟨*proof*⟩

Disjunction

lemma *meson-disj-exD*: $!!P\ Q.\ (\exists x.\ P(x))\ |\ (\exists x.\ Q(x))\ ==>\ \exists x.\ P(x)\ |\ Q(x)$
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and *meson-disj-exD1*: $!!P\ Q.\ (\exists x.\ P(x))\ |\ Q\ ==>\ \exists x.\ P(x)\ |\ Q$
and *meson-disj-exD2*: $!!P\ Q.\ P\ |\ (\exists x.\ Q(x))\ ==>\ \exists x.\ P\ |\ Q(x)$
⟨*proof*⟩

42.9.3 Generating clauses for the Meson Proof Procedure

Disjunctions

lemma *meson-disj-assoc*: $(P|Q)|R$ ==> $P|(Q|R)$
and *meson-disj-comm*: $P|Q$ ==> $Q|P$
and *meson-disj-FalseD1*: *False*$|P$ ==> P
and *meson-disj-FalseD2*: $P|$*False* ==> P
⟨*proof*⟩

42.10 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*: ~$P|Q$ ==> ((~P==>P) ==> Q)
⟨*proof*⟩

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: ~$P|Q$ ==> (P ==> Q)
⟨*proof*⟩

P should be a literal

lemma *make-pos-rule*: $P|Q$ ==> ((P==>~P) ==> 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*⟩

42.10.1 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*:
$[|\ P'|Q';\ \ P' ==> P;\ \ [|\ Q';\ P{==>}\mathit{False}\ |] ==> Q\ |] ==> P|Q$
⟨*proof*⟩

lemma *all-forward*: $[|\ \forall x.\ P'(x);\ \ !!x.\ P'(x) ==> P(x)\ |] ==> \forall x.\ P(x)$
⟨*proof*⟩

lemma *ex-forward*: $[|\ \exists x.\ P'(x);\ \ !!x.\ P'(x) ==> P(x)\ |] ==> \exists x.\ P(x)$
⟨*proof*⟩

Many of these bindings are used by the ATP linkup, and not just by legacy proof scripts.

⟨*ML*⟩

end

43 Infinite-Set: Infnite Sets and Related Concepts

theory *Infinite-Set*
imports *Hilbert-Choice Binomial*
begin

43.1 Infinite Sets

Some elementary facts about infinite sets, by Stefan Merz.

syntax
infinite :: *'a set* ⇒ *bool*
translations
infinite S == *S* ∉ *Finites*

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-nonempty*:
¬ (*infinite* {})
⟨*proof*⟩

lemma *infinite-remove*:
infinite S ⟹ *infinite* (*S* − {*a*})
⟨*proof*⟩

lemma *Diff-infinite-finite*:

assumes T: *finite* T **and** S: *infinite* S
shows *infinite* $(S - T)$
⟨*proof*⟩

lemma *Un-infinite*:
infinite $S \Longrightarrow$ *infinite* $(S \cup T)$
⟨*proof*⟩

lemma *infinite-super*:
assumes T: $S \subseteq T$ **and** S: *infinite* S
shows *infinite* T
⟨*proof*⟩

As a concrete example, we prove that the set of natural numbers is infinite.

lemma *finite-nat-bounded*:
assumes S: *finite* $(S{::}nat\ set)$
shows $\exists\, k.\ S \subseteq \{..<k\}$ (**is** $\exists\, k.\ ?bounded\ S\ k$)
⟨*proof*⟩

lemma *finite-nat-iff-bounded*:
finite $(S{::}nat\ set) = (\exists\, k.\ S \subseteq \{..<k\})$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*:
finite $(S{::}nat\ set) = (\exists\, k.\ S \subseteq \{..k\})$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*:
infinite $(S{::}nat\ set) = (\forall\, m.\ \exists\, n.\ m<n \wedge n \in S)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded-le*:
infinite $(S{::}nat\ set) = (\forall\, m.\ \exists\, n.\ m \le n \wedge n \in S)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*:
assumes k: $\forall\, m.\ k<m \longrightarrow (\exists\, n.\ m<n \wedge n \in S)$
shows *infinite* $(S{::}nat\ set)$
⟨*proof*⟩

theorem *nat-infinite* [*simp*]:
infinite $(UNIV :: nat\ set)$
⟨*proof*⟩

theorem *nat-not-finite* [*elim*]:
  $finite\ (UNIV::nat\ set) \Longrightarrow R$
⟨*proof*⟩

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

lemma *range-inj-infinite*:
  $inj\ (f::nat \Rightarrow 'a) \Longrightarrow infinite\ (range\ f)$
⟨*proof*⟩

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *linorder-injI*:
  assumes *hyp*: $\forall\, x\ y.\ x < (y::'a::linorder) \longrightarrow f\,x \neq f\,y$
  shows $inj\ f$
⟨*proof*⟩

lemma *infinite-countable-subset*:
  assumes *inf*: $infinite\ (S::'a\ set)$
  shows $\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S$
⟨*proof*⟩

theorem *infinite-iff-countable-subset*:
  $infinite\ S = (\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S)$
  (**is** $?lhs = ?rhs$)
⟨*proof*⟩

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

theorem *inf-img-fin-dom*:
  assumes *img*: $finite\ (f`A)$ **and** *dom*: $infinite\ A$
  shows $\exists\, y \in f`A.\ infinite\ (f\ -`\ \{y\})$
⟨*proof*⟩

theorems *inf-img-fin-domE* = *inf-img-fin-dom*[*THEN bexE*]

43.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

consts

Inf-many :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *INF 10*)
Alm-all :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *MOST 10*)

defs
INF-def: *Inf-many* $P \equiv$ *infinite* $\{x.\ P\ x\}$
MOST-def: *Alm-all* $P \equiv \neg(INF\ x.\ \neg\ P\ x)$

syntax (*xsymbols*)
MOST :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\forall_\infty$-./ -) [*0*,*10*] *10*)
INF :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\exists_\infty$-./ -) [*0*,*10*] *10*)

syntax (*HTML* **output**)
MOST :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\forall_\infty$-./ -) [*0*,*10*] *10*)
INF :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\exists_\infty$-./ -) [*0*,*10*] *10*)

lemma *INF-EX*:
$(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$
⟨*proof*⟩

lemma *MOST-iff-finiteNeg*:
$(\forall_\infty x.\ P\ x) =$ *finite* $\{x.\ \neg\ P\ x\}$
⟨*proof*⟩

lemma *ALL-MOST*:
$\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$
⟨*proof*⟩

lemma *INF-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
⟨*proof*⟩

lemma *MOST-mono*:
$[\![\ \forall_\infty x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x\ ]\!] \Longrightarrow \forall_\infty x.\ Q\ x$
⟨*proof*⟩

lemma *INF-nat*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall m.\ \exists n.\ m<n \wedge P\ n)$
⟨*proof*⟩

lemma *INF-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall m.\ \exists n.\ m\le n \wedge P\ n)$
⟨*proof*⟩

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists m.\ \forall n.\ m<n \longrightarrow P\ n)$
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists m.\ \forall n.\ m\le n \longrightarrow P\ n)$
⟨*proof*⟩

43.3 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

constdefs
  atmost-one :: *'a set* ⇒ *bool*
  atmost-one S ≡ ∀ *x y*. *x*∈*S* ∧ *y*∈*S* ⟶ *x*=*y*

lemma *atmost-one-empty*: *S*={} ⟹ *atmost-one S*
⟨*proof*⟩

lemma *atmost-one-singleton*: *S* = {*x*} ⟹ *atmost-one S*
⟨*proof*⟩

lemma *atmost-one-unique* [*elim*]: ⟦ *atmost-one S*; *x* ∈ *S*; *y* ∈ *S* ⟧ ⟹ *y*=*x*
⟨*proof*⟩

end

44 Extraction: Program extraction for HOL

theory *Extraction*
imports *Datatype*
uses *Tools/rewrite-hol-proof.ML*
begin

44.1 Setup

⟨*ML*⟩

lemmas [*extraction-expand*] =
  atomize-eq atomize-all atomize-imp
  allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
  notE' impE' impE iffE imp-cong simp-thms
  induct-forall-eq induct-implies-eq induct-equal-eq
  induct-forall-def induct-implies-def induct-impliesI
  induct-atomize induct-rulify1 induct-rulify2

datatype *sumbool* = *Left* | *Right*

44.2 Type of extracted program

extract-type
  typeof (*Trueprop P*) ≡ *typeof P*

  typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*(*'Q*)) ⟹
    typeof (*P* ⟶ *Q*) ≡ *Type* (*TYPE*(*'Q*))

$$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE(Null))$$

$$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$$
$$typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('P \Rightarrow 'Q))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$$
$$typeof\ (\forall x.\ P\ x) \equiv Type\ (TYPE(Null))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$$
$$typeof\ (\forall x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$$
$$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$$
$$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$$

$$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$$
$$typeof\ (P \lor Q) \equiv Type\ (TYPE(sumbool))$$

$$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$$
$$typeof\ (P \lor Q) \equiv Type\ (TYPE('Q\ option))$$

$$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$$
$$typeof\ (P \lor Q) \equiv Type\ (TYPE('P\ option))$$

$$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$$
$$typeof\ (P \lor Q) \equiv Type\ (TYPE('P + 'Q))$$

$$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$$
$$typeof\ (P \land Q) \equiv Type\ (TYPE('Q))$$

$$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$$
$$typeof\ (P \land Q) \equiv Type\ (TYPE('P))$$

$$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$$
$$typeof\ (P \land Q) \equiv Type\ (TYPE('P \times 'Q))$$

$$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \land (Q \longrightarrow P))$$

$$typeof\ (x \in P) \equiv typeof\ P$$

44.3 Realizability

realizability

$$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$$

$$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$$
$$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$$

$(\mathit{typeof}\ P) \equiv (\mathit{Type}\ (\mathit{TYPE}('P))) \Longrightarrow$
$\ (\mathit{typeof}\ Q) \equiv (\mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (P \longrightarrow Q)) \equiv (\forall\, x{::}'P.\ \mathit{realizes}\ x\ P \longrightarrow \mathit{realizes}\ \mathit{Null}\ Q)$

$(\mathit{realizes}\ t\ (P \longrightarrow Q)) \equiv (\forall\, x.\ \mathit{realizes}\ x\ P \longrightarrow \mathit{realizes}\ (t\ x)\ Q)$

$(\lambda x.\ \mathit{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (\forall\, x.\ P\ x)) \equiv (\forall\, x.\ \mathit{realizes}\ \mathit{Null}\ (P\ x))$

$(\mathit{realizes}\ t\ (\forall\, x.\ P\ x)) \equiv (\forall\, x.\ \mathit{realizes}\ (t\ x)\ (P\ x))$

$(\lambda x.\ \mathit{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (\exists\, x.\ P\ x)) \equiv (\mathit{realizes}\ \mathit{Null}\ (P\ t))$

$(\mathit{realizes}\ t\ (\exists\, x.\ P\ x)) \equiv (\mathit{realizes}\ (\mathit{snd}\ t)\ (P\ (\mathit{fst}\ t)))$

$(\mathit{typeof}\ P) \equiv (\mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\ (\mathit{typeof}\ Q) \equiv (\mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (P \vee Q)) \equiv$
$\quad (\mathit{case}\ t\ \mathit{of}\ \mathit{Left} \Rightarrow \mathit{realizes}\ \mathit{Null}\ P \mid \mathit{Right} \Rightarrow \mathit{realizes}\ \mathit{Null}\ Q)$

$(\mathit{typeof}\ P) \equiv (\mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (P \vee Q)) \equiv$
$\quad (\mathit{case}\ t\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{realizes}\ \mathit{Null}\ P \mid \mathit{Some}\ q \Rightarrow \mathit{realizes}\ q\ Q)$

$(\mathit{typeof}\ Q) \equiv (\mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (P \vee Q)) \equiv$
$\quad (\mathit{case}\ t\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{realizes}\ \mathit{Null}\ Q \mid \mathit{Some}\ p \Rightarrow \mathit{realizes}\ p\ P)$

$(\mathit{realizes}\ t\ (P \vee Q)) \equiv$
$\ (\mathit{case}\ t\ \mathit{of}\ \mathit{Inl}\ p \Rightarrow \mathit{realizes}\ p\ P \mid \mathit{Inr}\ q \Rightarrow \mathit{realizes}\ q\ Q)$

$(\mathit{typeof}\ P) \equiv (\mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (P \wedge Q)) \equiv (\mathit{realizes}\ \mathit{Null}\ P \wedge \mathit{realizes}\ t\ Q)$

$(\mathit{typeof}\ Q) \equiv (\mathit{Type}\ (\mathit{TYPE}(\mathit{Null}))) \Longrightarrow$
$\quad (\mathit{realizes}\ t\ (P \wedge Q)) \equiv (\mathit{realizes}\ t\ P \wedge \mathit{realizes}\ \mathit{Null}\ Q)$

$(\mathit{realizes}\ t\ (P \wedge Q)) \equiv (\mathit{realizes}\ (\mathit{fst}\ t)\ P \wedge \mathit{realizes}\ (\mathit{snd}\ t)\ Q)$

$\mathit{typeof}\ P \equiv \mathit{Type}\ (\mathit{TYPE}(\mathit{Null})) \Longrightarrow$
$\quad \mathit{realizes}\ t\ (\neg\ P) \equiv \neg\ \mathit{realizes}\ \mathit{Null}\ P$

$\mathit{typeof}\ P \equiv \mathit{Type}\ (\mathit{TYPE}('P)) \Longrightarrow$
$\quad \mathit{realizes}\ t\ (\neg\ P) \equiv (\forall\, x{::}'P.\ \neg\ \mathit{realizes}\ x\ P)$

$\mathit{typeof}\ (P{::}\mathit{bool}) \equiv \mathit{Type}\ (\mathit{TYPE}(\mathit{Null})) \Longrightarrow$
$\ \mathit{typeof}\ Q \equiv \mathit{Type}\ (\mathit{TYPE}(\mathit{Null})) \Longrightarrow$

$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

44.4 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 $R\ (case\ x\ of\ Inl\ p \Rightarrow f\ p \mid Inr\ q \Rightarrow g\ q)$
$\langle proof \rangle$

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 $R\ (case\ x\ of\ None \Rightarrow f \mid Some\ q \Rightarrow g\ q)$
$\langle proof \rangle$

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 $R\ (case\ x\ of\ Left \Rightarrow f \mid Right \Rightarrow g)$
$\langle proof \rangle$

theorem *conjI-realizer*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
$\langle proof \rangle$

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
$impI\ (P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ P\ Q\ pq\ (h\text{: -}).\ allI \cdot - \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot (h \cdot x))$

$impI\ (P)$: $Null$
$\Lambda\ P\ Q\ (h\text{: -}).\ allI \cdot - \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot (h \cdot x))$

$impI\ (Q)$: $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ impI \cdot - \cdot -$

$impI$: $Null\ impI$

$mp\ (P,\ Q)$: $\lambda pq.\ pq$
  $\Lambda\ P\ Q\ pq\ (h$: -$)\ p.\ mp \cdot - \cdot - \bullet (spec \cdot - \cdot p \bullet h)$

$mp\ (P)$: $Null$
  $\Lambda\ P\ Q\ (h$: -$)\ p.\ mp \cdot - \cdot - \bullet (spec \cdot - \cdot p \bullet h)$

$mp\ (Q)$: $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ mp \cdot - \cdot -$

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: $\lambda x.\ x\ \Lambda\ P\ x\ (h$: -$).\ h$

$exE\ (P,\ Q)$: $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
  $\Lambda\ P\ Q\ p\ (h$: -$)\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \bullet h$

$exE\ (P)$: $Null$
  $\Lambda\ P\ Q\ p.\ exE\text{-}realizer' \cdot - \cdot - \cdot -$

$exE\ (Q)$: $\lambda x\ pq.\ pq\ x$
  $\Lambda\ P\ Q\ x\ (h1$: -$)\ pq\ (h2$: -$).\ h2 \cdot x \bullet h1$

exE: $Null$
  $\Lambda\ P\ Q\ x\ (h1$: -$)\ (h2$: -$).\ h2 \cdot x \bullet h1$

$conjI\ (P,\ Q)$: $Pair$
  $\Lambda\ P\ Q\ p\ (h$: -$)\ q.\ conjI\text{-}realizer \cdot P \cdot p \cdot Q \cdot q \bullet h$

$conjI\ (P)$: $\lambda p.\ p$
  $\Lambda\ P\ Q\ p.\ conjI \cdot - \cdot -$

$conjI\ (Q)$: $\lambda q.\ q$
  $\Lambda\ P\ Q\ (h$: -$)\ q.\ conjI \cdot - \cdot - \bullet h$

$conjI$: $Null\ conjI$

$conjunct1\ (P,\ Q)$: fst
  $\Lambda\ P\ Q\ pq.\ conjunct1 \cdot - \cdot -$

$conjunct1\ (P)$: $\lambda p.\ p$

Λ P Q p. *conjunct1* · - · -

conjunct1 (Q): *Null*
Λ P Q q. *conjunct1* · - · -

conjunct1: *Null conjunct1*

conjunct2 (P, Q): *snd*
Λ P Q pq. *conjunct2* · - · -

conjunct2 (P): *Null*
Λ P Q p. *conjunct2* · - · -

conjunct2 (Q): $\lambda p.\ p$
Λ P Q p. *conjunct2* · - · -

conjunct2: *Null conjunct2*

disjI1 (P, Q): *Inl*
Λ P Q p. *iffD2* · - · - • (*sum.cases-1* · P · - · p)

disjI1 (P): *Some*
Λ P Q p. *iffD2* · - · - • (*option.cases-2* · - · P · p)

disjI1 (Q): *None*
Λ P Q. *iffD2* · - · - • (*option.cases-1* · - · -)

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): $\lambda pq\ pr\ qr.$
  (*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): $\lambda pq\ pr\ qr.$
  (*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*
Λ *P*. *FalseE* · -

FalseE: *Null FalseE*

notI (*P*): *Null*
Λ *P* (*h*: -). *allI* · - • (Λ *x*. *notI* · - • (*h* · *x*))

notI: *Null notI*

notE (*P*, *R*): λ*p*. *arbitrary*
Λ *P R* (*h*: -) *p*. *notE* · - · - • (*spec* · - · *p* • *h*)

notE (*P*): *Null*
Λ *P R* (*h*: -) *p*. *notE* · - · - • (*spec* · - · *p* • *h*)

notE (*R*): *arbitrary*
Λ *P R*. *notE* · - · -

notE: *Null notE*

subst (*P*): λ*s t ps*. *ps*
Λ *s t P* (*h*: -) *ps*. *subst* · *s* · *t* · *P ps* • *h*

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
Λ *Q P pq* (*h*: -) *p*.
mp · - - - • (*spec* · - · *p* • (*conjunct1* · - - - • *h*))

iffD1 (*P*): λ*p*. *p*
Λ *Q P p* (*h*: -). *mp* · - - - • (*conjunct1* · - - - • *h*)

iffD1 (*Q*): *Null*
Λ *Q P q1* (*h*: -) *q2*.
mp · - - - • (*spec* · - · *q2* • (*conjunct1* · - - - • *h*))

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
Λ *P Q pq* (*h*: -) *q*.
mp · - - - • (*spec* · - · *q* • (*conjunct2* · - - - • *h*))

iffD2 (*P*): λ*p*. *p*
Λ *P Q p* (*h*: -). *mp* · - - - • (*conjunct2* · - - - • *h*)

iffD2 (*Q*): *Null*
Λ *P Q q1* (*h*: -) *q2*.
mp · - - - • (*spec* · - · *q2* • (*conjunct2* · - - - • *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

45 Reconstruction: Reconstructing external resolution proofs

theory *Reconstruction*
imports *Hilbert-Choice Map Infinite-Set Extraction*
uses *Tools/res-lib.ML*

Tools/res-clause.ML
Tools/res-skolem-function.ML
Tools/res-axioms.ML
Tools/res-types-sorts.ML

Tools/ATP/recon-order-clauses.ML
Tools/ATP/recon-translate-proof.ML
Tools/ATP/recon-parse.ML
Tools/ATP/recon-transfer-proof.ML
Tools/ATP/AtpCommunication.ML
Tools/ATP/watcher.ML
Tools/ATP/res-clasimpset.ML
Tools/res-atp.ML
Tools/reconstruction.ML

begin

$\langle ML \rangle$

end

46 Main: Main HOL

theory *Main*
imports *SAT Reconstruction*
begin

Theory *Main* includes everything. Note that theory *PreList* already includes most HOL theories.

Late clause setup: installs *all* simprules and claset rules into the clause cache; cf. theory *Reconstruction*.

$\langle ML \rangle$

end

References

[1] H. Davenport. *The Higher Arithmetic*. Cambridge University Press, 1992.