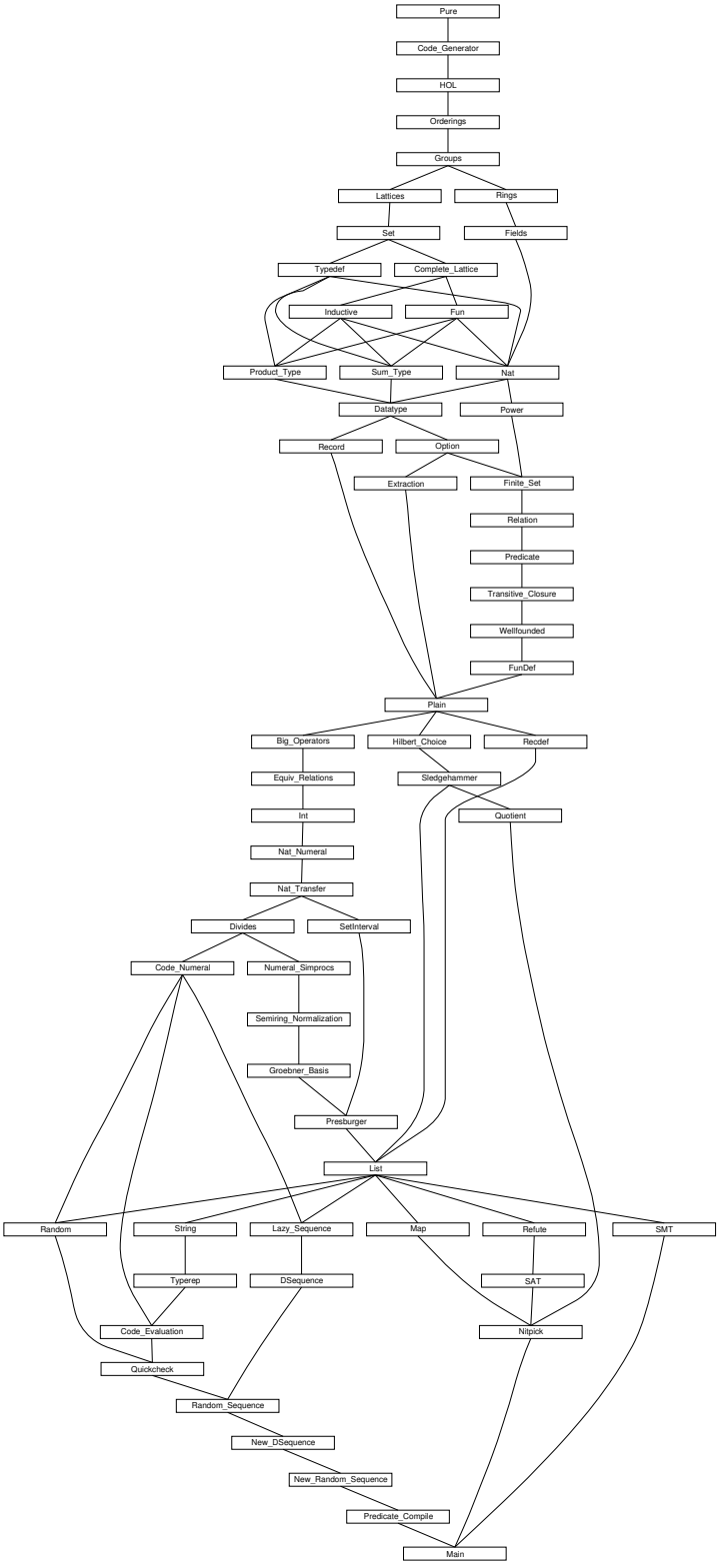
Isabelle/HOL — Higher-Order Logic

June 21, 2010

Contents


Pure
Code_Generator
HOL
Orderings
Groups
Lattices
Rings
Set
Fields
Typedef
Complete_Lattice
Inductive
Fun
Product_Type
Sum_Type
Nat
Datatype
Power
Record
Option
Extraction
Finite_Set
Relation
Predicate
Transitive_Closure
Wellfounded
FunDef
Plain
Big_Operators
Hilbert_Choice
Recdef
Equiv_Relations
Sledgehammer
Int
Quotient
Nat_Numeral
Nat_Transfer
Divides
SetInterval
Code_Numeral
Numeral_Simprocs
Semiring_Normalization
Groebner_Basis
Presburger
List
Random
String
Lazy_Sequence
Map
Refute
SMT
Typerep
DSequence
SAT
Code_Evaluation
Nitpick
Quickcheck
Random_Sequence
New_DSequence
New_Random_Sequence
Predicate_Compile
Main

1 Code-Generator: Loading the code generator modules

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

⟨*ML*⟩

end

2 HOL: The basis of Higher-Order Logic

theory *HOL*
imports *Pure* ~~/*src*/*Tools*/*Code-Generator*
uses
(*Tools*/*hologic.ML*)
~~/*src*/*Tools*/*IsaPlanner*/*zipper.ML*
~~/*src*/*Tools*/*IsaPlanner*/*isand.ML*
~~/*src*/*Tools*/*IsaPlanner*/*rw-tools.ML*
~~/*src*/*Tools*/*IsaPlanner*/*rw-inst.ML*
~~/*src*/*Tools*/*intuitionistic.ML*
~~/*src*/*Tools*/*project-rule.ML*
~~/*src*/*Tools*/*cong-tac.ML*
~~/*src*/*Provers*/*hypsubst.ML*
~~/*src*/*Provers*/*splitter.ML*
~~/*src*/*Provers*/*classical.ML*
~~/*src*/*Provers*/*blast.ML*
~~/*src*/*Provers*/*clasimp.ML*
~~/*src*/*Tools*/*coherent.ML*
~~/*src*/*Tools*/*eqsubst.ML*
~~/*src*/*Provers*/*quantifier1.ML*
(*Tools*/*simpdata.ML*)

~~/src/Tools/random-word.ML
~~/src/Tools/atomize-elim.ML
~~/src/Tools/induct.ML
(~~/src/Tools/induct-tacs.ML)
(Tools/recfun-codegen.ML)
begin

⟨*ML*⟩

2.1 Primitive logic

2.1.1 Core syntax

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

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

global

typedecl *bool*

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

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

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

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

local

consts
If :: [*bool*, '*a*, '*a*] => '*a* ((*if* (-)/ *then* (-)/ *else* (-)) [0, 0, 10] 10)

2.1.2 Additional concrete syntax

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

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

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

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

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

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

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

nonterminals
letbinds letbind
case-syn cases-syn

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

-bind :: [*pttrn*, '*a*] => *letbind* (*(2- =/ -) 10*)
:: *letbind* => *letbinds* (-)
-binds :: [*letbind*, *letbinds*] => *letbinds* (-;/ -)
-Let :: [*letbinds*, '*a*] => '*a* (*(let (-)/ in (-)) [0, 10] 10*)

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

translations

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

⟨*ML*⟩

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

notation (*xsymbols*)
All (**binder** ∀ *10*) **and**
Ex (**binder** ∃ *10*) **and**
Ex1 (**binder** ∃! *10*)

notation (*HTML* **output**)
All (**binder** ∀ *10*) **and**
Ex (**binder** ∃ *10*) **and**
Ex1 (**binder** ∃! *10*)

notation (*HOL*)
All (**binder** ! *10*) **and**
Ex (**binder** *?* *10*) **and**
Ex1 (**binder** *?!* *10*)

2.1.3 Axioms and basic definitions

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

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

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

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

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

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

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

axiomatization
$undefined :: 'a$

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

2.2 Fundamental rules

2.2.1 Equality

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

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

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

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

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

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

For calculational reasoning:

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

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

2.2.2 Congruence rules for application

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

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

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

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

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

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

⟨*ML*⟩

2.2.3 Equality of booleans – iff

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

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

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

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

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

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

2.2.4 True

lemma *TrueI*: *True*
⟨*proof*⟩

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

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

2.2.5 Universal quantifier

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

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

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

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

2.2.6 False

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

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

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

2.2.7 Negation

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

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

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

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

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

2.2.8 Implication

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

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

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

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

lemma *not-sym*: $t\ \sim{=}\ s\ =={>}\ s\ \sim{=}\ t$
⟨*proof*⟩

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

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

2.2.9 Existential quantifier

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

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

2.2.10 Conjunction

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

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

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

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

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

2.2.11 Disjunction

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

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

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

2.2.12 Classical logic

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

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

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

⟨*proof*⟩

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

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

2.2.13 Unique existence

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

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

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

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

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

2.2.14 THE: definite description operator

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

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

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

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

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

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

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

2.2.15 Classical intro rules for disjunction and existential quantifiers

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

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

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

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

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

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

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

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

2.2.16 Intuitionistic Reasoning

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

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

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

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

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

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

⟨*ML*⟩

2.2.17 Atomizing meta-level connectives

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

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

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

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

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

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

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

2.2.18 Atomizing elimination rules

⟨*ML*⟩

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

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

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

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

2.3 Package setup

2.3.1 Sledgehammer setup

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

⟨*ML*⟩

2.3.2 Classical Reasoner setup

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

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

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

⟨*ML*⟩

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

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

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

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

⟨*ML*⟩

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

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

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

lemma *alt-ex1E* [*elim*!]:
assumes *major*: ∃!*x*. *P x*
and *prem*: ⋀*x*. ⟦ *P x*; ∀ *y y'*. *P y* ∧ *P y'* ⟶ *y* = *y'* ⟧ ⟹ *R*
shows *R*
⟨*proof*⟩

⟨*ML*⟩

2.3.3 Simplifier

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

lemma *simp-thms*:
shows *not-not*: $(\sim\ \sim\ P) = P$
and *Not-eq-iff*: $((\sim P) = (\sim Q)) = (P = Q)$
and
$(P \sim= Q) = (P = (\sim Q))$
$(P \mid \sim P) = True \quad (\sim P \mid P) = True$
$(x = x) = True$
and *not-True-eq-False* [*code*]: $(\neg\ True) = False$
and *not-False-eq-True* [*code*]: $(\neg\ False) = True$
and
$(\sim P) \sim= P \ \ P \sim= (\sim P)$
$(True=P) = P$
and *eq-True*: $(P = True) = P$
and $(False=P) = (\sim P)$
and *eq-False*: $(P = False) = (\neg\ P)$
and
$(True \longrightarrow P) = P \ \ (False \longrightarrow P) = True$
$(P \longrightarrow True) = True \ \ (P \longrightarrow P) = True$
$(P \longrightarrow False) = (\sim P) \ \ (P \longrightarrow \sim P) = (\sim P)$
$(P\ \&\ True) = P \ \ (True\ \&\ P) = P$
$(P\ \&\ False) = False \ \ (False\ \&\ P) = False$
$(P\ \&\ P) = P \ \ (P\ \&\ (P\ \&\ Q)) = (P\ \&\ Q)$
$(P\ \&\ \sim P) = False \quad (\sim P\ \&\ P) = False$
$(P \mid True) = True \ \ (True \mid P) = True$
$(P \mid False) = P \ \ (False \mid P) = P$
$(P \mid P) = P \ \ (P \mid (P \mid Q)) = (P \mid Q)$ **and**
$(ALL\ x.\ P) = P \ \ (EX\ x.\ P) = P \ \ EX\ x.\ x{=}t \ \ EX\ x.\ t{=}x$
and
$!!P.\ (EX\ x.\ x{=}t\ \&\ P(x)) = P(t)$
$!!P.\ (EX\ x.\ t{=}x\ \&\ P(x)) = P(t)$
$!!P.\ (ALL\ x.\ x{=}t \longrightarrow P(x)) = P(t)$
$!!P.\ (ALL\ x.\ t{=}x \longrightarrow P(x)) = P(t)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The | congruence rule: not included by default!

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

if-then-else rules

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

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

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

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

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

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

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

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

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

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

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

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

let rules for simproc

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

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

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

definition *simp-implies* :: [*prop*, *prop*] ⇒ *prop* (**infixr** ⟹*simp*⟹ *1*) **where**
[*code del*]: *simp-implies* ≡ *op* ⟹

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

lemma *simp-impliesE*:
assumes *PQ*: $\mathit{PROP}\ P$ ⟹*simp*⟹ $\mathit{PROP}\ Q$
and *P*: $\mathit{PROP}\ P$

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

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

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

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

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

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

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

⟨*ML*⟩

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

⟨*ML*⟩

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

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

⟨*proof*⟩

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

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

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

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

⟨*ML*⟩

Simplifies x assuming c and y assuming c

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

⟨*proof*⟩

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

lemma *if-weak-cong* [*cong*]:
assumes $b = c$
shows (*if b then x else y*) = (*if c then x else y*)
⟨*proof*⟩

Prevents simplification of t: much faster

lemma *let-weak-cong*:
assumes $a = b$
shows (*let x = a in t x*) = (*let x = b in t x*)
⟨*proof*⟩

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

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

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

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

lemma *restrict-to-left*:
assumes $x = y$
shows $(x = z) = (y = z)$
⟨*proof*⟩

2.3.4 Generic cases and induction

Rule projections:

⟨*ML*⟩

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

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

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

definition *induct-conj* **where**

induct-conj A B == A ∧ B

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

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

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

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

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

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

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

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

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

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

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

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

Method setup.

⟨*ML*⟩

Pre-simplification of induction and cases rules

lemma [*induct-simp*]: $(!!x.\ \textit{induct-equal}\ x\ t \Longrightarrow \textit{PROP}\ P\ x) \equiv \textit{PROP}\ P\ t$
⟨*proof*⟩

lemma [*induct-simp*]: $(!!x.\ \textit{induct-equal}\ t\ x \Longrightarrow \textit{PROP}\ P\ x) \equiv \textit{PROP}\ P\ t$
⟨*proof*⟩

lemma [*induct-simp*]: $(\textit{induct-false} \Longrightarrow P) \equiv \textit{Trueprop}\ \textit{induct-true}$
⟨*proof*⟩

lemma [*induct-simp*]: $(\textit{induct-true} \Longrightarrow \textit{PROP}\ P) \equiv \textit{PROP}\ P$
⟨*proof*⟩

lemma [*induct-simp*]: $(\textit{PROP}\ P \Longrightarrow \textit{induct-true}) \equiv \textit{Trueprop}\ \textit{induct-true}$
⟨*proof*⟩

lemma [*induct-simp*]: $(!!x.\ \textit{induct-true}) \equiv \textit{Trueprop}\ \textit{induct-true}$
⟨*proof*⟩

lemma [*induct-simp*]: $\textit{induct-implies}\ \textit{induct-true}\ P \equiv P$
⟨*proof*⟩

lemma [*induct-simp*]: $(x = x) = \textit{True}$
⟨*proof*⟩

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

⟨*ML*⟩

2.3.5 Coherent logic

⟨*ML*⟩

2.3.6 Reorienting equalities

⟨*ML*⟩

2.4 Other simple lemmas and lemma duplicates

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

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

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

lemma *nnf-simps*:
$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)\ (\neg (P \vee Q)) = (\neg P \wedge \neg Q)\ (P \longrightarrow Q) = (\neg P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))\ (\neg(P = Q)) = ((P \wedge \neg Q) \vee (\neg P \wedge Q))$

$(\neg\ \neg(P)) = P$
⟨*proof*⟩

2.5 Basic ML bindings

⟨*ML*⟩

2.6 Code generator setup

2.6.1 SML code generator setup

⟨*ML*⟩

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

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

⟨*ML*⟩

2.6.2 Generic code generator preprocessor setup

⟨*ML*⟩

2.6.3 Equality

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

lemma *eq* [*code-unfold*, *code-inline del*]: *eq* = (*op* =)
⟨*proof*⟩

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

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

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

end

declare *equals-eq* [*code*]

⟨*ML*⟩

2.6.4 Generic code generator foundation

Datatypes

code-datatype *True False*

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

code-datatype *prop Trueprop*

Code equations

lemma [*code*]:
shows (*True* ⟹ *PROP Q*) ≡ *PROP Q*
and (*PROP Q* ⟹ *True*) ≡ *Trueprop True*
and (*P* ⟹ *R*) ≡ *Trueprop* (*P* ⟶ *R*) ⟨*proof*⟩

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

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

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

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

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

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

instance ⟨*proof*⟩

end

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

Equality

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

⟨*ML*⟩

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

⟨*ML*⟩

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

Cases

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

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

⟨*ML*⟩

code-abort *undefined*

2.6.5 Generic code generator target languages

type bool

code-type *bool*

(*SML bool*)
(*OCaml bool*)
(*Haskell Bool*)
(*Scala Boolean*)

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

code-reserved *SML*
bool true false not

code-reserved *OCaml*
bool not

code-reserved *Scala*
Boolean

using built-in Haskell equality

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

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

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

undefined

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

2.6.6 Evaluation and normalization by evaluation

Avoid some named infixes in evaluation environment

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

⟨*ML*⟩

2.7 Counterexample Search Units

2.7.1 Quickcheck

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

2.7.2 Nitpick setup

⟨*ML*⟩

2.8 Preprocessing for the predicate compiler

⟨*ML*⟩

2.9 Legacy tactics and ML bindings

⟨*ML*⟩

end

3 Orderings: Abstract orderings

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

3.1 Syntactic orders

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

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

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

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

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

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

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

end

3.2 Quasi orders

class *preorder* = *ord* +
assumes *less-le-not-le*: $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$
and *order-refl* [*iff*]: $x \leq x$
and *order-trans*: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
begin

Reflexivity.

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

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

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

Asymmetry.

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

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

Transitivity.

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

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

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

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

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

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

Transitivity rules for calculational reasoning

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

Dual order

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

end

3.3 Partial orders

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

Reflexivity.

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$
⟨*proof*⟩

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

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

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

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

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

Transitivity rules for calculational reasoning

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

⟨*proof*⟩

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

Asymmetry.

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

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

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

Least value operator

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

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

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

Dual order

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

end

3.4 Linear (total) orders

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dual order

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

⟨*proof*⟩

min/max

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

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

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

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

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

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

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

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

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

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

end

Explicit dictionaries for code generation

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

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

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

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

3.5 Reasoning tools setup

⟨*ML*⟩

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

context *order*
begin

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

context *linorder*
begin

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

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

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

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

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

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

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

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

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

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

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

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

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

op <= op <]

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

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

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

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

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

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

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

end

⟨*ML*⟩

3.6 Bounded quantifiers

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

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

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

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

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

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

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

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

translations
$ALL\ x{<}y.\ P \ \Rightarrow\ ALL\ x.\ x < y \longrightarrow P$
$EX\ x{<}y.\ P \ \Rightarrow\ EX\ x.\ x < y \wedge P$
$ALL\ x{<}{=}y.\ P \ \Rightarrow\ ALL\ x.\ x <= y \longrightarrow P$
$EX\ x{<}{=}y.\ P \ \Rightarrow\ EX\ x.\ x <= y \wedge P$
$ALL\ x{>}y.\ P \ \Rightarrow\ ALL\ x.\ x > y \longrightarrow P$
$EX\ x{>}y.\ P \ \Rightarrow\ EX\ x.\ x > y \wedge P$
$ALL\ x{>}{=}y.\ P \ \Rightarrow\ ALL\ x.\ x >= y \longrightarrow P$
$EX\ x{>}{=}y.\ P \ \Rightarrow\ EX\ x.\ x >= y \wedge P$

⟨*ML*⟩

3.7 Transitivity reasoning

context *ord*
begin

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

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

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

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

end

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemmas [*trans*] =
trans

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

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Monotonicity, least value operator and min/max

context *order*
begin

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

lemma *monoI* [*intro?*]:
fixes $f :: 'a \Rightarrow 'b::order$
shows $(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow mono\ f$
⟨*proof*⟩

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

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

lemma *strict-monoI* [*intro?*]:
assumes $\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y$
shows *strict-mono f*
⟨*proof*⟩

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

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

end

context *linorder*
begin

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

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

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

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

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

end

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

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

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

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

3.9 Top and bottom elements

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

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

3.10 Dense orders

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

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

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

end

3.11 Wellorders

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

begin

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

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

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

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

end

3.12 Order on bool

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

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

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

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

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

instance ⟨*proof*⟩

end

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

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

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

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

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

lemma *top-boolI*: *top*
⟨*proof*⟩

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

3.13 Order on functions

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

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

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

instance ⟨*proof*⟩

end

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

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

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

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

3.14 Name duplicates

lemmas *order-eq-refl* = *preorder-class.eq-refl*
lemmas *order-less-irrefl* = *preorder-class.less-irrefl*
lemmas *order-less-imp-le* = *preorder-class.less-imp-le*
lemmas *order-less-not-sym* = *preorder-class.less-not-sym*
lemmas *order-less-asym* = *preorder-class.less-asym*
lemmas *order-less-trans* = *preorder-class.less-trans*
lemmas *order-le-less-trans* = *preorder-class.le-less-trans*
lemmas *order-less-le-trans* = *preorder-class.less-le-trans*
lemmas *order-less-imp-not-less* = *preorder-class.less-imp-not-less*
lemmas *order-less-imp-triv* = *preorder-class.less-imp-triv*
lemmas *order-less-asym'* = *preorder-class.less-asym'*

lemmas *order-less-le* = *order-class.less-le*
lemmas *order-le-less* = *order-class.le-less*
lemmas *order-le-imp-less-or-eq* = *order-class.le-imp-less-or-eq*
lemmas *order-less-imp-not-eq* = *order-class.less-imp-not-eq*
lemmas *order-less-imp-not-eq2* = *order-class.less-imp-not-eq2*
lemmas *order-neq-le-trans* = *order-class.neq-le-trans*
lemmas *order-le-neq-trans* = *order-class.le-neq-trans*
lemmas *order-antisym* = *order-class.antisym*
lemmas *order-eq-iff* = *order-class.eq-iff*
lemmas *order-antisym-conv* = *order-class.antisym-conv*

lemmas *linorder-linear* = *linorder-class.linear*
lemmas *linorder-less-linear* = *linorder-class.less-linear*
lemmas *linorder-le-less-linear* = *linorder-class.le-less-linear*
lemmas *linorder-le-cases* = *linorder-class.le-cases*
lemmas *linorder-not-less* = *linorder-class.not-less*

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

end

4 Groups: Groups, also combined with orderings

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

4.1 Fact collections

⟨*ML*⟩

The rewrites accumulated in *algebra-simps* deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by *field-simps*.

⟨*ML*⟩

Lemmas *field-simps* multiply with denominators in (in)equations if they can be proved to be non-zero (for equations) or positive/negative (for inequations). Can be too aggressive and is therefore separate from the more benign *algebra-simps*.

⟨*ML*⟩

4.2 Abstract structures

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

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

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

begin

lemma *left-commute* [*ac-simps*]:
  $b * (a * c) = a * (b * c)$
⟨*proof*⟩

end

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

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

sublocale *comm-monoid* < *monoid* ⟨*proof*⟩

4.3 Generic operations

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

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

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

syntax
  -*index1* :: *index* ($_1$)
translations
  (*index*) $_1$ => (*index*) $_\diamond$

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

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

⟨*ML*⟩

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

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

class *uminus* =

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

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

⟨*ML*⟩

4.4 Semigroups and Monoids

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

sublocale *semigroup-add* < *add*!: *semigroup plus* ⟨*proof*⟩

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

sublocale *ab-semigroup-add* < *add*!: *abel-semigroup plus* ⟨*proof*⟩

context *ab-semigroup-add*
begin

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

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

end

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

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

sublocale *semigroup-mult* < *mult*!: *semigroup times* ⟨*proof*⟩

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

sublocale *ab-semigroup-mult* < *mult*!: *abel-semigroup times* ⟨*proof*⟩

context *ab-semigroup-mult*
begin

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

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

end

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

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

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

end

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

4.5 Groups

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

lemma *minus-unique*:
  assumes $a + b = 0$ **shows** $- a = b$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

The next two equations can make the simplifier loop!

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

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

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

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

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

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

end

class *ab-group-add* = *minus* + *uminus* + *comm-monoid-add* +
assumes *ab-left-minus*: $- a + a = 0$
assumes *ab-diff-minus*: $a - b = a + (- b)$
begin

subclass *group-add*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

end

4.6 (Partially) Ordered Groups

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

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

1979

- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

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

class *ordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
  assumes *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
begin

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

non-strict, in both arguments

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

end

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

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

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

Strict monotonicity in both arguments

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

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

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

end

class *ordered-ab-semigroup-add-imp-le* =
ordered-cancel-ab-semigroup-add +
assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$ **shows** $a < b$
⟨*proof*⟩

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

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

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

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

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

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

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

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

end

4.7 Support for reasoning about signs

class *ordered-comm-monoid-add* =

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

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

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

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

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

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

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

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

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

lemmas *add-sign-intros* =
add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

lemma *add-nonneg-eq-0-iff*:
assumes x: $0 \leq x$ **and** y: $0 \leq y$
shows $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

end

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

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

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

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

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

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

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

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

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

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

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

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

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

The next several equations can make the simplifier loop!

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

end

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

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

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

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

end

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *minus-min-eq-max*:

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

end

context *ordered-comm-monoid-add*
begin

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

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

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

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

end

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

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

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

end

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

class *abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +
assumes *abs-if*: $|a| = (if\ a < 0\ then\ -\ a\ else\ a)$

class *sgn-if* = *minus* + *uminus* + *zero* + *one* + *ord* + *sgn* +
assumes *sgn-if*: $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -\ 1)$
begin

lemma *sgn0* [*simp*]: $sgn\ 0 = 0$

⟨*proof*⟩

end

class *ordered-ab-group-add-abs* = *ordered-ab-group-add* + *abs* +
  assumes *abs-ge-zero* [*simp*]: $|a| \geq 0$
    and *abs-ge-self*: $a \leq |a|$
    and *abs-leI*: $a \leq b \Longrightarrow - a \leq b \Longrightarrow |a| \leq b$
    and *abs-minus-cancel* [*simp*]: $|-a| = |a|$
    and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$
begin

lemma *abs-minus-le-zero*: $- |a| \leq 0$
  ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

Needed for abelian cancellation simprocs:

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

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

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

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

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

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

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

4.8 Tools setup

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

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

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

lemmas *diff-less-0-iff-less* [*simp*, *no-atp*] = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-le-0-iff-le* [*simp*, *no-atp*] = *le-iff-diff-le-0* [*symmetric*]

⟨*ML*⟩

code-modulename *SML*
Groups Arith

code-modulename *OCaml*
Groups Arith

code-modulename *Haskell*

Groups Arith

end

5 Lattices: Abstract lattices

theory *Lattices*
imports *Orderings Groups*
begin

5.1 Abstract semilattice

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

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

lemma *left-idem* [*simp*]:
  $f\ a\ (f\ a\ b) = f\ a\ b$
  ⟨*proof*⟩

end

5.2 Idempotent semigroup

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

sublocale *ab-semigroup-idem-mult* < *times*!: *semilattice times* ⟨*proof*⟩

context *ab-semigroup-idem-mult*
begin

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

end

5.3 Concrete lattices

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

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

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

Dual lattice

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

end

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

5.3.1 Intro and elim rules

context *semilattice-inf*
begin

lemma *le-infI1*:
  $a \sqsubseteq x \Longrightarrow a \sqcap b \sqsubseteq x$
  ⟨*proof*⟩

lemma *le-infI2*:
  $b \sqsubseteq x \Longrightarrow a \sqcap b \sqsubseteq x$
  ⟨*proof*⟩

lemma *le-infI*: $x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcap b$
  ⟨*proof*⟩

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

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

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

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

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

end

context *semilattice-sup*
begin

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

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

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

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

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

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

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

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

end

5.3.2 Equational laws

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

context *semilattice-inf*
begin

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

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

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

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

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

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

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

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

end

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

context *semilattice-sup*
begin

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

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

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

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

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

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

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

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

end

context *lattice*
begin

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

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

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

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

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

Towards distributivity

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

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

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

lemma *distrib-imp1*:
assumes *D*: $!!x\ y\ z.\ x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
shows $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
⟨*proof*⟩

lemma *distrib-imp2*:
assumes *D*: $!!x\ y\ z.\ x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
⟨*proof*⟩

end

5.3.3 Strict order

context *semilattice-inf*
begin

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

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

end

context *semilattice-sup*
begin

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

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

end

5.4 Distributive lattices

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

context *distrib-lattice*
begin

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

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

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

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

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

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

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

end

5.5 Bounded lattices and boolean algebras

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

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

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

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

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

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

end

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

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

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

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

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

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

end

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

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

end

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

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

end

5.6 Uniqueness of inf and sup

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

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

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

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

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

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

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

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

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

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

5.8 Bool as lattice

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

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

definition
bool-diff-def: $A - B \longleftrightarrow A \wedge \neg B$

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

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

instance ⟨*proof*⟩

end

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

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

⟨*proof*⟩

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

5.9 Fun as lattice

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

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

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

instance ⟨*proof*⟩

end

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

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

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

end

6 Set: Set theory for higher-order logic

theory *Set*
imports *Lattices*
begin

6.1 Sets as predicates

global

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

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

local

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

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

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

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

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

not-mem ((-/ ∉ -) [*50*, *51*] *50*)

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

Set comprehensions

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

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

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

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

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

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

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

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

⟨*ML*⟩

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

Set enumerations

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

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

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

6.2 Subsets and bounded quantifiers

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

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

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

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

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

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

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

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

global

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

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

local

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

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

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

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

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

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

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

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

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

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

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

⟨*ML*⟩

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

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

⟨*ML*⟩

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

lemmas *strip* = *impI allI ballI*

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

Gives better instantiation for bound:

⟨*ML*⟩

lemma *ballE* [*elim*]: $ALL\ x{:}A.\ P\ x\ ==>\ (P\ x\ ==>\ Q)\ ==>\ (x\ \tilde{}{:}\ A\ ==>\ Q)\ ==>\ Q$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

Congruence rules

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

⟨*proof*⟩

lemma *strong-ball-cong* [*cong*]:
$A = B$ ==> (!!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*⟩

6.3 Basic operations

6.3.1 Subsets

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

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

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

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

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

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

lemma *subset-eq* [*no-atp*]: $A \leq B = (\forall x{\in}A.\ x \in B)$ ⟨*proof*⟩

lemma *contra-subsetD* [*no-atp*]: $A \subseteq B$ ==> $c \notin B$ ==> $c \notin A$
⟨*proof*⟩

lemma *subset-refl* [*simp*]: $A \subseteq A$
⟨*proof*⟩

lemma *subset-trans*: $A \subseteq B$ ==> $B \subseteq C$ ==> $A \subseteq C$

⟨*proof*⟩

lemma *set-rev-mp*: x:A ==> $A \subseteq B$ ==> x:B
⟨*proof*⟩

lemma *set-mp*: $A \subseteq B$ ==> x:A ==> x:B
⟨*proof*⟩

lemma *eq-mem-trans*: a=b ==> $b \in A$ ==> $a \in A$
⟨*proof*⟩

lemmas *basic-trans-rules* [*trans*] =
order-trans-rules set-rev-mp set-mp eq-mem-trans

6.3.2 Equality

lemma *set-ext*: **assumes** *prem*: (!!x. (x:A) = (x:B)) **shows** $A = B$
⟨*proof*⟩

lemma *expand-set-eq*: ($A = B$) = (*ALL* x. (x:A) = (x:B))
⟨*proof*⟩

lemma *subset-antisym* [*intro*!]: $A \subseteq B$ ==> $B \subseteq A$ ==> $A = B$
— Anti-symmetry of the subset relation.
⟨*proof*⟩

Equality rules from ZF set theory – are they appropriate here?

lemma *equalityD1*: $A = B$ ==> $A \subseteq B$
⟨*proof*⟩

lemma *equalityD2*: $A = B$ ==> $B \subseteq A$
⟨*proof*⟩

Be careful when adding this to the claset as *subset-empty* is in the simpset: $A = \{\}$ goes to $\{\} \subseteq A$ and $A \subseteq \{\}$ and then back to $A = \{\}$!

lemma *equalityE*: $A = B$ ==> ($A \subseteq B$ ==> $B \subseteq A$ ==> P) ==> P
⟨*proof*⟩

lemma *equalityCE* [*elim*]:
$A = B$ ==> ($c \in A$ ==> $c \in B$ ==> P) ==> ($c \notin A$ ==> $c \notin B$ ==> P)
==> P
⟨*proof*⟩

lemma *eqset-imp-iff*: $A = B$ ==> (x : A) = (x : B)
⟨*proof*⟩

lemma *eqelem-imp-iff*: $x = y$ ==> (x : A) = (y : A)
⟨*proof*⟩

6.3.3 The universal set – UNIV

abbreviation *UNIV* :: *'a set* **where**
UNIV ≡ *top*

lemma *UNIV-def*:
UNIV = {*x*. *True*}
⟨*proof*⟩

lemma *UNIV-I* [*simp*]: *x* : *UNIV*
⟨*proof*⟩

declare *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

lemma *UNIV-witness* [*intro?*]: *EX x*. *x* : *UNIV*
⟨*proof*⟩

lemma *subset-UNIV* [*simp*]: *A* ⊆ *UNIV*
⟨*proof*⟩

Eta-contracting these two rules (to remove *P*) causes them to be ignored because of their interaction with congruence rules.

lemma *ball-UNIV* [*simp*]: *Ball UNIV P* = *All P*
⟨*proof*⟩

lemma *bex-UNIV* [*simp*]: *Bex UNIV P* = *Ex P*
⟨*proof*⟩

lemma *UNIV-eq-I*: (⋀*x*. *x* ∈ *A*) ⟹ *UNIV* = *A*
⟨*proof*⟩

6.3.4 The empty set

lemma *empty-def*:
{} = {*x*. *False*}
⟨*proof*⟩

lemma *empty-iff* [*simp*]: (*c* : {}) = *False*
⟨*proof*⟩

lemma *emptyE* [*elim!*]: *a* : {} ==> *P*
⟨*proof*⟩

lemma *empty-subsetI* [*iff*]: {} ⊆ *A*
— One effect is to delete the ASSUMPTION {} ⊆ *A*
⟨*proof*⟩

lemma *equals0I*: (!!*y*. *y* ∈ *A* ==> *False*) ==> *A* = {}
⟨*proof*⟩

lemma *equals0D*: $A = \{\} \Longrightarrow a \notin A$
— Use for reasoning about disjointness: $A\ Int\ B = \{\}$
⟨*proof*⟩

lemma *ball-empty* [*simp*]: $Ball\ \{\}\ P = True$
⟨*proof*⟩

lemma *bex-empty* [*simp*]: $Bex\ \{\}\ P = False$
⟨*proof*⟩

lemma *UNIV-not-empty* [*iff*]: $UNIV \neq \{\}$
⟨*proof*⟩

6.3.5 The Powerset operator – Pow

definition $Pow :: 'a\ set \Rightarrow 'a\ set\ set$ **where**
Pow-def: $Pow\ A = \{B.\ B \leq A\}$

lemma *Pow-iff* [*iff*]: $(A \in Pow\ B) = (A \subseteq B)$
⟨*proof*⟩

lemma *PowI*: $A \subseteq B \Longrightarrow A \in Pow\ B$
⟨*proof*⟩

lemma *PowD*: $A \in Pow\ B \Longrightarrow A \subseteq B$
⟨*proof*⟩

lemma *Pow-bottom*: $\{\} \in Pow\ B$
⟨*proof*⟩

lemma *Pow-top*: $A \in Pow\ A$
⟨*proof*⟩

6.3.6 Set complement

lemma *Compl-iff* [*simp*]: $(c \in -A) = (c \notin A)$
⟨*proof*⟩

lemma *ComplI* [*intro!*]: $(c \in A \Longrightarrow False) \Longrightarrow c \in -A$
⟨*proof*⟩

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile ...

lemma *ComplD* [*dest!*]: $c : -A \Longrightarrow c\sim:A$
⟨*proof*⟩

lemmas *ComplE* = *ComplD* [*elim-format*]

lemma *Compl-eq*: $- A = \{x.\ \sim x : A\}$ ⟨*proof*⟩

6.3.7 Binary union – Un

abbreviation *union* :: *'a set* ⇒ *'a set* ⇒ *'a set* (**infixl** *Un 65*) **where**
op Un ≡ *sup*

notation (*xsymbols*)
union (**infixl** ∪ *65*)

notation (*HTML* **output**)
union (**infixl** ∪ *65*)

lemma *Un-def*:
$A \cup B = \{x.\ x \in A \vee x \in B\}$
⟨*proof*⟩

lemma *Un-iff* [*simp*]: $(c : A\ Un\ B) = (c{:}A \mid c{:}B)$
⟨*proof*⟩

lemma *UnI1* [*elim?*]: $c{:}A \Longrightarrow c : A\ Un\ B$
⟨*proof*⟩

lemma *UnI2* [*elim?*]: $c{:}B \Longrightarrow c : A\ Un\ B$
⟨*proof*⟩

Classical introduction rule: no commitment to A vs B.

lemma *UnCI* [*intro*!]: $(c^{\sim}{:}B \Longrightarrow c{:}A) \Longrightarrow c : A\ Un\ B$
⟨*proof*⟩

lemma *UnE* [*elim*!]: $c : A\ Un\ B \Longrightarrow (c{:}A \Longrightarrow P) \Longrightarrow (c{:}B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *insert-def*: $insert\ a\ B = \{x.\ x = a\} \cup B$
⟨*proof*⟩

lemma *mono-Un*: $mono\ f \Longrightarrow f\ A \cup f\ B \subseteq f\ (A \cup B)$
⟨*proof*⟩

6.3.8 Binary intersection – Int

abbreviation *inter* :: *'a set* ⇒ *'a set* ⇒ *'a set* (**infixl** *Int 70*) **where**
op Int ≡ *inf*

notation (*xsymbols*)
inter (**infixl** ∩ *70*)

notation (*HTML* **output**)
inter (**infixl** ∩ *70*)

lemma *Int-def*:
$A \cap B = \{x.\ x \in A \wedge x \in B\}$
⟨*proof*⟩

lemma *Int-iff* [*simp*]: $(c : A\ Int\ B) = (c{:}A\ \&\ c{:}B)$
⟨*proof*⟩

lemma *IntI* [*intro*!]: $c{:}A \Longrightarrow c{:}B \Longrightarrow c : A\ Int\ B$
⟨*proof*⟩

lemma *IntD1*: $c : A\ Int\ B \Longrightarrow c{:}A$
⟨*proof*⟩

lemma *IntD2*: $c : A\ Int\ B \Longrightarrow c{:}B$
⟨*proof*⟩

lemma *IntE* [*elim*!]: $c : A\ Int\ B \Longrightarrow (c{:}A \Longrightarrow c{:}B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *mono-Int*: $mono\ f \Longrightarrow f\ (A \cap B) \subseteq f\ A \cap f\ B$
⟨*proof*⟩

6.3.9 Set difference

lemma *Diff-iff* [*simp*]: $(c : A - B) = (c{:}A\ \&\ c{\sim}{:}B)$
⟨*proof*⟩

lemma *DiffI* [*intro*!]: $c : A \Longrightarrow c\ {\sim}{:}\ B \Longrightarrow c : A - B$
⟨*proof*⟩

lemma *DiffD1*: $c : A - B \Longrightarrow c : A$
⟨*proof*⟩

lemma *DiffD2*: $c : A - B \Longrightarrow c : B \Longrightarrow P$
⟨*proof*⟩

lemma *DiffE* [*elim*!]: $c : A - B \Longrightarrow (c{:}A \Longrightarrow c{\sim}{:}B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *set-diff-eq*: $A - B = \{x.\ x : A\ \&\ {\sim}\ x : B\}$ ⟨*proof*⟩

lemma *Compl-eq-Diff-UNIV*: $-A = (UNIV - A)$
⟨*proof*⟩

6.3.10 Augmenting a set − *insert*

lemma *insert-iff* [*simp*]: $(a : insert\ b\ A) = (a = b \mid a{:}A)$

⟨*proof*⟩

lemma *insertI1*: $a : \mathit{insert}\ a\ B$
⟨*proof*⟩

lemma *insertI2*: $a : B \Longrightarrow a : \mathit{insert}\ b\ B$
⟨*proof*⟩

lemma *insertE* [*elim*!]: $a : \mathit{insert}\ b\ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a{:}A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *insertCI* [*intro*!]: $(a^{\sim}{:}B \Longrightarrow a = b) \Longrightarrow a{:}\ \mathit{insert}\ b\ B$
— Classical introduction rule.
⟨*proof*⟩

lemma *subset-insert-iff*: $(A \subseteq \mathit{insert}\ x\ B) = (\mathit{if}\ x{:}A\ \mathit{then}\ A - \{x\} \subseteq B\ \mathit{else}\ A \subseteq B)$
⟨*proof*⟩

lemma *set-insert*:
assumes $x \in A$
obtains B **where** $A = \mathit{insert}\ x\ B$ **and** $x \notin B$
⟨*proof*⟩

lemma *insert-ident*: $x\ {}^{\sim}{:}\ A \Longrightarrow x\ {}^{\sim}{:}\ B \Longrightarrow (\mathit{insert}\ x\ A = \mathit{insert}\ x\ B) = (A = B)$
⟨*proof*⟩

6.3.11 Singletons, using insert

lemma *singletonI* [*intro*!,*no-atp*]: $a : \{a\}$
— Redundant? But unlike *insertCI*, it proves the subgoal immediately!
⟨*proof*⟩

lemma *singletonD* [*dest*!,*no-atp*]: $b : \{a\} \Longrightarrow b = a$
⟨*proof*⟩

lemmas *singletonE* = *singletonD* [*elim-format*]

lemma *singleton-iff*: $(b : \{a\}) = (b = a)$
⟨*proof*⟩

lemma *singleton-inject* [*dest*!]: $\{a\} = \{b\} \Longrightarrow a = b$
⟨*proof*⟩

lemma *singleton-insert-inj-eq* [*iff*,*no-atp*]:
$(\{b\} = \mathit{insert}\ a\ A) = (a = b\ \&\ A \subseteq \{b\})$
⟨*proof*⟩

lemma *singleton-insert-inj-eq'* [*iff*,*no-atp*]:
(*insert a A* = {*b*}) = (*a* = *b* & *A* ⊆ {*b*})
⟨*proof*⟩

lemma *subset-singletonD*: *A* ⊆ {*x*} ==> *A* = {} | *A* = {*x*}
⟨*proof*⟩

lemma *singleton-conv* [*simp*]: {*x*. *x* = *a*} = {*a*}
⟨*proof*⟩

lemma *singleton-conv2* [*simp*]: {*x*. *a* = *x*} = {*a*}
⟨*proof*⟩

lemma *diff-single-insert*: *A* − {*x*} ⊆ *B* ==> *x* ∈ *A* ==> *A* ⊆ *insert x B*
⟨*proof*⟩

lemma *doubleton-eq-iff*: ({*a*,*b*} = {*c*,*d*}) = (*a*=*c* & *b*=*d* | *a*=*d* & *b*=*c*)
⟨*proof*⟩

6.3.12 Image of a set under a function

Frequently *b* does not have the syntactic form of *f x*.

definition *image* :: ('*a* => '*b*) => '*a set* => '*b set* (**infixr** ‘ *90*) **where**
image-def [*no-atp*]: *f* ‘ *A* = {*y*. *EX x*:*A*. *y* = *f*(*x*)}

abbreviation
range :: ('*a* => '*b*) => '*b set* **where** — of function
range f == *f* ‘ *UNIV*

lemma *image-eqI* [*simp*, *intro*]: *b* = *f x* ==> *x*:*A* ==> *b* : *f*‘*A*
⟨*proof*⟩

lemma *imageI*: *x* : *A* ==> *f x* : *f* ‘ *A*
⟨*proof*⟩

lemma *rev-image-eqI*: *x*:*A* ==> *b* = *f x* ==> *b* : *f*‘*A*
— This version's more effective when we already have the required *x*.
⟨*proof*⟩

lemma *imageE* [*elim*!]:
b : (%*x*. *f x*)‘*A* ==> (!!*x*. *b* = *f x* ==> *x*:*A* ==> *P*) ==> *P*
— The eta-expansion gives variable-name preservation.
⟨*proof*⟩

lemma *image-Un*: *f*‘(*A Un B*) = *f*‘*A Un f*‘*B*
⟨*proof*⟩

lemma *image-iff*: (*z* : *f*‘*A*) = (*EX x*:*A*. *z* = *f x*)

⟨*proof*⟩

lemma *image-subset-iff*: $(f`A \subseteq B) = (\forall x{\in}A.\ f\ x \in B)$
— This rewrite rule would confuse users if made default.
⟨*proof*⟩

lemma *subset-image-iff*: $(B \subseteq f`A) = (EX\ AA.\ AA \subseteq A\ \&\ B = f`AA)$
⟨*proof*⟩

lemma *image-subsetI*: $(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f`A \subseteq B$
— Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.
⟨*proof*⟩

Range of a function – just a translation for image!

lemma *range-eqI*: $b = f\ x \Longrightarrow b \in range\ f$
⟨*proof*⟩

lemma *rangeI*: $f\ x \in range\ f$
⟨*proof*⟩

lemma *rangeE* [*elim?*]: $b \in range\ (\lambda x.\ f\ x) \Longrightarrow (!!x.\ b = f\ x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

6.3.13 Some rules with *if*

Elimination of $\{x.\ \ldots\ \&\ x{=}t\ \&\ \ldots\}$.

lemma *Collect-conv-if*: $\{x.\ x{=}a\ \&\ P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$
⟨*proof*⟩

lemma *Collect-conv-if2*: $\{x.\ a{=}x\ \&\ P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$
⟨*proof*⟩

Rewrite rules for boolean case-splitting: faster than *split-if* [*split*].

lemma *split-if-eq1*: $((if\ Q\ then\ x\ else\ y) = b) = ((Q \dashrightarrow x = b)\ \&\ (\sim Q \dashrightarrow y = b))$
⟨*proof*⟩

lemma *split-if-eq2*: $(a = (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a = x)\ \&\ (\sim Q \dashrightarrow a = y))$
⟨*proof*⟩

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

lemma *split-if-mem1*: $((if\ Q\ then\ x\ else\ y) : b) = ((Q \dashrightarrow x : b)\ \&\ (\sim Q \dashrightarrow y : b))$
⟨*proof*⟩

lemma *split-if-mem2*: $(a : (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y)) = ((Q \dashrightarrow a : x)\ \&\ (\sim Q \dashrightarrow a : y))$
⟨*proof*⟩

lemmas *split-ifs = if-bool-eq-conj split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2*

6.4 Further operations and lemmas

6.4.1 The "proper subset" relation

lemma *psubsetI* [*intro*!,*no-atp*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
⟨*proof*⟩

lemma *psubsetE* [*elim*!,*no-atp*]:
$[|A \subset B;\ [|A \subseteq B;\ \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
⟨*proof*⟩

lemma *psubset-insert-iff*:
$(A \subset \mathit{insert}\ x\ B) = (\mathit{if}\ x \in B\ \mathit{then}\ A \subset B\ \mathit{else}\ \mathit{if}\ x \in A\ \mathit{then}\ A - \{x\} \subset B\ \mathit{else}\ A \subseteq B)$
⟨*proof*⟩

lemma *psubset-eq*: $(A \subset B) = (A \subseteq B\ \&\ A \neq B)$
⟨*proof*⟩

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$
⟨*proof*⟩

lemma *psubset-trans*: $[|\ A \subset B;\ B \subset C\ |] \Longrightarrow A \subset C$
⟨*proof*⟩

lemma *psubsetD*: $[|\ A \subset B;\ c \in A\ |] \Longrightarrow c \in B$
⟨*proof*⟩

lemma *psubset-subset-trans*: $A \subset B \Longrightarrow B \subseteq C \Longrightarrow A \subset C$
⟨*proof*⟩

lemma *subset-psubset-trans*: $A \subseteq B \Longrightarrow B \subset C \Longrightarrow A \subset C$
⟨*proof*⟩

lemma *psubset-imp-ex-mem*: $A \subset B \Longrightarrow \exists\, b.\ b \in (B - A)$
⟨*proof*⟩

lemma *atomize-ball*:
$(!!x.\ x \in A \Longrightarrow P\ x) \equiv \mathit{Trueprop}\ (\forall x \in A.\ P\ x)$
⟨*proof*⟩

lemmas [*symmetric*, *rulify*] = *atomize-ball*
and [*symmetric*, *defn*] = *atomize-ball*

6.4.2 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq \mathit{insert}\ a\ B$
⟨*proof*⟩

lemma *subset-insertI2*: $A \subseteq B \Longrightarrow A \subseteq \mathit{insert}\ b\ B$
⟨*proof*⟩

lemma *subset-insert*: $x \notin A \Longrightarrow (A \subseteq \mathit{insert}\ x\ B) = (A \subseteq B)$
⟨*proof*⟩

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
⟨*proof*⟩

lemma *Un-upper2*: $B \subseteq A \cup B$
⟨*proof*⟩

lemma *Un-least*: $A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow A \cup B \subseteq C$
⟨*proof*⟩

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
⟨*proof*⟩

lemma *Int-lower2*: $A \cap B \subseteq B$
⟨*proof*⟩

lemma *Int-greatest*: $C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B$
⟨*proof*⟩

Set difference.

lemma *Diff-subset*: $A - B \subseteq A$
⟨*proof*⟩

lemma *Diff-subset-conv*: $(A - B \subseteq C) = (A \subseteq B \cup C)$
⟨*proof*⟩

6.4.3 Equalities involving union, intersection, inclusion, etc.

{}.

lemma *Collect-const* [*simp*]: $\{s.\ P\} = (\mathit{if}\ P\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
— supersedes *Collect-False-empty*
⟨*proof*⟩

lemma *subset-empty* [*simp*]: $(A \subseteq \{\}) = (A = \{\})$
⟨*proof*⟩

lemma *not-psubset-empty* [*iff*]: $\neg (A < \{\})$
⟨*proof*⟩

lemma *Collect-empty-eq* [*simp*]: $(Collect\ P = \{\}) = (\forall x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *empty-Collect-eq* [*simp*]: $(\{\} = Collect\ P) = (\forall x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *Collect-neg-eq*: $\{x.\ \neg\ P\ x\} = -\ \{x.\ P\ x\}$
⟨*proof*⟩

lemma *Collect-disj-eq*: $\{x.\ P\ x \mid Q\ x\} = \{x.\ P\ x\} \cup \{x.\ Q\ x\}$
⟨*proof*⟩

lemma *Collect-imp-eq*: $\{x.\ P\ x \longrightarrow Q\ x\} = -\{x.\ P\ x\} \cup \{x.\ Q\ x\}$
⟨*proof*⟩

lemma *Collect-conj-eq*: $\{x.\ P\ x\ \&\ Q\ x\} = \{x.\ P\ x\} \cap \{x.\ Q\ x\}$
⟨*proof*⟩

insert.

lemma *insert-is-Un*: *insert a A* = $\{a\}$ *Un A*
— NOT SUITABLE FOR REWRITING since $\{a\}$ == *insert a* $\{\}$
⟨*proof*⟩

lemma *insert-not-empty* [*simp*]: *insert a A* $\neq \{\}$
⟨*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*: $a \in A \Longrightarrow \exists B.\ A = insert\ a\ B\ \&\ a \notin B$
— use new B rather than $A - \{a\}$ to avoid infinite unfolding
⟨*proof*⟩

lemma *insert-Collect*: $insert\ a\ (Collect\ P) = \{u.\ u \neq a \longrightarrow P\ u\}$
⟨*proof*⟩

lemma *insert-inter-insert*[*simp*]: $insert\ a\ A \cap insert\ a\ B = insert\ a\ (A \cap B)$
⟨*proof*⟩

lemma *insert-disjoint* [*simp*,*no-atp*]:
$(insert\ a\ A \cap B = \{\}) = (a \notin B \wedge A \cap B = \{\})$
$(\{\} = insert\ a\ A \cap B) = (a \notin B \wedge \{\} = A \cap B)$
⟨*proof*⟩

lemma *disjoint-insert* [*simp*,*no-atp*]:
$(B \cap insert\ a\ A = \{\}) = (a \notin B \wedge B \cap A = \{\})$
$(\{\} = A \cap insert\ b\ B) = (b \notin A \wedge \{\} = A \cap B)$
⟨*proof*⟩

image.

lemma *image-empty* [*simp*]: $f`\{\} = \{\}$
⟨*proof*⟩

lemma *image-insert* [*simp*]: $f\ `\ insert\ a\ B = insert\ (f\ a)\ (f`B)$
⟨*proof*⟩

lemma *image-constant*: $x \in A \Longrightarrow (\lambda x.\ c)\ `\ A = \{c\}$
⟨*proof*⟩

lemma *image-constant-conv*: $(\%x.\ c)\ `\ A = (if\ A = \{\}\ then\ \{\}\ else\ \{c\})$
⟨*proof*⟩

lemma *image-image*: $f\ `\ (g\ `\ A) = (\lambda x.\ f\ (g\ x))\ `\ A$
⟨*proof*⟩

lemma *insert-image* [*simp*]: $x \in A \Longrightarrow insert\ (f\ x)\ (f`A) = f`A$
⟨*proof*⟩

lemma *image-is-empty* [*iff*]: $(f`A = \{\}) = (A = \{\})$
⟨*proof*⟩

lemma *empty-is-image*[*iff*]: $(\{\} = f\ `\ A) = (A = \{\})$
⟨*proof*⟩

lemma *image-Collect* [*no-atp*]: $f\ `\ \{x.\ P\ x\} = \{f\ x \mid x.\ P\ x\}$
— NOT suitable as a default simprule: the RHS isn't simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational

properties than does the RHS.
⟨*proof*⟩

lemma *if-image-distrib* [*simp*]:
$(\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ f\ x\ \mathit{else}\ g\ x)\ `\ S$
$= (f\ `\ (S \cap \{x.\ P\ x\})) \cup (g\ `\ (S \cap \{x.\ \neg\ P\ x\}))$
⟨*proof*⟩

lemma *image-cong*: $M = N \Longrightarrow (!!x.\ x \in N \Longrightarrow f\ x = g\ x) \Longrightarrow f`M = g`N$
⟨*proof*⟩

range.

lemma *full-SetCompr-eq* [*no-atp*]: $\{u.\ \exists x.\ u = f\ x\} = \mathit{range}\ f$
⟨*proof*⟩

lemma *range-composition*: $\mathit{range}\ (\lambda x.\ f\ (g\ x)) = f`\mathit{range}\ g$
⟨*proof*⟩

Int

lemma *Int-absorb* [*simp*]: $A \cap A = A$
⟨*proof*⟩

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$
⟨*proof*⟩

lemma *Int-commute*: $A \cap B = B \cap A$
⟨*proof*⟩

lemma *Int-left-commute*: $A \cap (B \cap C) = B \cap (A \cap C)$
⟨*proof*⟩

lemma *Int-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
⟨*proof*⟩

lemmas *Int-ac* = *Int-assoc Int-left-absorb Int-commute Int-left-commute*
— Intersection is an AC-operator

lemma *Int-absorb1*: $B \subseteq A \Longrightarrow A \cap B = B$
⟨*proof*⟩

lemma *Int-absorb2*: $A \subseteq B \Longrightarrow A \cap B = A$
⟨*proof*⟩

lemma *Int-empty-left* [*simp*]: $\{\} \cap B = \{\}$
⟨*proof*⟩

lemma *Int-empty-right* [*simp*]: $A \cap \{\} = \{\}$
⟨*proof*⟩

lemma *disjoint-eq-subset-Compl*: $(A \cap B = \{\}) = (A \subseteq -B)$
⟨*proof*⟩

lemma *disjoint-iff-not-equal*: $(A \cap B = \{\}) = (\forall x{\in}A.\ \forall y{\in}B.\ x \neq y)$
⟨*proof*⟩

lemma *Int-UNIV-left* [*simp*]: $UNIV \cap B = B$
⟨*proof*⟩

lemma *Int-UNIV-right* [*simp*]: $A \cap UNIV = A$
⟨*proof*⟩

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
⟨*proof*⟩

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
⟨*proof*⟩

lemma *Int-UNIV* [*simp*,*no-atp*]: $(A \cap B = UNIV) = (A = UNIV\ \&\ B = UNIV)$
⟨*proof*⟩

lemma *Int-subset-iff* [*simp*]: $(C \subseteq A \cap B) = (C \subseteq A\ \&\ C \subseteq B)$
⟨*proof*⟩

lemma *Int-Collect*: $(x \in A \cap \{x.\ P\ x\}) = (x \in A\ \&\ P\ x)$
⟨*proof*⟩

Un.

lemma *Un-absorb* [*simp*]: $A \cup A = A$
⟨*proof*⟩

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
⟨*proof*⟩

lemma *Un-commute*: $A \cup B = B \cup A$
⟨*proof*⟩

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
⟨*proof*⟩

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
⟨*proof*⟩

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*
— Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B \Longrightarrow A \cup B = B$
⟨*proof*⟩

lemma *Un-absorb2*: $B \subseteq A ==> A \cup B = A$
⟨*proof*⟩

lemma *Un-empty-left* [*simp*]: $\{\} \cup B = B$
⟨*proof*⟩

lemma *Un-empty-right* [*simp*]: $A \cup \{\} = A$
⟨*proof*⟩

lemma *Un-UNIV-left* [*simp*]: $UNIV \cup B = UNIV$
⟨*proof*⟩

lemma *Un-UNIV-right* [*simp*]: $A \cup UNIV = UNIV$
⟨*proof*⟩

lemma *Un-insert-left* [*simp*]: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
⟨*proof*⟩

lemma *Un-insert-right* [*simp*]: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
⟨*proof*⟩

lemma *Int-insert-left*:
$(insert\ a\ B)\ Int\ C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
⟨*proof*⟩

lemma *Int-insert-left-if0*[*simp*]:
$a \notin C \Longrightarrow (insert\ a\ B)\ Int\ C = B \cap C$
⟨*proof*⟩

lemma *Int-insert-left-if1*[*simp*]:
$a \in C \Longrightarrow (insert\ a\ B)\ Int\ C = insert\ a\ (B\ Int\ C)$
⟨*proof*⟩

lemma *Int-insert-right*:
$A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
⟨*proof*⟩

lemma *Int-insert-right-if0*[*simp*]:
$a \notin A \Longrightarrow A\ Int\ (insert\ a\ B) = A\ Int\ B$
⟨*proof*⟩

lemma *Int-insert-right-if1*[*simp*]:
$a \in A \Longrightarrow A\ Int\ (insert\ a\ B) = insert\ a\ (A\ Int\ B)$
⟨*proof*⟩

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
⟨*proof*⟩

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
⟨*proof*⟩

lemma *Un-Int-crazy*:
$(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
⟨*proof*⟩

lemma *subset-Un-eq*: $(A \subseteq B) = (A \cup B = B)$
⟨*proof*⟩

lemma *Un-empty* [*iff*]: $(A \cup B = \{\}) = (A = \{\} \ \& \ B = \{\})$
⟨*proof*⟩

lemma *Un-subset-iff* [*simp*]: $(A \cup B \subseteq C) = (A \subseteq C \ \& \ B \subseteq C)$
⟨*proof*⟩

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
⟨*proof*⟩

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
⟨*proof*⟩

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap -A = \{\}$
⟨*proof*⟩

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
⟨*proof*⟩

lemma *Compl-partition*: $A \cup -A = UNIV$
⟨*proof*⟩

lemma *Compl-partition2*: $-A \cup A = UNIV$
⟨*proof*⟩

lemma *double-complement* [*simp*]: $-(-A) = (A::'a\ set)$
⟨*proof*⟩

lemma *Compl-Un* [*simp*]: $-(A \cup B) = (-A) \cap (-B)$
⟨*proof*⟩

lemma *Compl-Int* [*simp*]: $-(A \cap B) = (-A) \cup (-B)$
⟨*proof*⟩

lemma *subset-Compl-self-eq*: $(A \subseteq -A) = (A = \{\})$
⟨*proof*⟩

lemma *Un-Int-assoc-eq*: $((A \cap B) \cup C = A \cap (B \cup C)) = (C \subseteq A)$
— Halmos, Naive Set Theory, page 16.

⟨*proof*⟩

lemma *Compl-UNIV-eq* [*simp*]: $-UNIV = \{\}$
⟨*proof*⟩

lemma *Compl-empty-eq* [*simp*]: $-\{\} = UNIV$
⟨*proof*⟩

lemma *Compl-subset-Compl-iff* [*iff*]: $(-A \subseteq -B) = (B \subseteq A)$
⟨*proof*⟩

lemma *Compl-eq-Compl-iff* [*iff*]: $(-A = -B) = (A = (B::'a\ set))$
⟨*proof*⟩

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

lemma *ball-Un*: $(\forall x \in A \cup B.\ P\ x) = ((\forall x{\in}A.\ P\ x)\ \&\ (\forall x{\in}B.\ P\ x))$
⟨*proof*⟩

lemma *bex-Un*: $(\exists x \in A \cup B.\ P\ x) = ((\exists x{\in}A.\ P\ x) \mid (\exists x{\in}B.\ P\ x))$
⟨*proof*⟩

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
⟨*proof*⟩

lemma *Diff-eq-empty-iff* [*simp*,*no-atp*]: $(A - B = \{\}) = (A \subseteq B)$
⟨*proof*⟩

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
⟨*proof*⟩

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - (B::'a\ set)$
⟨*proof*⟩

lemma *Diff-triv*: $A \cap B = \{\} \Longrightarrow A - B = A$
⟨*proof*⟩

lemma *empty-Diff* [*simp*]: $\{\} - A = \{\}$
⟨*proof*⟩

lemma *Diff-empty* [*simp*]: $A - \{\} = A$
⟨*proof*⟩

lemma *Diff-UNIV* [*simp*]: $A - UNIV = \{\}$
⟨*proof*⟩

lemma *Diff-insert0* [*simp*,*no-atp*]: $x \notin A \Longrightarrow A - \mathit{insert}\ x\ B = A - B$
⟨*proof*⟩

lemma *Diff-insert*: $A - \mathit{insert}\ a\ B = A - B - \{a\}$
— NOT SUITABLE FOR REWRITING since $\{a\} == \mathit{insert}\ a\ 0$
⟨*proof*⟩

lemma *Diff-insert2*: $A - \mathit{insert}\ a\ B = A - \{a\} - B$
— NOT SUITABLE FOR REWRITING since $\{a\} == \mathit{insert}\ a\ 0$
⟨*proof*⟩

lemma *insert-Diff-if*: $\mathit{insert}\ x\ A - B = (\mathit{if}\ x \in B\ \mathit{then}\ A - B\ \mathit{else}\ \mathit{insert}\ x\ (A - B))$
⟨*proof*⟩

lemma *insert-Diff1* [*simp*]: $x \in B \Longrightarrow \mathit{insert}\ x\ A - B = A - B$
⟨*proof*⟩

lemma *insert-Diff-single*[*simp*]: $\mathit{insert}\ a\ (A - \{a\}) = \mathit{insert}\ a\ A$
⟨*proof*⟩

lemma *insert-Diff*: $a \in A \Longrightarrow \mathit{insert}\ a\ (A - \{a\}) = A$
⟨*proof*⟩

lemma *Diff-insert-absorb*: $x \notin A \Longrightarrow (\mathit{insert}\ x\ A) - \{x\} = A$
⟨*proof*⟩

lemma *Diff-disjoint* [*simp*]: $A \cap (B - A) = \{\}$
⟨*proof*⟩

lemma *Diff-partition*: $A \subseteq B \Longrightarrow A \cup (B - A) = B$
⟨*proof*⟩

lemma *double-diff*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow B - (C - A) = A$
⟨*proof*⟩

lemma *Un-Diff-cancel* [*simp*]: $A \cup (B - A) = A \cup B$
⟨*proof*⟩

lemma *Un-Diff-cancel2* [*simp*]: $(B - A) \cup A = B \cup A$
⟨*proof*⟩

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$
⟨*proof*⟩

lemma *Diff-Int*: $A - (B \cap C) = (A - B) \cup (A - C)$
⟨*proof*⟩

lemma *Un-Diff*: $(A \cup B) - C = (A - C) \cup (B - C)$

⟨*proof*⟩

lemma *Int-Diff*: $(A \cap B) - C = A \cap (B - C)$
⟨*proof*⟩

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$
⟨*proof*⟩

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
⟨*proof*⟩

lemma *Diff-Compl* [*simp*]: $A - (- B) = A \cap B$
⟨*proof*⟩

lemma *Compl-Diff-eq* [*simp*]: $- (A - B) = -A \cup B$
⟨*proof*⟩

Quantification over type *bool*.

lemma *bool-induct*: $P\ True \Longrightarrow P\ False \Longrightarrow P\ x$
⟨*proof*⟩

lemma *all-bool-eq*: $(\forall b.\ P\ b) \longleftrightarrow P\ True \wedge P\ False$
⟨*proof*⟩

lemma *bool-contrapos*: $P\ x \Longrightarrow \neg\ P\ False \Longrightarrow P\ True$
⟨*proof*⟩

lemma *ex-bool-eq*: $(\exists b.\ P\ b) \longleftrightarrow P\ True \vee P\ False$
⟨*proof*⟩

Pow

lemma *Pow-empty* [*simp*]: $Pow\ \{\} = \{\{\}\}$
⟨*proof*⟩

lemma *Pow-insert*: $Pow\ (insert\ a\ A) = Pow\ A \cup (insert\ a\ `\ Pow\ A)$
⟨*proof*⟩

lemma *Pow-Compl*: $Pow\ (-\ A) = \{-B \mid B.\ A \in Pow\ B\}$
⟨*proof*⟩

lemma *Pow-UNIV* [*simp*]: $Pow\ UNIV = UNIV$
⟨*proof*⟩

lemma *Un-Pow-subset*: $Pow\ A \cup Pow\ B \subseteq Pow\ (A \cup B)$
⟨*proof*⟩

lemma *Pow-Int-eq* [*simp*]: $Pow\ (A \cap B) = Pow\ A \cap Pow\ B$
⟨*proof*⟩

Miscellany.

lemma *set-eq-subset*: $(A = B) = (A \subseteq B \ \& \ B \subseteq A)$
⟨*proof*⟩

lemma *subset-iff*: $(A \subseteq B) = (\forall t.\ t \in A \longrightarrow t \in B)$
⟨*proof*⟩

lemma *subset-iff-psubset-eq*: $(A \subseteq B) = ((A \subset B) \mid (A = B))$
⟨*proof*⟩

lemma *all-not-in-conv* [*simp*]: $(\forall x.\ x \notin A) = (A = \{\})$
⟨*proof*⟩

lemma *ex-in-conv*: $(\exists x.\ x \in A) = (A \neq \{\})$
⟨*proof*⟩

lemma *distinct-lemma*: $f\ x \neq f\ y \Longrightarrow x \neq y$
⟨*proof*⟩

6.4.4 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \Longrightarrow f`A \subseteq f`B$
⟨*proof*⟩

lemma *Pow-mono*: $A \subseteq B \Longrightarrow Pow\ A \subseteq Pow\ B$
⟨*proof*⟩

lemma *insert-mono*: $C \subseteq D \Longrightarrow insert\ a\ C \subseteq insert\ a\ D$
⟨*proof*⟩

lemma *Un-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cup B \subseteq C \cup D$
⟨*proof*⟩

lemma *Int-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cap B \subseteq C \cap D$
⟨*proof*⟩

lemma *Diff-mono*: $A \subseteq C \Longrightarrow D \subseteq B \Longrightarrow A - B \subseteq C - D$
⟨*proof*⟩

lemma *Compl-anti-mono*: $A \subseteq B \Longrightarrow -B \subseteq -A$
⟨*proof*⟩

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B \Longrightarrow x \in A \longrightarrow x \in B$
⟨*proof*⟩

lemma *conj-mono*: $P1 \longrightarrow Q1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1\ \&\ P2) \longrightarrow (Q1\ \&\ Q2)$

⟨*proof*⟩

lemma *disj-mono*: *P1* −−> *Q1* ==> *P2* −−> *Q2* ==> (*P1* | *P2*) −−> (*Q1* | *Q2*)
⟨*proof*⟩

lemma *imp-mono*: *Q1* −−> *P1* ==> *P2* −−> *Q2* ==> (*P1* −−> *P2*) −−> (*Q1* −−> *Q2*)
⟨*proof*⟩

lemma *imp-refl*: *P* −−> *P* ⟨*proof*⟩

lemma *not-mono*: *Q* −−> *P* ==> ~ *P* −−> ~ *Q*
⟨*proof*⟩

lemma *ex-mono*: (!!*x*. *P x* −−> *Q x*) ==> (*EX x*. *P x*) −−> (*EX x*. *Q x*)
⟨*proof*⟩

lemma *all-mono*: (!!*x*. *P x* −−> *Q x*) ==> (*ALL x*. *P x*) −−> (*ALL x*. *Q x*)
⟨*proof*⟩

lemma *Collect-mono*: (!!*x*. *P x* −−> *Q x*) ==> *Collect P* ⊆ *Collect Q*
⟨*proof*⟩

lemma *Int-Collect-mono*:
A ⊆ *B* ==> (!!*x*. *x* ∈ *A* ==> *P x* −−> *Q x*) ==> *A* ∩ *Collect P* ⊆ *B* ∩ *Collect Q*
⟨*proof*⟩

lemmas *basic-monos* =
subset-refl imp-refl disj-mono conj-mono
ex-mono Collect-mono in-mono

lemma *eq-to-mono*: *a* = *b* ==> *c* = *d* ==> *b* −−> *d* ==> *a* −−> *c*
⟨*proof*⟩

6.4.5 Inverse image of a function

definition *vimage* :: ('*a* => '*b*) => '*b set* => '*a set* (**infixr** −‘ *90*) **where**
[*code del*]: *f* −‘ *B* == {*x*. *f x* : *B*}

lemma *vimage-eq* [*simp*]: (*a* : *f* −‘ *B*) = (*f a* : *B*)
⟨*proof*⟩

lemma *vimage-singleton-eq*: (*a* : *f* −‘ {*b*}) = (*f a* = *b*)
⟨*proof*⟩

lemma *vimageI* [*intro*]: *f a* = *b* ==> *b*:*B* ==> *a* : *f* −‘ *B*
⟨*proof*⟩

lemma *vimageI2*: $f\ a : A \Longrightarrow a : f\ -`\ A$
⟨*proof*⟩

lemma *vimageE* [*elim*!]: a: $f\ -`\ B \Longrightarrow (!!x.\ f\ a = x \Longrightarrow x{:}B \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *vimageD*: $a : f\ -`\ A \Longrightarrow f\ a : A$
⟨*proof*⟩

lemma *vimage-empty* [*simp*]: $f\ -`\ \{\} = \{\}$
⟨*proof*⟩

lemma *vimage-Compl*: $f\ -`\ (-A) = -(f\ -`\ A)$
⟨*proof*⟩

lemma *vimage-Un* [*simp*]: $f\ -`\ (A\ Un\ B) = (f\ -`\ A)\ Un\ (f\ -`\ B)$
⟨*proof*⟩

lemma *vimage-Int* [*simp*]: $f\ -`\ (A\ Int\ B) = (f\ -`\ A)\ Int\ (f\ -`\ B)$
⟨*proof*⟩

lemma *vimage-Collect-eq* [*simp*]: $f\ -`\ Collect\ P = \{y.\ P\ (f\ y)\}$
⟨*proof*⟩

lemma *vimage-Collect*: $(!!x.\ P\ (f\ x) = Q\ x) \Longrightarrow f\ -`\ (Collect\ P) = Collect\ Q$
⟨*proof*⟩

lemma *vimage-insert*: $f-`(insert\ a\ B) = (f-`\{a\})\ Un\ (f-`B)$
— NOT suitable for rewriting because of the recurrence of $\{a\}$.
⟨*proof*⟩

lemma *vimage-Diff*: $f\ -`\ (A - B) = (f\ -`\ A) - (f\ -`\ B)$
⟨*proof*⟩

lemma *vimage-UNIV* [*simp*]: $f\ -`\ UNIV = UNIV$
⟨*proof*⟩

lemma *vimage-mono*: $A \subseteq B \Longrightarrow f\ -`\ A \subseteq f\ -`\ B$
— monotonicity
⟨*proof*⟩

lemma *vimage-image-eq* [*no-atp*]: $f\ -`\ (f\ `\ A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
⟨*proof*⟩

lemma *image-vimage-subset*: $f\ `\ (f\ -`\ A) <= A$
⟨*proof*⟩

lemma *image-vimage-eq* [*simp*]: f ‘ $(f -‘ A) = A$ *Int range* f
⟨*proof*⟩

lemma *vimage-const* [*simp*]: $((\lambda x.\ c) -‘ A) = (\mathit{if}\ c \in A\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
⟨*proof*⟩

lemma *vimage-if* [*simp*]: $((\lambda x.\ \mathit{if}\ x \in B\ \mathit{then}\ c\ \mathit{else}\ d) -‘ A) =$
$(\mathit{if}\ c \in A\ \mathit{then}\ (\mathit{if}\ d \in A\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ B)$
$\mathit{else\ if}\ d \in A\ \mathit{then} -B\ \mathit{else}\ \{\})$
⟨*proof*⟩

lemma *vimage-inter-cong*:
$(\bigwedge w.\ w \in S \Longrightarrow f\ w = g\ w) \Longrightarrow f -‘ y \cap S = g -‘ y \cap S$
⟨*proof*⟩

lemma *image-Int-subset*: $f‘(A\ \mathit{Int}\ B) <= f‘A\ \mathit{Int}\ f‘B$
⟨*proof*⟩

lemma *image-diff-subset*: $f‘A - f‘B <= f‘(A - B)$
⟨*proof*⟩

6.4.6 Getting the Contents of a Singleton Set

definition *contents* :: $'a\ set \Rightarrow 'a$ **where**
[*code del*]: $\mathit{contents}\ X = (\mathit{THE}\ x.\ X = \{x\})$

lemma *contents-eq* [*simp*]: $\mathit{contents}\ \{x\} = x$
⟨*proof*⟩

6.4.7 Least value operator

lemma *Least-mono*:
$\mathit{mono}\ (f::'a::\mathit{order} => 'b::\mathit{order}) ==> EX\ x:S.\ ALL\ y:S.\ x <= y$
$==> (\mathit{LEAST}\ y.\ y : f ‘ S) = f\ (\mathit{LEAST}\ x.\ x : S)$
— Courtesy of Stephan Merz
⟨*proof*⟩

6.5 Misc

Rudimentary code generation

lemma *insert-code* [*code*]: $\mathit{insert}\ y\ A\ x \longleftrightarrow y = x \vee A\ x$
⟨*proof*⟩

lemma *vimage-code* [*code*]: $(f -‘ A)\ x = A\ (f\ x)$
⟨*proof*⟩

Misc theorem and ML bindings

lemmas *equalityI* = *subset-antisym*

⟨*ML*⟩

end

7 Typedef: HOL type definitions

theory *Typedef*
imports *Set*
uses
(*Tools/typedef.ML*)
(*Tools/typecopy.ML*)
(*Tools/typedef-codegen.ML*)
begin

⟨*ML*⟩

locale *type-definition* =
fixes *Rep* **and** *Abs* **and** *A*
assumes *Rep*: *Rep x* ∈ *A*
and *Rep-inverse*: *Abs* (*Rep x*) = *x*
and *Abs-inverse*: *y* ∈ *A* ==> *Rep* (*Abs y*) = *y*
— This will be axiomatized for each typedef!
begin

lemma *Rep-inject*:
(*Rep x* = *Rep y*) = (*x* = *y*)
⟨*proof*⟩

lemma *Abs-inject*:
assumes *x*: *x* ∈ *A* **and** *y*: *y* ∈ *A*
shows (*Abs x* = *Abs y*) = (*x* = *y*)
⟨*proof*⟩

lemma *Rep-cases* [*cases set*]:
assumes *y*: *y* ∈ *A*
and *hyp*: !!*x*. *y* = *Rep x* ==> *P*
shows *P*
⟨*proof*⟩

lemma *Abs-cases* [*cases type*]:
assumes *r*: !!*y*. *x* = *Abs y* ==> *y* ∈ *A* ==> *P*
shows *P*
⟨*proof*⟩

lemma *Rep-induct* [*induct set*]:
assumes *y*: *y* ∈ *A*
and *hyp*: !!*x*. *P* (*Rep x*)
shows *P y*

⟨*proof*⟩

lemma *Abs-induct* [*induct type*]:
  assumes *r*: !!*y*. *y* ∈ *A* ==> *P* (*Abs y*)
  shows *P x*
⟨*proof*⟩

lemma *Rep-range*: *range Rep* = *A*
⟨*proof*⟩

lemma *Abs-image*: *Abs* ‘ *A* = *UNIV*
⟨*proof*⟩

end

⟨*ML*⟩

end

8 Complete-Lattice: Complete lattices, with special focus on sets

theory *Complete-Lattice*
imports *Set*
begin

notation
  less-eq (**infix** ⊑ *50*) **and**
  less (**infix** ⊏ *50*) **and**
  inf (**infixl** ⊓ *70*) **and**
  sup (**infixl** ⊔ *65*) **and**
  top (⊤) **and**
  bot (⊥)

8.1 Syntactic infimum and supremum operations

class *Inf* =
  fixes *Inf* :: *'a set* ⇒ *'a* (⨅- [*900*] *900*)

class *Sup* =
  fixes *Sup* :: *'a set* ⇒ *'a* (⨆- [*900*] *900*)

8.2 Abstract complete lattices

class *complete-lattice* = *bounded-lattice* + *Inf* + *Sup* +
  assumes *Inf-lower*: *x* ∈ *A* ⟹ ⨅*A* ⊑ *x*
    and *Inf-greatest*: (⋀*x*. *x* ∈ *A* ⟹ *z* ⊑ *x*) ⟹ *z* ⊑ ⨅*A*
  assumes *Sup-upper*: *x* ∈ *A* ⟹ *x* ⊑ ⨆*A*

and *Sup-least*: $(\bigwedge x.\ x \in A \Longrightarrow x \sqsubseteq z) \Longrightarrow \bigsqcup A \sqsubseteq z$
begin

lemma *dual-complete-lattice*:
class.complete-lattice Sup Inf $(op \geq)$ $(op >)$ $(op \sqcup)$ $(op \sqcap)$ $\top$ $\bot$
⟨*proof*⟩

lemma *Inf-Sup*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \sqsubseteq a\}$
⟨*proof*⟩

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \sqsubseteq b\}$
⟨*proof*⟩

lemma *Inf-empty*:
$\bigsqcap \{\} = \top$
⟨*proof*⟩

lemma *Sup-empty*:
$\bigsqcup \{\} = \bot$
⟨*proof*⟩

lemma *Inf-insert*: $\bigsqcap insert\ a\ A = a \sqcap \bigsqcap A$
⟨*proof*⟩

lemma *Sup-insert*: $\bigsqcup insert\ a\ A = a \sqcup \bigsqcup A$
⟨*proof*⟩

lemma *Inf-singleton* [*simp*]:
$\bigsqcap \{a\} = a$
⟨*proof*⟩

lemma *Sup-singleton* [*simp*]:
$\bigsqcup \{a\} = a$
⟨*proof*⟩

lemma *Inf-binary*:
$\bigsqcap \{a,\ b\} = a \sqcap b$
⟨*proof*⟩

lemma *Sup-binary*:
$\bigsqcup \{a,\ b\} = a \sqcup b$
⟨*proof*⟩

lemma *Inf-UNIV*:
$\bigsqcap UNIV = bot$
⟨*proof*⟩

lemma *Sup-UNIV*:
$\bigsqcup UNIV = top$

⟨*proof*⟩

lemma *Sup-le-iff*: *Sup A* ⊑ *b* ⟷ (∀ *a*∈*A*. *a* ⊑ *b*)
⟨*proof*⟩

lemma *le-Inf-iff*: *b* ⊑ *Inf A* ⟷ (∀ *a*∈*A*. *b* ⊑ *a*)
⟨*proof*⟩

definition *SUPR* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
SUPR A f = ⨆ (*f* ‘ *A*)

definition *INFI* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
INFI A f = ⨅ (*f* ‘ *A*)

end

syntax
-SUP1 :: *pttrns* => *'b* => *'b* ((*3SUP -./ -*) [*0*, *10*] *10*)
-SUP :: *pttrn* => *'a set* => *'b* => *'b* ((*3SUP -:-./ -*) [*0*, *0*, *10*] *10*)
-INF1 :: *pttrns* => *'b* => *'b* ((*3INF -./ -*) [*0*, *10*] *10*)
-INF :: *pttrn* => *'a set* => *'b* => *'b* ((*3INF -:-./ -*) [*0*, *0*, *10*] *10*)

translations
SUP x y. B == *SUP x. SUP y. B*
SUP x. B == *CONST SUPR CONST UNIV* (%*x. B*)
SUP x. B == *SUP x*:*CONST UNIV. B*
SUP x:*A. B* == *CONST SUPR A* (%*x. B*)
INF x y. B == *INF x. INF y. B*
INF x. B == *CONST INFI CONST UNIV* (%*x. B*)
INF x. B == *INF x*:*CONST UNIV. B*
INF x:*A. B* == *CONST INFI A* (%*x. B*)

⟨*ML*⟩

context *complete-lattice*
begin

lemma *le-SUPI*: *i* : *A* ⟹ *M i* ⊑ (*SUP i*:*A. M i*)
⟨*proof*⟩

lemma *SUP-leI*: (⋀*i. i* : *A* ⟹ *M i* ⊑ *u*) ⟹ (*SUP i*:*A. M i*) ⊑ *u*
⟨*proof*⟩

lemma *INF-leI*: *i* : *A* ⟹ (*INF i*:*A. M i*) ⊑ *M i*
⟨*proof*⟩

lemma *le-INFI*: (⋀*i. i* : *A* ⟹ *u* ⊑ *M i*) ⟹ *u* ⊑ (*INF i*:*A. M i*)
⟨*proof*⟩

lemma *SUP-le-iff*: $(SUP\ i{:}A.\ M\ i) \sqsubseteq u \longleftrightarrow (\forall\, i \in A.\ M\ i \sqsubseteq u)$
⟨*proof*⟩

lemma *le-INF-iff*: $u \sqsubseteq (INF\ i{:}A.\ M\ i) \longleftrightarrow (\forall\, i \in A.\ u \sqsubseteq M\ i)$
⟨*proof*⟩

lemma *SUP-const*[*simp*]: $A \neq \{\} \Longrightarrow (SUP\ i{:}A.\ M) = M$
⟨*proof*⟩

lemma *INF-const*[*simp*]: $A \neq \{\} \Longrightarrow (INF\ i{:}A.\ M) = M$
⟨*proof*⟩

end

8.3 *bool* and - ⇒ - as complete lattice

instantiation *bool* :: *complete-lattice*
begin

definition
Inf-bool-def: $\bigsqcap A \longleftrightarrow (\forall\, x{\in}A.\ x)$

definition
Sup-bool-def: $\bigsqcup A \longleftrightarrow (\exists\, x{\in}A.\ x)$

instance ⟨*proof*⟩

end

lemma *Inf-empty-bool* [*simp*]:
$\bigsqcap \{\}$
⟨*proof*⟩

lemma *not-Sup-empty-bool* [*simp*]:
$\neg \bigsqcup \{\}$
⟨*proof*⟩

lemma *INFI-bool-eq*:
INFI = *Ball*
⟨*proof*⟩

lemma *SUPR-bool-eq*:
SUPR = *Bex*
⟨*proof*⟩

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

definition

Inf-fun-def [*code del*]: $\bigsqcap A = (\lambda x.\ \bigsqcap \{y.\ \exists f{\in}A.\ y = f\,x\})$

definition
Sup-fun-def [*code del*]: $\bigsqcup A = (\lambda x.\ \bigsqcup \{y.\ \exists f{\in}A.\ y = f\,x\})$

instance ⟨*proof*⟩

end

lemma *Inf-empty-fun*:
$\bigsqcap \{\} = (\lambda\text{-}.\ \bigsqcap \{\})$
⟨*proof*⟩

lemma *Sup-empty-fun*:
$\bigsqcup \{\} = (\lambda\text{-}.\ \bigsqcup \{\})$
⟨*proof*⟩

8.4 Union

abbreviation *Union* :: *'a set set* ⇒ *'a set* **where**
Union S ≡ $\bigsqcup S$

notation (*xsymbols*)
Union ($\bigcup$- [*90*] *90*)

lemma *Union-eq*:
$\bigcup A = \{x.\ \exists B \in A.\ x \in B\}$
⟨*proof*⟩

lemma *Union-iff* [*simp*, *no-atp*]:
$A \in \bigcup C \longleftrightarrow (\exists X{\in}C.\ A{\in}X)$
⟨*proof*⟩

lemma *UnionI* [*intro*]:
$X \in C \Longrightarrow A \in X \Longrightarrow A \in \bigcup C$
— The order of the premises presupposes that C is rigid; A may be flexible.
⟨*proof*⟩

lemma *UnionE* [*elim*!]:
$A \in \bigcup C \Longrightarrow (\bigwedge X.\ A{\in}X \Longrightarrow X{\in}C \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *Union-upper*: $B \in A \Longrightarrow B \subseteq$ *Union A*
⟨*proof*⟩

lemma *Union-least*: $(!!X.\ X \in A \Longrightarrow X \subseteq C) \Longrightarrow$ *Union A* $\subseteq C$
⟨*proof*⟩

lemma *Un-eq-Union*: $A \cup B = \bigcup \{A, B\}$

⟨*proof*⟩

lemma *Union-empty* [*simp*]: *Union*({}) = {}
⟨*proof*⟩

lemma *Union-UNIV* [*simp*]: *Union UNIV* = *UNIV*
⟨*proof*⟩

lemma *Union-insert* [*simp*]: *Union* (*insert a B*) = *a* ∪ ⋃ *B*
⟨*proof*⟩

lemma *Union-Un-distrib* [*simp*]: ⋃(*A Un B*) = ⋃ *A* ∪ ⋃ *B*
⟨*proof*⟩

lemma *Union-Int-subset*: ⋃(*A* ∩ *B*) ⊆ ⋃ *A* ∩ ⋃ *B*
⟨*proof*⟩

lemma *Union-empty-conv* [*simp*,*no-atp*]: (⋃ *A* = {}) = (∀ *x*∈*A*. *x* = {})
⟨*proof*⟩

lemma *empty-Union-conv* [*simp*,*no-atp*]: ({} = ⋃ *A*) = (∀ *x*∈*A*. *x* = {})
⟨*proof*⟩

lemma *Union-disjoint*: (⋃ *C* ∩ *A* = {}) = (∀ *B*∈*C*. *B* ∩ *A* = {})
⟨*proof*⟩

lemma *subset-Pow-Union*: *A* ⊆ *Pow* (⋃ *A*)
⟨*proof*⟩

lemma *Union-Pow-eq* [*simp*]: ⋃(*Pow A*) = *A*
⟨*proof*⟩

lemma *Union-mono*: *A* ⊆ *B* ==> ⋃ *A* ⊆ ⋃ *B*
⟨*proof*⟩

8.5 Unions of families

abbreviation *UNION* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *'b set* **where**
UNION ≡ *SUPR*

syntax
-UNION1 :: *pttrns* => *'b set* => *'b set* ((*3UN -./ -*) [*0*, *10*] *10*)
-UNION :: *pttrn* => *'a set* => *'b set* => *'b set* ((*3UN -:-./ -*) [*0*, *0*, *10*] *10*)

syntax (*xsymbols*)
-UNION1 :: *pttrns* => *'b set* => *'b set* ((*3⋃ -./ -*) [*0*, *10*] *10*)
-UNION :: *pttrn* => *'a set* => *'b set* => *'b set* ((*3⋃ -∈-./ -*) [*0*, *0*, *10*] *10*)

syntax (*latex* **output**)
-UNION1 :: *pttrns => 'b set => 'b set* (($3\bigcup(00\_)/\ \_)$ [*0*, *10*] *10*)
-UNION :: *pttrn => 'a set => 'b set => 'b set* (($3\bigcup(00\_{\in}\_)/\ \_)$ [*0*, *0*, *10*] *10*)

translations
UN x y. B == *UN x. UN y. B*
UN x. B == *CONST UNION CONST UNIV* (%*x. B*)
UN x. B == *UN x:CONST UNIV. B*
UN x:A. B == *CONST UNION A* (%*x. B*)

Note the difference between ordinary xsymbol syntax of indexed unions and intersections (e.g. $\bigcup a_1{\in}A_1.\ B$) and their LaTeX rendition: $\bigcup_{a_1 \in A_1} B$. The former does not make the index expression a subscript of the union/intersection symbol because this leads to problems with nested subscripts in Proof General.

⟨*ML*⟩

lemma *UNION-eq-Union-image*:
$(\bigcup x{\in}A.\ B\ x) = \bigcup(B`A)$
⟨*proof*⟩

lemma *Union-def*:
$\bigcup S = (\bigcup x{\in}S.\ x)$
⟨*proof*⟩

lemma *UNION-def* [*no-atp*]:
$(\bigcup x{\in}A.\ B\ x) = \{y.\ \exists x{\in}A.\ y \in B\ x\}$
⟨*proof*⟩

lemma *Union-image-eq* [*simp*]:
$\bigcup(B`A) = (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *UN-iff* [*simp*]: (*b*: (*UN x*:*A*. *B x*)) = (*EX x*:*A*. *b*: *B x*)
⟨*proof*⟩

lemma *UN-I* [*intro*]: *a*:*A* ==> *b*: *B a* ==> *b*: (*UN x*:*A*. *B x*)
— The order of the premises presupposes that *A* is rigid; *b* may be flexible.
⟨*proof*⟩

lemma *UN-E* [*elim*!]: *b* : (*UN x*:*A*. *B x*) ==> (!!*x*. *x*:*A* ==> *b*: *B x* ==> *R*) ==> *R*
⟨*proof*⟩

lemma *UN-cong* [*cong*]:
A = *B* ==> (!!*x*. *x*:*B* ==> *C x* = *D x*) ==> (*UN x*:*A*. *C x*) = (*UN x*:*B*. *D x*)
⟨*proof*⟩

lemma *strong-UN-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B =simp\Rightarrow C\ x = D\ x) \Longrightarrow (UN\ x{:}A.\ C\ x) = (UN\ x{:}B.\ D\ x)$
⟨*proof*⟩

lemma *image-eq-UN*: $f`A = (UN\ x{:}A.\ \{f\ x\})$
⟨*proof*⟩

lemma *UN-upper*: $a \in A \Longrightarrow B\ a \subseteq (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *UN-least*: $(!!x.\ x \in A \Longrightarrow B\ x \subseteq C) \Longrightarrow (\bigcup x{\in}A.\ B\ x) \subseteq C$
⟨*proof*⟩

lemma *Collect-bex-eq* [*no-atp*]: $\{x.\ \exists y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
⟨*proof*⟩

lemma *UN-insert-distrib*: $u \in A \Longrightarrow (\bigcup x{\in}A.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *UN-empty* [*simp*,*no-atp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
⟨*proof*⟩

lemma *UN-empty2* [*simp*]: $(\bigcup x{\in}A.\ \{\}) = \{\}$
⟨*proof*⟩

lemma *UN-singleton* [*simp*]: $(\bigcup x{\in}A.\ \{x\}) = A$
⟨*proof*⟩

lemma *UN-absorb*: $k \in I \Longrightarrow A\ k \cup (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ A\ i)$
⟨*proof*⟩

lemma *UN-insert* [*simp*]: $(\bigcup x{\in}insert\ a\ A.\ B\ x) = B\ a \cup UNION\ A\ B$
⟨*proof*⟩

lemma *UN-Un*[*simp*]: $(\bigcup i \in A \cup B.\ M\ i) = (\bigcup i{\in}A.\ M\ i) \cup (\bigcup i{\in}B.\ M\ i)$
⟨*proof*⟩

lemma *UN-UN-flatten*: $(\bigcup x \in (\bigcup y{\in}A.\ B\ y).\ C\ x) = (\bigcup y{\in}A.\ \bigcup x{\in}B\ y.\ C\ x)$
⟨*proof*⟩

lemma *UN-subset-iff*: $((\bigcup i{\in}I.\ A\ i) \subseteq B) = (\forall i{\in}I.\ A\ i \subseteq B)$
⟨*proof*⟩

lemma *image-Union*: $f\ `\ \bigcup S = (\bigcup x{\in}S.\ f\ `\ x)$
⟨*proof*⟩

lemma *UN-constant* [*simp*]: $(\bigcup y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{\}\ \mathit{else}\ c)$
⟨*proof*⟩

lemma *UN-eq*: $(\bigcup x{\in}A.\ B\ x) = \bigcup(\{\,Y.\ \exists x{\in}A.\ Y = B\ x\})$
⟨*proof*⟩

lemma *UNION-empty-conv*[*simp*]:
$(\{\} = (\mathit{UN}\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = \{\})$
$((\mathit{UN}\ x{:}A.\ B\ x) = \{\}) = (\forall x{\in}A.\ B\ x = \{\})$
⟨*proof*⟩

lemma *Collect-ex-eq* [*no-atp*]: $\{x.\ \exists y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
⟨*proof*⟩

lemma *ball-UN*: $(\forall z \in \mathit{UNION}\ A\ B.\ P\ z) = (\forall x{\in}A.\ \forall z \in B\ x.\ P\ z)$
⟨*proof*⟩

lemma *bex-UN*: $(\exists z \in \mathit{UNION}\ A\ B.\ P\ z) = (\exists x{\in}A.\ \exists z{\in}B\ x.\ P\ z)$
⟨*proof*⟩

lemma *Un-eq-UN*: $A \cup B = (\bigcup b.\ \mathit{if}\ b\ \mathit{then}\ A\ \mathit{else}\ B)$
⟨*proof*⟩

lemma *UN-bool-eq*: $(\bigcup b{::}\mathit{bool}.\ A\ b) = (A\ \mathit{True} \cup A\ \mathit{False})$
⟨*proof*⟩

lemma *UN-Pow-subset*: $(\bigcup x{\in}A.\ \mathit{Pow}\ (B\ x)) \subseteq \mathit{Pow}\ (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *UN-mono*:
$A \subseteq B \Longrightarrow (!!x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcup x{\in}A.\ f\ x) \subseteq (\bigcup x{\in}B.\ g\ x)$
⟨*proof*⟩

lemma *vimage-Union*: $f\ -`\ (\mathit{Union}\ A) = (\mathit{UN}\ X{:}A.\ f\ -`\ X)$
⟨*proof*⟩

lemma *vimage-UN*: $f-`(\mathit{UN}\ x{:}A.\ B\ x) = (\mathit{UN}\ x{:}A.\ f\ -`\ B\ x)$
⟨*proof*⟩

lemma *vimage-eq-UN*: $f-`B = (\mathit{UN}\ y{:}\ B.\ f-`\{y\})$
— NOT suitable for rewriting
⟨*proof*⟩

lemma *image-UN*: $(f\ `\ (\mathit{UNION}\ A\ B)) = (\mathit{UN}\ x{:}A.(f\ `\ (B\ x)))$
⟨*proof*⟩

8.6 Inter

abbreviation *Inter* :: *'a set set* ⇒ *'a set* **where**
Inter S ≡ ⋂ *S*

notation (*xsymbols*)
Inter (⋂- [*90*] *90*)

lemma *Inter-eq* [*code del*]:
⋂ *A* = {*x*. ∀ *B* ∈ *A*. *x* ∈ *B*}
⟨*proof*⟩

lemma *Inter-iff* [*simp*,*no-atp*]: (*A* : *Inter C*) = (*ALL X*:*C*. *A*:*X*)
⟨*proof*⟩

lemma *InterI* [*intro*!]: (!!*X*. *X*:*C* ==> *A*:*X*) ==> *A* : *Inter C*
⟨*proof*⟩

A "destruct" rule – every X in C contains A as an element, but $A \in X$ 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* ∈ *C*.
⟨*proof*⟩

lemma *Inter-lower*: *B* ∈ *A* ==> *Inter A* ⊆ *B*
⟨*proof*⟩

lemma *Inter-subset*:
[| !!*X*. *X* ∈ *A* ==> *X* ⊆ *B*; *A* ~= {} |] ==> ⋂ *A* ⊆ *B*
⟨*proof*⟩

lemma *Inter-greatest*: (!!*X*. *X* ∈ *A* ==> *C* ⊆ *X*) ==> *C* ⊆ *Inter A*
⟨*proof*⟩

lemma *Int-eq-Inter*: *A* ∩ *B* = ⋂{*A*, *B*}
⟨*proof*⟩

lemma *Inter-empty* [*simp*]: ⋂{} = *UNIV*
⟨*proof*⟩

lemma *Inter-UNIV* [*simp*]: ⋂ *UNIV* = {}
⟨*proof*⟩

lemma *Inter-insert* [*simp*]: ⋂(*insert a B*) = *a* ∩ ⋂ *B*
⟨*proof*⟩

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$
⟨*proof*⟩

lemma *Inter-Un-distrib*: $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$
⟨*proof*⟩

lemma *Inter-UNIV-conv* [*simp*,*no-atp*]:
$(\bigcap A = UNIV) = (\forall x{\in}A.\ x = UNIV)$
$(UNIV = \bigcap A) = (\forall x{\in}A.\ x = UNIV)$
⟨*proof*⟩

lemma *Inter-anti-mono*: $B \subseteq A \Longrightarrow \bigcap A \subseteq \bigcap B$
⟨*proof*⟩

8.7 Intersections of families

abbreviation *INTER* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *'b set* **where**
INTER ≡ *INFI*

syntax
-INTER1 :: *pttrns* => *'b set* => *'b set* (*(3INT -./ -) [0, 10] 10*)
-INTER :: *pttrn* => *'a set* => *'b set* => *'b set* (*(3INT -:-./ -) [0, 0, 10] 10*)

syntax (*xsymbols*)
-INTER1 :: *pttrns* => *'b set* => *'b set* (*(3⋂-./ -) [0, 10] 10*)
-INTER :: *pttrn* => *'a set* => *'b set* => *'b set* (*(3⋂-∈-./ -) [0, 0, 10] 10*)

syntax (*latex* **output**)
-INTER1 :: *pttrns* => *'b set* => *'b set* (*(3⋂(00_)/ -) [0, 10] 10*)
-INTER :: *pttrn* => *'a set* => *'b set* => *'b set* (*(3⋂(00_-∈-)/ -) [0, 0, 10] 10*)

translations
INT x y. B == *INT x. INT y. B*
INT x. B == *CONST INTER CONST UNIV* (%*x. B*)
INT x. B == *INT x*:*CONST UNIV. B*
INT x:*A. B* == *CONST INTER A* (%*x. B*)

⟨*ML*⟩

lemma *INTER-eq-Inter-image*:
$(\bigcap x{\in}A.\ B\ x) = \bigcap (B`A)$
⟨*proof*⟩

lemma *Inter-def*:
$\bigcap S = (\bigcap x{\in}S.\ x)$
⟨*proof*⟩

lemma *INTER-def*:
$(\bigcap x{\in}A.\ B\ x) = \{y.\ \forall x{\in}A.\ y \in B\ x\}$
⟨*proof*⟩

lemma *Inter-image-eq* [*simp*]:
$\bigcap(B`A) = (\bigcap x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *INT-iff* [*simp*]: $(b{:}\ (INT\ x{:}A.\ B\ x)) = (ALL\ x{:}A.\ b{:}\ B\ x)$
⟨*proof*⟩

lemma *INT-I* [*intro!*]: $(!!x.\ x{:}A \Longrightarrow b{:}\ B\ x) \Longrightarrow b : (INT\ x{:}A.\ B\ x)$
⟨*proof*⟩

lemma *INT-D* [*elim*]: $b : (INT\ x{:}A.\ B\ x) \Longrightarrow a{:}A \Longrightarrow b{:}\ B\ a$
⟨*proof*⟩

lemma *INT-E* [*elim*]: $b : (INT\ x{:}A.\ B\ x) \Longrightarrow (b{:}\ B\ a \Longrightarrow R) \Longrightarrow (a^{\sim}{:}A \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination – by the Excluded Middle on $a \in A$.
⟨*proof*⟩

lemma *INT-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (INT\ x{:}A.\ C\ x) = (INT\ x{:}B.\ D\ x)$
⟨*proof*⟩

lemma *Collect-ball-eq*: $\{x.\ \forall y{\in}A.\ P\ x\ y\} = (\bigcap y{\in}A.\ \{x.\ P\ x\ y\})$
⟨*proof*⟩

lemma *Collect-all-eq*: $\{x.\ \forall y.\ P\ x\ y\} = (\bigcap y.\ \{x.\ P\ x\ y\})$
⟨*proof*⟩

lemma *INT-lower*: $a \in A \Longrightarrow (\bigcap x{\in}A.\ B\ x) \subseteq B\ a$
⟨*proof*⟩

lemma *INT-greatest*: $(!!x.\ x \in A \Longrightarrow C \subseteq B\ x) \Longrightarrow C \subseteq (\bigcap x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *INT-empty* [*simp*]: $(\bigcap x{\in}\{\}.\ B\ x) = UNIV$
⟨*proof*⟩

lemma *INT-absorb*: $k \in I \Longrightarrow A\ k \cap (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ A\ i)$
⟨*proof*⟩

lemma *INT-subset-iff*: $(B \subseteq (\bigcap i{\in}I.\ A\ i)) = (\forall i{\in}I.\ B \subseteq A\ i)$
⟨*proof*⟩

lemma *INT-insert* [*simp*]: $(\bigcap x \in insert\ a\ A.\ B\ x) = B\ a \cap INTER\ A\ B$
⟨*proof*⟩

lemma *INT-Un*: $(\bigcap i \in A \cup B.\ M\ i) = (\bigcap i \in A.\ M\ i) \cap (\bigcap i{\in}B.\ M\ i)$
⟨*proof*⟩

lemma *INT-insert-distrib*:
$u \in A \Longrightarrow (\bigcap x{\in}A.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcap x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *INT-constant* [*simp*]: $(\bigcap y{\in}A.\ c) = (if\ A = \{\}\ then\ UNIV\ else\ c)$
⟨*proof*⟩

lemma *INT-eq*: $(\bigcap x{\in}A.\ B\ x) = \bigcap(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
— Look: it has an *existential* quantifier
⟨*proof*⟩

lemma *INTER-UNIV-conv*[*simp*]:
$(UNIV = (INT\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = UNIV)$
$((INT\ x{:}A.\ B\ x) = UNIV) = (\forall x{\in}A.\ B\ x = UNIV)$
⟨*proof*⟩

lemma *INT-bool-eq*: $(\bigcap b{::}bool.\ A\ b) = (A\ True \cap A\ False)$
⟨*proof*⟩

lemma *Pow-INT-eq*: $Pow\ (\bigcap x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ Pow\ (B\ x))$
⟨*proof*⟩

lemma *INT-anti-mono*:
$B \subseteq A \Longrightarrow (!!x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcap x{\in}A.\ f\ x) \subseteq (\bigcap x{\in}A.\ g\ x)$
— The last inclusion is POSITIVE!
⟨*proof*⟩

lemma *vimage-INT*: $f{-}`(INT\ x{:}A.\ B\ x) = (INT\ x{:}A.\ f\ -`\ B\ x)$
⟨*proof*⟩

8.8 Distributive laws

lemma *Int-Union*: $A \cap \bigcup B = (\bigcup C{\in}B.\ A \cap C)$
⟨*proof*⟩

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C{\in}B.\ C \cap A)$
⟨*proof*⟩

lemma *Un-Union-image*: $(\bigcup x{\in}C.\ A\ x \cup B\ x) = \bigcup(A`C) \cup \bigcup(B`C)$
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
— Union of a family of unions
⟨*proof*⟩

lemma *UN-Un-distrib*: $(\bigcup i{\in}I.\ A\ i \cup B\ i) = (\bigcup i{\in}I.\ A\ i) \cup (\bigcup i{\in}I.\ B\ i)$
— Equivalent version
⟨*proof*⟩

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap C{\in}B.\ A \cup C)$
⟨*proof*⟩

lemma *Int-Inter-image*: $(\bigcap x{\in}C.\ A\ x \cap B\ x) = \bigcap (A`C) \cap \bigcap (B`C)$
⟨*proof*⟩

lemma *INT-Int-distrib*: $(\bigcap i{\in}I.\ A\ i \cap B\ i) = (\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i)$
— Equivalent version
⟨*proof*⟩

lemma *Int-UN-distrib*: $B \cap (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ B \cap A\ i)$
— Halmos, Naive Set Theory, page 35.
⟨*proof*⟩

lemma *Un-INT-distrib*: $B \cup (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ B \cup A\ i)$
⟨*proof*⟩

lemma *Int-UN-distrib2*: $(\bigcup i{\in}I.\ A\ i) \cap (\bigcup j{\in}J.\ B\ j) = (\bigcup i{\in}I.\ \bigcup j{\in}J.\ A\ i \cap B\ j)$
⟨*proof*⟩

lemma *Un-INT-distrib2*: $(\bigcap i{\in}I.\ A\ i) \cup (\bigcap j{\in}J.\ B\ j) = (\bigcap i{\in}I.\ \bigcap j{\in}J.\ A\ i \cup B\ j)$
⟨*proof*⟩

8.9 Complement

lemma *Compl-UN* [*simp*]: $-(\bigcup x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ -B\ x)$
⟨*proof*⟩

lemma *Compl-INT* [*simp*]: $-(\bigcap x{\in}A.\ B\ x) = (\bigcup x{\in}A.\ -B\ x)$
⟨*proof*⟩

8.10 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps* [*simp*]:
$!!a\ B\ C.\ (UN\ x{:}C.\ insert\ a\ (B\ x)) = (if\ C{=}\{\}\ then\ \{\}\ else\ insert\ a\ (UN\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Un\ B) = ((if\ C{=}\{\}\ then\ \{\}\ else\ (UN\ x{:}C.\ A\ x)\ Un\ B))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ Un\ B\ x) = ((if\ C{=}\{\}\ then\ \{\}\ else\ A\ Un\ (UN\ x{:}C.\ B\ x)))$

!!$A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Int\ B) = ((UN\ x{:}C.\ A\ x)\ Int\ B)$
!!$A\ B\ C.\ (UN\ x{:}C.\ A\ Int\ B\ x) = (A\ Int\ (UN\ x{:}C.\ B\ x))$
!!$A\ B\ C.\ (UN\ x{:}C.\ A\ x - B) = ((UN\ x{:}C.\ A\ x) - B)$
!!$A\ B\ C.\ (UN\ x{:}C.\ A - B\ x) = (A - (INT\ x{:}C.\ B\ x))$
!!$A\ B.\ (UN\ x{:}\ Union\ A.\ B\ x) = (UN\ y{:}A.\ UN\ x{:}y.\ B\ x)$
!!$A\ B\ C.\ (UN\ z{:}\ UNION\ A\ B.\ C\ z) = (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z)$
!!$A\ B\ f.\ (UN\ x{:}f`A.\ B\ x) = (UN\ a{:}A.\ B\ (f\ a))$
⟨*proof*⟩

lemma *INT-simps* [*simp*]:
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Int\ B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x)\ Int\ B)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ Int\ B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A\ Int\ (INT\ x{:}C.\ B\ x))$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x - B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x) - B)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A - B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A - (UN\ x{:}C.\ B\ x))$
!!$a\ B\ C.\ (INT\ x{:}C.\ insert\ a\ (B\ x)) = insert\ a\ (INT\ x{:}C.\ B\ x)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Un\ B) = ((INT\ x{:}C.\ A\ x)\ Un\ B)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ Un\ B\ x) = (A\ Un\ (INT\ x{:}C.\ B\ x))$
!!$A\ B.\ (INT\ x{:}\ Union\ A.\ B\ x) = (INT\ y{:}A.\ INT\ x{:}y.\ B\ x)$
!!$A\ B\ C.\ (INT\ z{:}\ UNION\ A\ B.\ C\ z) = (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z)$
!!$A\ B\ f.\ (INT\ x{:}f`A.\ B\ x) = (INT\ a{:}A.\ B\ (f\ a))$
⟨*proof*⟩

lemma *ball-simps* [*simp*,*no-atp*]:
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ |\ Q) = ((ALL\ x{:}A.\ P\ x)\ |\ Q)$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ |\ Q\ x) = (P\ |\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ {-}{-}{>}\ Q\ x) = (P\ {-}{-}{>}\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ {-}{-}{>}\ Q) = ((EX\ x{:}A.\ P\ x)\ {-}{-}{>}\ Q)$
!!$P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
!!$P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
!!$a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
!!$A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$
!!$P\ Q.\ (ALL\ x{:}Collect\ Q.\ P\ x) = (ALL\ x.\ Q\ x\ {-}{-}{>}\ P\ x)$
!!$A\ P\ f.\ (ALL\ x{:}f`A.\ P\ x) = (ALL\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(ALL\ x{:}A.\ P\ x)) = (EX\ x{:}A.\ \sim P\ x)$
⟨*proof*⟩

lemma *bex-simps* [*simp*,*no-atp*]:
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ x\ \&\ Q) = ((EX\ x{:}A.\ P\ x)\ \&\ Q)$
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x{:}A.\ Q\ x))$
!!$P.\ (EX\ x{:}\{\}.\ P\ x) = False$
!!$P.\ (EX\ x{:}UNIV.\ P\ x) = (EX\ x.\ P\ x)$
!!$a\ B\ P.\ (EX\ x{:}insert\ a\ B.\ P\ x) = (P(a)\ |\ (EX\ x{:}B.\ P\ x))$
!!$A\ P.\ (EX\ x{:}Union\ A.\ P\ x) = (EX\ y{:}A.\ EX\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (EX\ x{:}\ UNION\ A\ B.\ P\ x) = (EX\ a{:}A.\ EX\ x{:}B\ a.\ P\ x)$

$!!P\ Q.\ (EX\ x{:}Collect\ Q.\ P\ x) = (EX\ x.\ Q\ x\ \&\ P\ x)$
$!!A\ P\ f.\ (EX\ x{:}f`A.\ P\ x) = (EX\ x{:}A.\ P\ (f\ x))$
$!!A\ P.\ (\sim(EX\ x{:}A.\ P\ x)) = (ALL\ x{:}A.\ \sim P\ x)$
⟨*proof*⟩

lemma *ball-conj-distrib*:
$(ALL\ x{:}A.\ P\ x\ \&\ Q\ x) = ((ALL\ x{:}A.\ P\ x)\ \&\ (ALL\ x{:}A.\ Q\ x))$
⟨*proof*⟩

lemma *bex-disj-distrib*:
$(EX\ x{:}A.\ P\ x\ |\ Q\ x) = ((EX\ x{:}A.\ P\ x)\ |\ (EX\ x{:}A.\ Q\ x))$
⟨*proof*⟩

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:
$!!a\ B\ C.\ insert\ a\ (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ \{a\}\ else\ (UN\ x{:}C.\ insert\ a\ (B\ x)))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x)\ Un\ B = (if\ C{=}\{\}\ then\ B\ else\ (UN\ x{:}C.\ A\ x\ Un\ B))$
$!!A\ B\ C.\ A\ Un\ (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (UN\ x{:}C.\ A\ Un\ B\ x))$
$!!A\ B\ C.\ ((UN\ x{:}C.\ A\ x)\ Int\ B) = (UN\ x{:}C.\ A\ x\ Int\ B)$
$!!A\ B\ C.\ (A\ Int\ (UN\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A\ Int\ B\ x)$
$!!A\ B\ C.\ ((UN\ x{:}C.\ A\ x) - B) = (UN\ x{:}C.\ A\ x - B)$
$!!A\ B\ C.\ (A - (INT\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A - B\ x)$
$!!A\ B.\ (UN\ y{:}A.\ UN\ x{:}y.\ B\ x) = (UN\ x{:}\ Union\ A.\ B\ x)$
$!!A\ B\ C.\ (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z) = (UN\ z{:}\ UNION\ A\ B.\ C\ z)$
$!!A\ B\ f.\ (UN\ a{:}A.\ B\ (f\ a)) = (UN\ x{:}f`A.\ B\ x)$
⟨*proof*⟩

lemma *INT-extend-simps*:
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x)\ Int\ B = (if\ C{=}\{\}\ then\ B\ else\ (INT\ x{:}C.\ A\ x\ Int\ B))$
$!!A\ B\ C.\ A\ Int\ (INT\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (INT\ x{:}C.\ A\ Int\ B\ x))$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x) - B = (if\ C{=}\{\}\ then\ UNIV{-}B\ else\ (INT\ x{:}C.\ A\ x - B))$
$!!A\ B\ C.\ A - (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (INT\ x{:}C.\ A - B\ x))$
$!!a\ B\ C.\ insert\ a\ (INT\ x{:}C.\ B\ x) = (INT\ x{:}C.\ insert\ a\ (B\ x))$
$!!A\ B\ C.\ ((INT\ x{:}C.\ A\ x)\ Un\ B) = (INT\ x{:}C.\ A\ x\ Un\ B)$
$!!A\ B\ C.\ A\ Un\ (INT\ x{:}C.\ B\ x) = (INT\ x{:}C.\ A\ Un\ B\ x)$
$!!A\ B.\ (INT\ y{:}A.\ INT\ x{:}y.\ B\ x) = (INT\ x{:}\ Union\ A.\ B\ x)$
$!!A\ B\ C.\ (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z) = (INT\ z{:}\ UNION\ A\ B.\ C\ z)$
$!!A\ B\ f.\ (INT\ a{:}A.\ B\ (f\ a)) = (INT\ x{:}f`A.\ B\ x)$
⟨*proof*⟩

no-notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
Inf ($\bigsqcap$- [*900*] *900*) **and**

Sup (⨆ - [*900*] *900*) **and**
top (⊤) **and**
bot (⊥)

lemmas *mem-simps* =
insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [*simp*] above.

end

9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Complete-Lattice*
uses
(*Tools/inductive.ML*)
Tools/dseq.ML
(*Tools/inductive-codegen.ML*)
Tools/Datatype/datatype-aux.ML
Tools/Datatype/datatype-prop.ML
Tools/Datatype/datatype-case.ML
(*Tools/Datatype/datatype-abs-proofs.ML*)
(*Tools/Datatype/datatype-data.ML*)
(*Tools/old-primrec.ML*)
(*Tools/primrec.ML*)
(*Tools/Datatype/datatype-codegen.ML*)
begin

9.1 Least and greatest fixed points

context *complete-lattice*
begin

definition
lfp :: ('*a* ⇒ '*a*) ⇒ '*a* **where**
lfp f = *Inf* {*u*. *f u* ≤ *u*} — least fixed point

definition
gfp :: ('*a* ⇒ '*a*) ⇒ '*a* **where**
gfp f = *Sup* {*u*. *u* ≤ *f u*} — greatest fixed point

9.2 Proof of Knaster-Tarski Theorem using *lfp*

lfp f is the least upper bound of the set {*u*. *f u* ≤ *u*}

lemma *lfp-lowerbound*: *f A* ≤ *A* ==> *lfp f* ≤ *A*

⟨*proof*⟩

lemma *lfp-greatest*: (!!*u*. *f u* ≤ *u* ==> *A* ≤ *u*) ==> *A* ≤ *lfp f*
⟨*proof*⟩

end

lemma *lfp-lemma2*: *mono f* ==> *f* (*lfp f*) ≤ *lfp f*
⟨*proof*⟩

lemma *lfp-lemma3*: *mono f* ==> *lfp f* ≤ *f* (*lfp f*)
⟨*proof*⟩

lemma *lfp-unfold*: *mono f* ==> *lfp f* = *f* (*lfp f*)
⟨*proof*⟩

lemma *lfp-const*: *lfp* (λ*x*. *t*) = *t*
⟨*proof*⟩

9.3 General induction rules for least fixed points

theorem *lfp-induct*:
assumes *mono*: *mono f* **and** *ind*: *f* (*inf* (*lfp f*) *P*) <= *P*
shows *lfp f* <= *P*
⟨*proof*⟩

lemma *lfp-induct-set*:
assumes *lfp*: *a*: *lfp*(*f*)
and *mono*: *mono*(*f*)
and *indhyp*: !!*x*. [| *x*: *f*(*lfp*(*f*) *Int* {*x*. *P*(*x*)}) |] ==> *P*(*x*)
shows *P*(*a*)
⟨*proof*⟩

lemma *lfp-ordinal-induct*:
fixes *f* :: 'a::*complete-lattice* ⇒ 'a
assumes *mono*: *mono f*
and *P-f*: ⋀*S*. *P S* ⟹ *P* (*f S*)
and *P-Union*: ⋀*M*. ∀ *S*∈*M*. *P S* ⟹ *P* (*Sup M*)
shows *P* (*lfp f*)
⟨*proof*⟩

lemma *lfp-ordinal-induct-set*:
assumes *mono*: *mono f*
and *P-f*: !!*S*. *P S* ==> *P*(*f S*)
and *P-Union*: !!*M*. !*S*:*M*. *P S* ==> *P*(*Union M*)
shows *P*(*lfp f*)
⟨*proof*⟩

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

lemma *def-lfp-unfold*: [| h==*lfp*(f); *mono*(f) |] ==> $h = f(h)$
⟨*proof*⟩

lemma *def-lfp-induct*:
[| A == *lfp*(f); *mono*(f);
f (*inf A P*) ≤ *P*
|] ==> $A \leq P$
⟨*proof*⟩

lemma *def-lfp-induct-set*:
[| A == *lfp*(f); *mono*(f); a:A;
!!x. [| x: f(A *Int* {x. $P(x)$}) |] ==> $P(x)$
|] ==> $P(a)$
⟨*proof*⟩

lemma *lfp-mono*: (!!Z. *f Z* ≤ *g Z*) ==> *lfp f* ≤ *lfp g*
⟨*proof*⟩

9.4 Proof of Knaster-Tarski Theorem using *gfp*

gfp f is the greatest lower bound of the set {u. $u \leq f\ u$}

lemma *gfp-upperbound*: $X \leq f\ X$ ==> $X \leq$ *gfp f*
⟨*proof*⟩

lemma *gfp-least*: (!!u. $u \leq f\ u$ ==> $u \leq X$) ==> *gfp f* ≤ 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*⟩

9.5 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: [| a: X; $X \subseteq f(X)$ |] ==> a : *gfp*(f)
⟨*proof*⟩

lemma *weak-coinduct-image*: !!X. [| a : X; g‘$X \subseteq f$ (g‘X) |] ==> $g\ a$: *gfp f*
⟨*proof*⟩

lemma *coinduct-lemma*:
[| *X* ≤ *f* (*sup X* (*gfp f*)); *mono f* |] ==> *sup X* (*gfp f*) ≤ *f* (*sup X* (*gfp f*))

⟨*proof*⟩

strong version, thanks to Coen and Frost

lemma *coinduct-set*: [| *mono*(*f*); *a*: *X*; *X* ⊆ *f*(*X Un gfp*(*f*)) |] ==> *a* : *gfp*(*f*)
⟨*proof*⟩

lemma *coinduct*: [| *mono*(*f*); *X* ≤ *f* (*sup X* (*gfp f*)) |] ==> *X* ≤ *gfp*(*f*)
⟨*proof*⟩

lemma *gfp-fun-UnI2*: [| *mono*(*f*); *a*: *gfp*(*f*) |] ==> *a*: *f*(*X Un gfp*(*f*))
⟨*proof*⟩

9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition *X* ⊆ *f X* to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: *mono*(*f*) ==> *mono*(%*x*. *f*(*x*) *Un X Un B*)
⟨*proof*⟩

lemma *coinduct3-lemma*:
[| *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*))); *mono*(*f*) |]
==> *lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*)) ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*)))
⟨*proof*⟩

lemma *coinduct3*:
[| *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*))) |] ==> *a* : *gfp*(*f*)
⟨*proof*⟩

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

lemma *def-gfp-unfold*: [| *A*==*gfp*(*f*); *mono*(*f*) |] ==> *A* = *f*(*A*)
⟨*proof*⟩

lemma *def-coinduct*:
[| *A*==*gfp*(*f*); *mono*(*f*); *X* ≤ *f*(*sup X A*) |] ==> *X* ≤ *A*
⟨*proof*⟩

lemma *def-coinduct-set*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*X Un A*) |] ==> *a*: *A*
⟨*proof*⟩

lemma *def-Collect-coinduct*:
[| *A* == *gfp*(%*w*. *Collect*(*P*(*w*))); *mono*(%*w*. *Collect*(*P*(*w*)));
a: *X*; !!*z*. *z*: *X* ==> *P* (*X Un A*) *z* |] ==>
a : *A*
⟨*proof*⟩

lemma *def-coinduct3*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un A*)) |] ==> *a*: *A*

⟨*proof*⟩

Monotonicity of *gfp*!

lemma *gfp-mono*: $(!!Z.\ f\ Z \leq g\ Z) ==> gfp\ f \leq gfp\ g$
⟨*proof*⟩

9.7 Inductive predicates and sets

Package setup.

theorems *basic-monos* =
subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
Collect-mono in-mono vimage-mono

⟨*ML*⟩

theorems [*mono*] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-mono not-mono
Ball-def Bex-def
induct-rulify-fallback

9.8 Inductive datatypes and primitive recursion

Package setup.

⟨*ML*⟩

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((%-) *10*)
syntax (*xsymbols*)
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((λ-) *10*)

⟨*ML*⟩

end

10 Fun: Notions about functions

theory *Fun*
imports *Complete-Lattice*
begin

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma *expand-fun-eq*: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$
⟨*proof*⟩

lemma *apply-inverse*:
$f\ x = u \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow g\ (f\ x) = x) \Longrightarrow P\ x \Longrightarrow x = g\ u$
⟨*proof*⟩

10.1 The Identity Function *id*

definition
$id :: 'a \Rightarrow 'a$
where
$id = (\lambda x.\ x)$

lemma *id-apply* [*simp*]: $id\ x = x$
⟨*proof*⟩

lemma *image-ident* [*simp*]: $(\%x.\ x)\ `\ Y = Y$
⟨*proof*⟩

lemma *image-id* [*simp*]: $id\ `\ Y = Y$
⟨*proof*⟩

lemma *vimage-ident* [*simp*]: $(\%x.\ x) -` Y = Y$
⟨*proof*⟩

lemma *vimage-id* [*simp*]: $id -` A = A$
⟨*proof*⟩

10.2 The Composition Operator $f \circ g$

definition
$comp :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** *o* *55*)
where
$f\ o\ g = (\lambda x.\ f\ (g\ x))$

notation (*xsymbols*)
comp (**infixl** $\circ$ *55*)

notation (*HTML* **output**)
comp (**infixl** $\circ$ *55*)

compatibility

lemmas *o-def* = *comp-def*

lemma *o-apply* [*simp*]: $(f\ o\ g)\ x = f\ (g\ x)$
⟨*proof*⟩

lemma *o-assoc*: $f\ o\ (g\ o\ h) = f\ o\ g\ o\ h$
⟨*proof*⟩

lemma *id-o* [*simp*]: $id\ o\ g = g$
⟨*proof*⟩

lemma *o-id* [*simp*]: $f\ o\ id = f$
⟨*proof*⟩

lemma *o-eq-dest*:
$a\ o\ b = c\ o\ d \Longrightarrow a\ (b\ v) = c\ (d\ v)$
⟨*proof*⟩

lemma *o-eq-elim*:
$a\ o\ b = c\ o\ d \Longrightarrow ((\bigwedge v.\ a\ (b\ v) = c\ (d\ v)) \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *image-compose*: $(f\ o\ g)\ `\ r = f`(g`r)$
⟨*proof*⟩

lemma *vimage-compose*: $(g \circ f)\ -`\ x = f\ -`\ (g\ -`\ x)$
⟨*proof*⟩

lemma *UN-o*: $UNION\ A\ (g\ o\ f) = UNION\ (f`A)\ g$
⟨*proof*⟩

10.3 The Forward Composition Operator *fcomp*

definition
$fcomp :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$ (**infixl** *o>* *60*)
where
$f\ o>\ g = (\lambda x.\ g\ (f\ x))$

lemma *fcomp-apply*: $(f\ o>\ g)\ x = g\ (f\ x)$
⟨*proof*⟩

lemma *fcomp-assoc*: $(f\ o>\ g)\ o>\ h = f\ o>\ (g\ o>\ h)$
⟨*proof*⟩

lemma *id-fcomp* [*simp*]: $id\ o>\ g = g$
⟨*proof*⟩

lemma *fcomp-id* [*simp*]: $f\ o>\ id = f$
⟨*proof*⟩

code-const *fcomp*
(*Eval* **infixl** *1 #>*)

no-notation *fcomp* (**infixl** *o>* *60*)

10.4 Injectivity and Surjectivity

definition

inj-on :: [*'a* => *'b*, *'a set*] => *bool* **where**
— injective
inj-on f A == ! *x*:*A*. ! *y*:*A*. *f*(*x*)=*f*(*y*) −−> *x*=*y*

A common special case: functions injective over the entire domain type.

abbreviation
inj f == *inj-on f UNIV*

definition
bij-betw :: (*'a* => *'b*) => *'a set* => *'b set* => *bool* **where** — bijective
[*code del*]: *bij-betw f A B* ⟷ *inj-on f A* & *f* ‘ *A* = *B*

definition
surj :: (*'a* => *'b*) => *bool* **where**
— surjective
surj f == ! *y*. ? *x*. *y*=*f*(*x*)

definition
bij :: (*'a* => *'b*) => *bool* **where**
— bijective
bij f == *inj f* & *surj f*

lemma *injI*:
assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *inj f*
⟨*proof*⟩

For Proofs in *Tools/Datatype/datatype-rep-proofs*

lemma *datatype-injI*:
(!! *x*. *ALL y*. *f*(*x*) = *f*(*y*) −−> *x*=*y*) ==> *inj*(*f*)
⟨*proof*⟩

theorem *range-ex1-eq*: *inj f* ⟹ *b* : *range f* = (*EX*! *x*. *b* = *f x*)
⟨*proof*⟩

lemma *injD*: [| *inj*(*f*); *f*(*x*) = *f*(*y*) |] ==> *x*=*y*
⟨*proof*⟩

lemma *inj-on-eq-iff*: *inj-on f A* ==> *x*:*A* ==> *y*:*A* ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
⟨*proof*⟩

lemma *inj-eq*: *inj f* ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
⟨*proof*⟩

lemma *inj-on-id*[*simp*]: *inj-on id A*
⟨*proof*⟩

lemma *inj-on-id2*[*simp*]: *inj-on* (%*x*. *x*) *A*
⟨*proof*⟩

lemma *surj-id*[*simp*]: *surj id*
⟨*proof*⟩

lemma *bij-id*[*simp*]: *bij id*
⟨*proof*⟩

lemma *inj-onI*:
(!! *x y*. [| *x*:*A*; *y*:*A*; *f*(*x*) = *f*(*y*) |] ==> *x*=*y*) ==> *inj-on f A*
⟨*proof*⟩

lemma *inj-on-inverseI*: (!!*x*. *x*:*A* ==> *g*(*f*(*x*)) = *x*) ==> *inj-on f A*
⟨*proof*⟩

lemma *inj-onD*: [| *inj-on f A*; *f*(*x*)=*f*(*y*); *x*:*A*; *y*:*A* |] ==> *x*=*y*
⟨*proof*⟩

lemma *inj-on-iff*: [| *inj-on f A*; *x*:*A*; *y*:*A* |] ==> (*f*(*x*)=*f*(*y*)) = (*x*=*y*)
⟨*proof*⟩

lemma *comp-inj-on*:
[| *inj-on f A*; *inj-on g* (*f*‘*A*) |] ==> *inj-on* (*g o f*) *A*
⟨*proof*⟩

lemma *inj-on-imageI*: *inj-on* (*g o f*) *A* ⟹ *inj-on g* (*f* ‘ *A*)
⟨*proof*⟩

lemma *inj-on-image-iff*: ⟦ *ALL x*:*A*. *ALL y*:*A*. (*g*(*f x*) = *g*(*f y*)) = (*g x* = *g y*);
inj-on f A ⟧ ⟹ *inj-on g* (*f* ‘ *A*) = *inj-on g A*
⟨*proof*⟩

lemma *inj-on-contraD*: [| *inj-on f A*; ~*x*=*y*; *x*:*A*; *y*:*A* |] ==> ~ *f*(*x*)=*f*(*y*)
⟨*proof*⟩

lemma *inj-singleton*: *inj* (%*s*. {*s*})
⟨*proof*⟩

lemma *inj-on-empty*[*iff*]: *inj-on f* {}
⟨*proof*⟩

lemma *subset-inj-on*: [| *inj-on f B*; *A* <= *B* |] ==> *inj-on f A*
⟨*proof*⟩

lemma *inj-on-Un*:
inj-on f (*A Un B*) =
(*inj-on f A* & *inj-on f B* & *f*‘(*A*−*B*) *Int f*‘(*B*−*A*) = {})
⟨*proof*⟩

lemma *inj-on-insert*[*iff*]:

inj-on f (insert a A) = (inj-on f A & f a ~: f`(A-{a}))
⟨*proof*⟩

lemma *inj-on-diff*: *inj-on f A ==> inj-on f (A-B)*
⟨*proof*⟩

lemma *surjI*: *(!! x. g(f x) = x) ==> surj g*
⟨*proof*⟩

lemma *surj-range*: *surj f ==> range f = UNIV*
⟨*proof*⟩

lemma *surjD*: *surj f ==> EX x. y = f x*
⟨*proof*⟩

lemma *surjE*: *surj f ==> (!!x. y = f x ==> C) ==> C*
⟨*proof*⟩

lemma *comp-surj*: *[| surj f; surj g |] ==> surj (g o f)*
⟨*proof*⟩

lemma *bijI*: *[| inj f; surj f |] ==> bij f*
⟨*proof*⟩

lemma *bij-is-inj*: *bij f ==> inj f*
⟨*proof*⟩

lemma *bij-is-surj*: *bij f ==> surj f*
⟨*proof*⟩

lemma *bij-betw-imp-inj-on*: *bij-betw f A B ⟹ inj-on f A*
⟨*proof*⟩

lemma *bij-comp*: *bij f ⟹ bij g ⟹ bij (g o f)*
⟨*proof*⟩

lemma *bij-betw-trans*:
bij-betw f A B ⟹ bij-betw g B C ⟹ bij-betw (g o f) A C
⟨*proof*⟩

lemma *bij-betw-inv*: **assumes** *bij-betw f A B* **shows** *EX g. bij-betw g B A*
⟨*proof*⟩

lemma *surj-image-vimage-eq*: *surj f ==> f ` (f -` A) = A*
⟨*proof*⟩

lemma *inj-vimage-image-eq*: *inj f ==> f -` (f ` A) = A*
⟨*proof*⟩

lemma *vimage-subsetD*: *surj f ==> f −‘ B <= A ==> B <= f ‘ A*
⟨*proof*⟩

lemma *vimage-subsetI*: *inj f ==> B <= f ‘ A ==> f −‘ B <= A*
⟨*proof*⟩

lemma *vimage-subset-eq*: *bij f ==> (f −‘ B <= A) = (B <= f ‘ A)*
⟨*proof*⟩

lemma *inj-on-Un-image-eq-iff*: *inj-on f (A ∪ B) ⟹ f ‘ A = f ‘ B ⟷ A = B*
⟨*proof*⟩

lemma *inj-on-image-Int*:
[| *inj-on f C*; *A<=C*; *B<=C* |] *==> f‘(A Int B) = f‘A Int f‘B*
⟨*proof*⟩

lemma *inj-on-image-set-diff*:
[| *inj-on f C*; *A<=C*; *B<=C* |] *==> f‘(A−B) = f‘A − f‘B*
⟨*proof*⟩

lemma *image-Int*: *inj f ==> f‘(A Int B) = f‘A Int f‘B*
⟨*proof*⟩

lemma *image-set-diff*: *inj f ==> f‘(A−B) = f‘A − f‘B*
⟨*proof*⟩

lemma *inj-image-mem-iff*: *inj f ==> (f a : f‘A) = (a : A)*
⟨*proof*⟩

lemma *inj-image-subset-iff*: *inj f ==> (f‘A <= f‘B) = (A<=B)*
⟨*proof*⟩

lemma *inj-image-eq-iff*: *inj f ==> (f‘A = f‘B) = (A = B)*
⟨*proof*⟩

lemma *image-INT*:
[| *inj-on f C*; *ALL x:A. B x <= C*; *j:A* |]
==> f ‘ (INTER A B) = (INT x:A. f ‘ B x)
⟨*proof*⟩

lemma *bij-image-INT*: *bij f ==> f ‘ (INTER A B) = (INT x:A. f ‘ B x)*
⟨*proof*⟩

lemma *surj-Compl-image-subset*: *surj f ==> −(f‘A) <= f‘(−A)*
⟨*proof*⟩

lemma *inj-image-Compl-subset*: *inj f ==> f‘(−A) <= −(f‘A)*

⟨*proof*⟩

lemma *bij-image-Compl-eq*: *bij f* ==> *f*‘(−*A*) = −(*f*‘*A*)
⟨*proof*⟩

lemma (**in** *ordered-ab-group-add*) *inj-uminus*[*simp*, *intro*]: *inj-on uminus A*
⟨*proof*⟩

lemma (**in** *linorder*) *strict-mono-imp-inj-on*: *strict-mono f* ⟹ *inj-on f A*
⟨*proof*⟩

10.5 Function Updating

definition
fun-upd :: (′*a* => ′*b*) => ′*a* => ′*b* => (′*a* => ′*b*) **where**
fun-upd f a b == % *x*. *if x*=*a then b else f x*

nonterminals
updbinds updbind
syntax
-*updbind* :: [′*a*, ′*a*] => *updbind* ((2- :=/ -))
:: *updbind* => *updbinds* (-)
-*updbinds*:: [*updbind*, *updbinds*] => *updbinds* (-,/ -)
-*Update* :: [′*a*, *updbinds*] => ′*a* (-/′((-)′) [1000, 0] 900)

translations
-*Update f* (-*updbinds b bs*) == -*Update* (-*Update f b*) *bs*
f(*x*:=*y*) == *CONST fun-upd f x y*

lemma *fun-upd-idem-iff*: (*f*(*x*:=*y*) = *f*) = (*f x* = *y*)
⟨*proof*⟩

lemmas *fun-upd-idem* = *fun-upd-idem-iff* [*THEN iffD2*, *standard*]

lemmas *fun-upd-triv* = *refl* [*THEN fun-upd-idem*]
declare *fun-upd-triv* [*iff*]

lemma *fun-upd-apply* [*simp*]: (*f*(*x*:=*y*))*z* = (*if z*=*x then y else f z*)
⟨*proof*⟩

lemma *fun-upd-same*: (*f*(*x*:=*y*)) *x* = *y*
⟨*proof*⟩

lemma *fun-upd-other*: *z*~=*x* ==> (*f*(*x*:=*y*)) *z* = *f z*

⟨*proof*⟩

lemma *fun-upd-upd* [*simp*]: $f(x{:=}y,x{:=}z) = f(x{:=}z)$
⟨*proof*⟩

lemma *fun-upd-twist*: $a \neq c \Longrightarrow (m(a{:=}b))(c{:=}d) = (m(c{:=}d))(a{:=}b)$
⟨*proof*⟩

lemma *inj-on-fun-updI*: $\llbracket$ *inj-on f A*; $y \notin f`A$ $\rrbracket \Longrightarrow$ *inj-on* $(f(x{:=}y))$ A
⟨*proof*⟩

lemma *fun-upd-image*:
$f(x{:=}y)$ ` $A$ = (*if* $x \in A$ *then insert* $y$ ($f$ ` ($A - \{x\}$)) *else* f ` A)
⟨*proof*⟩

lemma *fun-upd-comp*: $f \circ (g(x := y)) = (f \circ g)(x := f\ y)$
⟨*proof*⟩

10.6 *override-on*

definition
override-on :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$
where
override-on f g A = $(\lambda a.$ *if* $a \in A$ *then g a else f a*$)$

lemma *override-on-emptyset*[*simp*]: *override-on f g* $\{\}$ = *f*
⟨*proof*⟩

lemma *override-on-apply-notin*[*simp*]: $a \notin A \Longrightarrow$ (*override-on f g A*) *a* = *f a*
⟨*proof*⟩

lemma *override-on-apply-in*[*simp*]: $a \in A \Longrightarrow$ (*override-on f g A*) *a* = *g a*
⟨*proof*⟩

10.7 *swap*

definition
swap :: $'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$
where
swap a b f = *f* (*a* := *f b*, *b*:= *f a*)

lemma *swap-self* [*simp*]: *swap a a f* = *f*
⟨*proof*⟩

lemma *swap-commute*: *swap a b f* = *swap b a f*
⟨*proof*⟩

lemma *swap-nilpotent* [*simp*]: *swap a b* (*swap a b f*) = *f*
⟨*proof*⟩

lemma *swap-triple*:
assumes $a \neq c$ **and** $b \neq c$
shows $swap\ a\ b\ (swap\ b\ c\ (swap\ a\ b\ f)) = swap\ a\ c\ f$
⟨*proof*⟩

lemma *comp-swap*: $f \circ swap\ a\ b\ g = swap\ a\ b\ (f \circ g)$
⟨*proof*⟩

lemma *inj-on-imp-inj-on-swap*:
$[|inj\text{-}on\ f\ A;\ a \in A;\ b \in A|] ==> inj\text{-}on\ (swap\ a\ b\ f)\ A$
⟨*proof*⟩

lemma *inj-on-swap-iff* [*simp*]:
assumes *A*: $a \in A\ b \in A$ **shows** $inj\text{-}on\ (swap\ a\ b\ f)\ A = inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *surj-imp-surj-swap*: $surj\ f ==> surj\ (swap\ a\ b\ f)$
⟨*proof*⟩

lemma *surj-swap-iff* [*simp*]: $surj\ (swap\ a\ b\ f) = surj\ f$
⟨*proof*⟩

lemma *bij-swap-iff*: $bij\ (swap\ a\ b\ f) = bij\ f$
⟨*proof*⟩

hide-const (**open**) *swap*

10.8 Inversion of injective functions

definition *the-inv-into* :: $'a\ set => ('a => 'b) => ('b => 'a)$ **where**
$the\text{-}inv\text{-}into\ A\ f == \%x.\ THE\ y.\ y : A\ \&\ f\ y = x$

lemma *the-inv-into-f-f*:
$[|\ inj\text{-}on\ f\ A;\ \ x : A\ |] ==> the\text{-}inv\text{-}into\ A\ f\ (f\ x) = x$
⟨*proof*⟩

lemma *f-the-inv-into-f*:
$inj\text{-}on\ f\ A ==> y : f`A\ \ ==> f\ (the\text{-}inv\text{-}into\ A\ f\ y) = y$
⟨*proof*⟩

lemma *the-inv-into-into*:
$[|\ inj\text{-}on\ f\ A;\ x : f\ `\ A;\ A <= B\ |] ==> the\text{-}inv\text{-}into\ A\ f\ x : B$
⟨*proof*⟩

lemma *the-inv-into-onto*[*simp*]:
$inj\text{-}on\ f\ A ==> the\text{-}inv\text{-}into\ A\ f\ `\ (f\ `\ A) = A$
⟨*proof*⟩

lemma *the-inv-into-f-eq*:

[| *inj-on f A*; *f x* = *y*; *x* : *A* |] ==> *the-inv-into A f y* = *x*
⟨*proof*⟩

lemma *the-inv-into-comp*:
[| *inj-on f* (*g* ' *A*); *inj-on g A*; *x* : *f* ' *g* ' *A* |] ==>
the-inv-into A (*f o g*) *x* = (*the-inv-into A g o the-inv-into* (*g* ' *A*) *f*) *x*
⟨*proof*⟩

lemma *inj-on-the-inv-into*:
inj-on f A ⟹ *inj-on* (*the-inv-into A f*) (*f* ' *A*)
⟨*proof*⟩

lemma *bij-betw-the-inv-into*:
bij-betw f A B ⟹ *bij-betw* (*the-inv-into A f*) *B A*
⟨*proof*⟩

abbreviation *the-inv* :: ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a*) **where**
the-inv f ≡ *the-inv-into UNIV f*

lemma *the-inv-f-f*:
assumes *inj f*
shows *the-inv f* (*f x*) = *x* ⟨*proof*⟩

10.9 Proof tool setup

simplifies terms of the form f(...,x:=y,...,x:=z,...) to f(...,x:=z,...)

⟨*ML*⟩

10.10 Code generator setup

types-code
fun ((- –>/ -))
attach (*term-of*) ⟪
fun term-of-fun-type - aT - bT - = Free (<*function*>, *aT* –-> *bT*);
⟫
attach (*test*) ⟪
fun gen-fun-type aF aT bG bT i =
let
val tab = *Unsynchronized.ref* [];
fun mk-upd (*x*, (-, *y*)) *t* = *Const* (*Fun.fun-upd*,
(*aT* –-> *bT*) –-> *aT* –-> *bT* –-> *aT* –-> *bT*) $ *t* $ *aF x* $ *y* ()
in
(*fn x* =>
case AList.lookup op = (!*tab*) *x of*
NONE =>
let val p as (*y*, -) = *bG i*
in (*tab* := (*x*, *p*) :: !*tab*; *y*) *end*
| *SOME* (*y*, -) => *y*,
fn () => *Basics.fold mk-upd* (!*tab*) (*Const* (*HOL.undefined*, *aT* –-> *bT*)))

end;
⟫

code-const *op* ∘
(*SML* **infixl** *5 o*)
(*Haskell* **infixr** *9* .)

code-const *id*
(*Haskell id*)

end

11 Product-Type: Cartesian products

theory *Product-Type*
imports *Typedef Inductive Fun*
uses
(*Tools/split-rule.ML*)
(*Tools/inductive-set.ML*)
begin

11.1 *bool* is a datatype

rep-datatype *True False* ⟨*proof*⟩

declare *case-split* [*cases type*: *bool*]
— prefer plain propositional version

lemma
shows [*code*]: *eq-class.eq False P* ⟷ ¬ *P*
and [*code*]: *eq-class.eq True P* ⟷ *P*
and [*code*]: *eq-class.eq P False* ⟷ ¬ *P*
and [*code*]: *eq-class.eq P True* ⟷ *P*
and [*code nbe*]: *eq-class.eq P P* ⟷ *True*
⟨*proof*⟩

code-const *eq-class.eq* :: *bool* ⇒ *bool* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-instance *bool* :: *eq*
(*Haskell* −)

11.2 The *unit* type

typedef *unit* = {*True*}
⟨*proof*⟩

definition

Unity :: *unit* (′()′)
where
() = *Abs-unit True*

lemma *unit-eq* [*no-atp*]: *u* = ()
⟨*proof*⟩

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

⟨*ML*⟩

rep-datatype () ⟨*proof*⟩

lemma *unit-all-eq1*: (!!*x*::*unit*. *PROP P x*) == *PROP P* ()
⟨*proof*⟩

lemma *unit-all-eq2*: (!!*x*::*unit*. *PROP P*) == *PROP P*
⟨*proof*⟩

This rewrite counters the effect of *unit-eq-proc* on %*u*::*unit*. *f u*, replacing it by *f* rather than by %*u*. *f* ().

lemma *unit-abs-eta-conv* [*simp*,*no-atp*]: (%*u*::*unit*. *f* ()) = *f*
⟨*proof*⟩

instantiation *unit* :: *default*
begin

definition *default* = ()

instance ⟨*proof*⟩

end

lemma [*code*]:
eq-class.eq (*u*::*unit*) *v* ⟷ *True* ⟨*proof*⟩

code-type *unit*
(*SML unit*)
(*OCaml unit*)
(*Haskell* ())
(*Scala Unit*)

code-const *Unity*
(*SML* ())
(*OCaml* ())
(*Haskell* ())
(*Scala* ())

code-instance *unit* :: *eq*

(*Haskell* −)

code-const *eq-class.eq* :: *unit* ⇒ *unit* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-reserved *SML*
unit

code-reserved *OCaml*
unit

code-reserved *Scala*
Unit

11.3 The product type

11.3.1 Type definition

definition *Pair-Rep* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ **where**
$Pair\text{-}Rep\ a\ b = (\lambda x\ y.\ x = a \wedge y = b)$

global

typedef (*Prod*)
$('a, 'b) *$ (**infixr** $*$ *20*)
$= \{f.\ \exists\, a\ b.\ f = Pair\text{-}Rep\ (a{::}'a)\ (b{::}'b)\}$
⟨*proof*⟩

type-notation (*xsymbols*)
$*$ ((- ×/ -) [*21*, *20*] *20*)
type-notation (*HTML* **output**)
$*$ ((- ×/ -) [*21*, *20*] *20*)

consts
Pair :: $'a \Rightarrow 'b \Rightarrow 'a \times 'b$

local

defs
Pair-def: $Pair\ a\ b == Abs\text{-}Prod\ (Pair\text{-}Rep\ a\ b)$

rep-datatype (*prod*) *Pair* ⟨*proof*⟩

11.3.2 Tuple syntax

global consts
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$

local defs
split-prod-case: $split == prod\text{-}case$

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminals
tuple-args patterns

syntax
-tuple :: *'a => tuple-args => 'a * 'b* ((*1 '(-,/ -')*))
-tuple-arg :: *'a => tuple-args* (-)
-tuple-args :: *'a => tuple-args => tuple-args* (-,/ -)
-pattern :: [*pttrn*, *patterns*] *=> pttrn* (*'(-,/ -')*)
:: *pttrn => patterns* (-)
-patterns :: [*pttrn*, *patterns*] *=> patterns* (-,/ -)

translations
(*x*, *y*) == *CONST Pair x y*
-tuple x (*-tuple-args y z*) == *-tuple x* (*-tuple-arg* (*-tuple y z*))
%(*x*, *y*, *zs*). *b* == *CONST split* (%*x* (*y*, *zs*). *b*)
%(*x*, *y*). *b* == *CONST split* (%*x y*. *b*)
-abs (*CONST Pair x y*) *t* => %(*x*, *y*). *t*
— The last rule accommodates tuples in 'case C ... (x,y) ... =¿ ...' The (x,y) is parsed as 'Pair x y' because it is logic, not pttrn

⟨*ML*⟩

11.3.3 Code generator setup

lemma *split-case-cert*:
assumes *CASE* ≡ *split f*
shows *CASE* (*a*, *b*) ≡ *f a b*
⟨*proof*⟩

⟨*ML*⟩

code-type *
(*SML* **infix** *2* *)
(*OCaml* **infix** *2* *)
(*Haskell* !((-),/ (-)))
(*Scala* ((-),/ (-)))

code-const *Pair*
(*SML* !((-),/ (-)))
(*OCaml* !((-),/ (-)))
(*Haskell* !((-),/ (-)))
(*Scala* !((-),/ (-)))

code-instance * :: *eq*
(*Haskell* −)

code-const *eq-class.eq* :: *'a*::*eq* × *'b*::*eq* ⇒ *'a* × *'b* ⇒ *bool*

(*Haskell* **infixl** *4* ==)

types-code
  ∗ ((- ∗/ -))
attach (*term-of*) ⟪
fun term-of-id-42 aF aT bF bT (*x*, *y*) = *HOLogic.pair-const aT bT* $ *aF x* $ *bF y*;
⟫
attach (*test*) ⟪
fun gen-id-42 aG aT bG bT i =
  let
    val (*x*, *t*) = *aG i*;
    val (*y*, *u*) = *bG i*
  in ((*x*, *y*), *fn* () => *HOLogic.pair-const aT bT* $ *t* () $ *u* ()) *end*;
⟫

consts-code
  Pair ((-,/ -))

⟨*ML*⟩

11.3.4 Fundamental operations and properties

lemma *surj-pair* [*simp*]: *EX x y*. *p* = (*x*, *y*)
  ⟨*proof*⟩

global consts
  fst :: ′*a* × ′*b* ⇒ ′*a*
  snd :: ′*a* × ′*b* ⇒ ′*b*

local defs
  fst-def: *fst p* == *case p of* (*a*, *b*) ⇒ *a*
  snd-def: *snd p* == *case p of* (*a*, *b*) ⇒ *b*

lemma *fst-conv* [*simp*, *code*]: *fst* (*a*, *b*) = *a*
  ⟨*proof*⟩

lemma *snd-conv* [*simp*, *code*]: *snd* (*a*, *b*) = *b*
  ⟨*proof*⟩

code-const *fst* **and** *snd*
  (*Haskell fst* **and** *snd*)

lemma *prod-case-unfold*: *prod-case* = (%*c p*. *c* (*fst p*) (*snd p*))
  ⟨*proof*⟩

lemma *fst-eqD*: *fst* (*x*, *y*) = *a* ==> *x* = *a*
  ⟨*proof*⟩

lemma *snd-eqD*: $snd\ (x,\ y) = a \Longrightarrow y = a$
⟨*proof*⟩

lemma *pair-collapse* [*simp*]: $(fst\ p,\ snd\ p) = p$
⟨*proof*⟩

lemmas *surjective-pairing* = *pair-collapse* [*symmetric*]

lemma *Pair-fst-snd-eq*: $s = t \longleftrightarrow fst\ s = fst\ t \wedge snd\ s = snd\ t$
⟨*proof*⟩

lemma *prod-eqI* [*intro?*]: $fst\ p = fst\ q \Longrightarrow snd\ p = snd\ q \Longrightarrow p = q$
⟨*proof*⟩

lemma *split-conv* [*simp*, *code*]: $split\ f\ (a,\ b) = f\ a\ b$
⟨*proof*⟩

lemma *splitI*: $f\ a\ b \Longrightarrow split\ f\ (a,\ b)$
⟨*proof*⟩

lemma *splitD*: $split\ f\ (a,\ b) \Longrightarrow f\ a\ b$
⟨*proof*⟩

lemma *split-Pair* [*simp*]: $(\lambda(x,\ y).\ (x,\ y)) = id$
⟨*proof*⟩

lemma *split-eta*: $(\lambda(x,\ y).\ f\ (x,\ y)) = f$
— Subsumes the old *split-Pair* when f is the identity function.
⟨*proof*⟩

lemma *split-comp*: $split\ (f \circ g)\ x = f\ (g\ (fst\ x))\ (snd\ x)$
⟨*proof*⟩

lemma *split-twice*: $split\ f\ (split\ g\ p) = split\ (\lambda x\ y.\ split\ f\ (g\ x\ y))\ p$
⟨*proof*⟩

lemma *The-split*: $The\ (split\ P) = (THE\ xy.\ P\ (fst\ xy)\ (snd\ xy))$
⟨*proof*⟩

lemma *split-weak-cong*: $p = q \Longrightarrow split\ c\ p = split\ c\ q$
— Prevents simplification of c: much faster
⟨*proof*⟩

lemma *cond-split-eta*: $(!!x\ y.\ f\ x\ y = g\ (x,\ y)) \Longrightarrow (\%(x,\ y).\ f\ x\ y) = g$
⟨*proof*⟩

lemma *split-paired-all*: $(!!x.\ PROP\ P\ x) \equiv (!!a\ b.\ PROP\ P\ (a,\ b))$
⟨*proof*⟩

The rule *split-paired-all* does not work with the Simplifier because it also

affects premises in congrence rules, where this can lead to premises of the form !!*a b*. ... = *?P*(*a*, *b*) which cannot be solved by reflexivity.

lemmas *split-tupled-all* = *split-paired-all unit-all-eq2*

⟨*ML*⟩

lemma *split-paired-All* [*simp*]: (*ALL x*. *P x*) = (*ALL a b*. *P* (*a*, *b*))
— [*iff*] is not a good idea because it makes *blast* loop
⟨*proof*⟩

lemma *split-paired-Ex* [*simp*]: (*EX x*. *P x*) = (*EX a b*. *P* (*a*, *b*))
⟨*proof*⟩

lemma *split-paired-The*: (*THE x*. *P x*) = (*THE* (*a*, *b*). *P* (*a*, *b*))
— Can't be added to simpset: loops!
⟨*proof*⟩

Simplification procedure for *cond-split-eta*. Using *split-eta* as a rewrite rule is not general enough, and using *cond-split-eta* directly would render some existing proofs very inefficient; similarly for *split-beta*.

⟨*ML*⟩

lemma *split-beta* [*mono*]: (%(*x*, *y*). *P x y*) *z* = *P* (*fst z*) (*snd z*)
⟨*proof*⟩

lemma *split-split* [*no-atp*]: *R*(*split c p*) = (*ALL x y*. *p* = (*x*, *y*) −−> *R*(*c x y*))
— For use with *split* and the Simplifier.
⟨*proof*⟩

split-split could be declared as [*split*] done after the Splitter has been speeded up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

lemma *split-split-asm* [*no-atp*]: *R* (*split c p*) = (~(*EX x y*. *p* = (*x*, *y*) & (~*R* (*c x y*))))
⟨*proof*⟩

split used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

lemma *splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *split c p*
⟨*proof*⟩

lemma *splitI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *split c p x*
⟨*proof*⟩

lemma *splitE*: *split c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*

⟨*proof*⟩

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
⟨*proof*⟩

lemma *splitE2*:
[| *Q* (*split P z*); !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R* |] ==> *R*
⟨*proof*⟩

lemma *splitD'*: *split R* (*a*,*b*) *c* ==> *R a b c*
⟨*proof*⟩

lemma *mem-splitI*: *z*: *c a b* ==> *z*: *split c* (*a*, *b*)
⟨*proof*⟩

lemma *mem-splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *z*: *c a b* |] ==> *z*: *split c p*
⟨*proof*⟩

lemma *mem-splitE*:
assumes *major*: *z* ∈ *split c p*
and *cases*: ⋀*x y*. *p* = (*x*, *y*) ⟹ *z* ∈ *c x y* ⟹ *Q*
shows *Q*
⟨*proof*⟩

declare *mem-splitI2* [*intro*!] *mem-splitI* [*intro*!] *splitI2'* [*intro*!] *splitI2* [*intro*!] *splitI* [*intro*!]
declare *mem-splitE* [*elim*!] *splitE'* [*elim*!] *splitE* [*elim*!]

⟨*ML*⟩

lemma *split-eta-SetCompr* [*simp*,*no-atp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
⟨*proof*⟩

lemma *split-eta-SetCompr2* [*simp*,*no-atp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P x y*) = *split P*
⟨*proof*⟩

lemma *split-part* [*simp*]: (%(*a*,*b*). *P* & *Q a b*) = (%*ab*. *P* & *split Q ab*)
— Allows simplifications of nested splits in case of independent predicates.
⟨*proof*⟩

lemma *split-comp-eq*:
fixes *f* :: *'a* => *'b* => *'c* **and** *g* :: *'d* => *'a*
shows (%*u*. *f* (*g* (*fst u*)) (*snd u*)) = (*split* (%*x*. *f* (*g x*)))
⟨*proof*⟩

lemma *pair-imageI* [*intro*]: (*a*, *b*) : *A* ==> *f a b* : (%(*a*, *b*). *f a b*) ' *A*

⟨*proof*⟩

lemma *The-split-eq* [*simp*]: (*THE* (*x'*,*y'*). *x* = *x'* & *y* = *y'*) = (*x*, *y*)
⟨*proof*⟩

Setup of internal *split-rule*.

lemmas *prod-caseI* = *prod.cases* [*THEN iffD2*, *standard*]

lemma *prod-caseI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *prod-case c p*
⟨*proof*⟩

lemma *prod-caseI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *prod-case c p x*
⟨*proof*⟩

lemma *prod-caseE*: *prod-case c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*)
==> *Q*
⟨*proof*⟩

lemma *prod-caseE'*: *prod-case c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==>
Q) ==> *Q*
⟨*proof*⟩

declare *prod-caseI2'* [*intro*!] *prod-caseI2* [*intro*!] *prod-caseI* [*intro*!]
declare *prod-caseE'* [*elim*!] *prod-caseE* [*elim*!]

lemma *prod-case-split*:
prod-case = *split*
⟨*proof*⟩

lemma *prod-case-beta*:
prod-case f p = *f* (*fst p*) (*snd p*)
⟨*proof*⟩

lemma *prod-cases3* [*cases type*]:
obtains (*fields*) *a b c* **where** *y* = (*a*, *b*, *c*)
⟨*proof*⟩

lemma *prod-induct3* [*case-names fields*, *induct type*]:
(!!*a b c*. *P* (*a*, *b*, *c*)) ==> *P x*
⟨*proof*⟩

lemma *prod-cases4* [*cases type*]:
obtains (*fields*) *a b c d* **where** *y* = (*a*, *b*, *c*, *d*)
⟨*proof*⟩

lemma *prod-induct4* [*case-names fields*, *induct type*]:
(!!*a b c d*. *P* (*a*, *b*, *c*, *d*)) ==> *P x*
⟨*proof*⟩

lemma *prod-cases5* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d\ e$ **where** $y = (a,\ b,\ c,\ d,\ e)$
⟨*proof*⟩

lemma *prod-induct5* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e.\ P\ (a,\ b,\ c,\ d,\ e)) ==> P\ x$
⟨*proof*⟩

lemma *prod-cases6* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d\ e\ f$ **where** $y = (a,\ b,\ c,\ d,\ e,\ f)$
⟨*proof*⟩

lemma *prod-induct6* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f.\ P\ (a,\ b,\ c,\ d,\ e,\ f)) ==> P\ x$
⟨*proof*⟩

lemma *prod-cases7* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d\ e\ f\ g$ **where** $y = (a,\ b,\ c,\ d,\ e,\ f,\ g)$
⟨*proof*⟩

lemma *prod-induct7* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f\ g.\ P\ (a,\ b,\ c,\ d,\ e,\ f,\ g)) ==> P\ x$
⟨*proof*⟩

lemma *split-def*:
$split = (\lambda c\ p.\ c\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

definition *internal-split* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$ **where**
$internal\text{-}split == split$

lemma *internal-split-conv*: $internal\text{-}split\ c\ (a,\ b) = c\ a\ b$
⟨*proof*⟩

⟨*ML*⟩

hide-const *internal-split*

11.3.5 Derived operations

global consts
$curry$ :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

local defs
$curry\text{-}def$: $curry == (\%c\ x\ y.\ c\ (Pair\ x\ y))$

lemma *curry-conv* [*simp*, *code*]: $curry\ f\ a\ b = f\ (a,\ b)$
⟨*proof*⟩

lemma *curryI* [*intro*!]: $f\ (a,\ b) \Longrightarrow curry\ f\ a\ b$
⟨*proof*⟩

lemma *curryD* [*dest*!]: $curry\ f\ a\ b \Longrightarrow f\ (a,\ b)$
⟨*proof*⟩

lemma *curryE*: $curry\ f\ a\ b \Longrightarrow (f\ (a,\ b) \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *curry-split* [*simp*]: $curry\ (split\ f) = f$
⟨*proof*⟩

lemma *split-curry* [*simp*]: $split\ (curry\ f) = f$
⟨*proof*⟩

The composition-uncurry combinator.

notation *fcomp* (**infixl** *o>* *60*)

definition *scomp* :: $('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (**infixl** *o→* *60*)
where
$f\ o\!\!\rightarrow\ g = (\lambda x.\ split\ g\ (f\ x))$

lemma *scomp-apply*: $(f\ o\!\!\rightarrow\ g)\ x = split\ g\ (f\ x)$
⟨*proof*⟩

lemma *Pair-scomp*: $Pair\ x\ o\!\!\rightarrow\ f = f\ x$
⟨*proof*⟩

lemma *scomp-Pair*: $x\ o\!\!\rightarrow\ Pair = x$
⟨*proof*⟩

lemma *scomp-scomp*: $(f\ o\!\!\rightarrow\ g)\ o\!\!\rightarrow\ h = f\ o\!\!\rightarrow\ (\lambda x.\ g\ x\ o\!\!\rightarrow\ h)$
⟨*proof*⟩

lemma *scomp-fcomp*: $(f\ o\!\!\rightarrow\ g)\ o>\ h = f\ o\!\!\rightarrow\ (\lambda x.\ g\ x\ o>\ h)$
⟨*proof*⟩

lemma *fcomp-scomp*: $(f\ o>\ g)\ o\!\!\rightarrow\ h = f\ o>\ (g\ o\!\!\rightarrow\ h)$
⟨*proof*⟩

code-const *scomp*
(*Eval* **infixl** *3* #−>)

no-notation *fcomp* (**infixl** *o>* *60*)
no-notation *scomp* (**infixl** *o→* *60*)

prod-fun — action of the product functor upon functions.

definition *prod-fun* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'd$ **where**
[*code del*]: $prod\text{-}fun\ f\ g = (\lambda(x,\ y).\ (f\ x,\ g\ y))$

lemma *prod-fun* [*simp*, *code*]: *prod-fun f g* (*a*, *b*) = (*f a*, *g b*)
⟨*proof*⟩

lemma *fst-prod-fun*[*simp*]: *fst* (*prod-fun f g x*) = *f* (*fst x*)
⟨*proof*⟩

lemma *snd-prod-fun*[*simp*]: *snd* (*prod-fun f g x*) = *g* (*snd x*)
⟨*proof*⟩

lemma *fst-comp-prod-fun*[*simp*]: *fst* ∘ *prod-fun f g* = *f* ∘ *fst*
⟨*proof*⟩

lemma *snd-comp-prod-fun*[*simp*]: *snd* ∘ *prod-fun f g* = *g* ∘ *snd*
⟨*proof*⟩

lemma *prod-fun-compose*:
prod-fun (*f1 o f2*) (*g1 o g2*) = (*prod-fun f1 g1 o prod-fun f2 g2*)
⟨*proof*⟩

lemma *prod-fun-ident* [*simp*]: *prod-fun* (%*x*. *x*) (%*y*. *y*) = (%*z*. *z*)
⟨*proof*⟩

lemma *prod-fun-imageI* [*intro*]: (*a*, *b*) : *r* ==> (*f a*, *g b*) : *prod-fun f g* ‘ *r*
⟨*proof*⟩

lemma *prod-fun-imageE* [*elim*!]:
assumes *major*: *c*: (*prod-fun f g*)‘*r*
and *cases*: !!*x y*. [| *c*=(*f*(*x*),*g*(*y*)); (*x*,*y*):*r* |] ==> *P*
shows *P*
⟨*proof*⟩

definition *apfst* :: (′*a* ⇒ ′*c*) ⇒ ′*a* × ′*b* ⇒ ′*c* × ′*b* **where**
apfst f = *prod-fun f id*

definition *apsnd* :: (′*b* ⇒ ′*c*) ⇒ ′*a* × ′*b* ⇒ ′*a* × ′*c* **where**
apsnd f = *prod-fun id f*

lemma *apfst-conv* [*simp*, *code*]:
apfst f (*x*, *y*) = (*f x*, *y*)
⟨*proof*⟩

lemma *apsnd-conv* [*simp*, *code*]:
apsnd f (*x*, *y*) = (*x*, *f y*)
⟨*proof*⟩

lemma *fst-apfst* [*simp*]:

fst (*apfst f x*) = *f* (*fst x*)
⟨*proof*⟩

lemma *fst-apsnd* [*simp*]:
fst (*apsnd f x*) = *fst x*
⟨*proof*⟩

lemma *snd-apfst* [*simp*]:
snd (*apfst f x*) = *snd x*
⟨*proof*⟩

lemma *snd-apsnd* [*simp*]:
snd (*apsnd f x*) = *f* (*snd x*)
⟨*proof*⟩

lemma *apfst-compose*:
apfst f (*apfst g x*) = *apfst* (*f* ∘ *g*) *x*
⟨*proof*⟩

lemma *apsnd-compose*:
apsnd f (*apsnd g x*) = *apsnd* (*f* ∘ *g*) *x*
⟨*proof*⟩

lemma *apfst-apsnd* [*simp*]:
apfst f (*apsnd g x*) = (*f* (*fst x*), *g* (*snd x*))
⟨*proof*⟩

lemma *apsnd-apfst* [*simp*]:
apsnd f (*apfst g x*) = (*g* (*fst x*), *f* (*snd x*))
⟨*proof*⟩

lemma *apfst-id* [*simp*] :
apfst id = *id*
⟨*proof*⟩

lemma *apsnd-id* [*simp*] :
apsnd id = *id*
⟨*proof*⟩

lemma *apfst-eq-conv* [*simp*]:
apfst f x = *apfst g x* ⟷ *f* (*fst x*) = *g* (*fst x*)
⟨*proof*⟩

lemma *apsnd-eq-conv* [*simp*]:
apsnd f x = *apsnd g x* ⟷ *f* (*snd x*) = *g* (*snd x*)
⟨*proof*⟩

lemma *apsnd-apfst-commute*:
apsnd f (*apfst g p*) = *apfst g* (*apsnd f p*)

⟨*proof*⟩

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: [*'a set*, *'a* => *'b set*] => (*'a* × *'b*) *set* **where**
Sigma-def: *Sigma A B* == *UN x*:*A*. *UN y*:*B x*. {*Pair x y*}

abbreviation
Times :: [*'a set*, *'b set*] => (*'a* ∗ *'b*) *set*
(**infixr** <∗> *80*) **where**
A <∗> *B* == *Sigma A* (%-. *B*)

notation (*xsymbols*)
Times (**infixr** × *80*)

notation (*HTML* **output**)
Times (**infixr** × *80*)

syntax
-*Sigma* :: [*pttrn*, *'a set*, *'b set*] => (*'a* ∗ *'b*) *set* ((*3SIGMA* -:-./ -) [*0*, *0*, *10*] *10*)
translations
SIGMA x:*A*. *B* == *CONST Sigma A* (%*x*. *B*)

lemma *SigmaI* [*intro*!]: [| *a*:*A*; *b*:*B*(*a*) |] ==> (*a*,*b*) : *Sigma A B*
⟨*proof*⟩

lemma *SigmaE* [*elim*!]:
[| *c*: *Sigma A B*;
!!*x y*.[| *x*:*A*; *y*:*B*(*x*); *c*=(*x*,*y*) |] ==> *P*
|] ==> *P*
— The general elimination rule.
⟨*proof*⟩

Elimination of (*a*, *b*) ∈ *A* × *B* – introduces no eigenvariables.

lemma *SigmaD1*: (*a*, *b*) : *Sigma A B* ==> *a* : *A*
⟨*proof*⟩

lemma *SigmaD2*: (*a*, *b*) : *Sigma A B* ==> *b* : *B a*
⟨*proof*⟩

lemma *SigmaE2*:
[| (*a*, *b*) : *Sigma A B*;
[| *a*:*A*; *b*:*B*(*a*) |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *Sigma-cong*:
⟦*A* = *B*; !!*x*. *x* ∈ *B* ⟹ *C x* = *D x*⟧
⟹ (*SIGMA x*: *A*. *C x*) = (*SIGMA x*: *B*. *D x*)
⟨*proof*⟩

lemma *Sigma-mono*: [| $A <= C$; !!x. x:A ==> $B\ x <= D\ x$ |] ==> *Sigma* $A\ B$ <= *Sigma* $C\ D$
⟨*proof*⟩

lemma *Sigma-empty1* [*simp*]: *Sigma* {} $B = \{\}$
⟨*proof*⟩

lemma *Sigma-empty2* [*simp*]: $A <*> \{\} = \{\}$
⟨*proof*⟩

lemma *UNIV-Times-UNIV* [*simp*]: *UNIV* <*> *UNIV* = *UNIV*
⟨*proof*⟩

lemma *Compl-Times-UNIV1* [*simp*]: $-(UNIV <*> A) = UNIV <*> (-A)$
⟨*proof*⟩

lemma *Compl-Times-UNIV2* [*simp*]: $-(A <*> UNIV) = (-A) <*> UNIV$
⟨*proof*⟩

lemma *mem-Sigma-iff* [*iff*]: $((a,b)$: *Sigma* $A\ B) = (a$:A & b:$B(a))$
⟨*proof*⟩

lemma *Times-subset-cancel2*: x:C ==> $(A <*> C <= B <*> C) = (A <= B)$
⟨*proof*⟩

lemma *Times-eq-cancel2*: x:C ==> $(A <*> C = B <*> C) = (A = B)$
⟨*proof*⟩

lemma *SetCompr-Sigma-eq*:
Collect (*split* (%$x\ y$. $P\ x$ & $Q\ x\ y$)) = (*SIGMA* x:*Collect* P. *Collect* $(Q\ x)$)
⟨*proof*⟩

lemma *Collect-split* [*simp*]: $\{(a,b).\ P\ a$ & $Q\ b\} =$ *Collect* $P <*>$ *Collect* Q
⟨*proof*⟩

lemma *UN-Times-distrib*:
$(UN\ (a,b)$:$(A <*> B).\ E\ a <*> F\ b) = (UNION\ A\ E) <*> (UNION\ B\ F)$
— Suggested by Pierre Chartier
⟨*proof*⟩

lemma *split-paired-Ball-Sigma* [*simp*,*no-atp*]:
(*ALL* z: *Sigma* $A\ B$. $P\ z) = (ALL\ x$:A. *ALL* y: $B\ x$. $P(x,y))$
⟨*proof*⟩

lemma *split-paired-Bex-Sigma* [*simp*,*no-atp*]:
(*EX* z: *Sigma* $A\ B$. $P\ z) = (EX\ x$:A. *EX* y: $B\ x$. $P(x,y))$
⟨*proof*⟩

lemma *Sigma-Un-distrib1*: $(SIGMA\ i{:}I\ Un\ J.\ C(i)) = (SIGMA\ i{:}I.\ C(i))\ Un\ (SIGMA\ j{:}J.\ C(j))$
⟨*proof*⟩

lemma *Sigma-Un-distrib2*: $(SIGMA\ i{:}I.\ A(i)\ Un\ B(i)) = (SIGMA\ i{:}I.\ A(i))\ Un\ (SIGMA\ i{:}I.\ B(i))$
⟨*proof*⟩

lemma *Sigma-Int-distrib1*: $(SIGMA\ i{:}I\ Int\ J.\ C(i)) = (SIGMA\ i{:}I.\ C(i))\ Int\ (SIGMA\ j{:}J.\ C(j))$
⟨*proof*⟩

lemma *Sigma-Int-distrib2*: $(SIGMA\ i{:}I.\ A(i)\ Int\ B(i)) = (SIGMA\ i{:}I.\ A(i))\ Int\ (SIGMA\ i{:}I.\ B(i))$
⟨*proof*⟩

lemma *Sigma-Diff-distrib1*: $(SIGMA\ i{:}I - J.\ C(i)) = (SIGMA\ i{:}I.\ C(i)) - (SIGMA\ j{:}J.\ C(j))$
⟨*proof*⟩

lemma *Sigma-Diff-distrib2*: $(SIGMA\ i{:}I.\ A(i) - B(i)) = (SIGMA\ i{:}I.\ A(i)) - (SIGMA\ i{:}I.\ B(i))$
⟨*proof*⟩

lemma *Sigma-Union*: $Sigma\ (Union\ X)\ B = (UN\ A{:}X.\ Sigma\ A\ B)$
⟨*proof*⟩

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A\ Un\ B) <*> C = (A <*> C)\ Un\ (B <*> C)$
⟨*proof*⟩

lemma *Times-Int-distrib1*: $(A\ Int\ B) <*> C = (A <*> C)\ Int\ (B <*> C)$
⟨*proof*⟩

lemma *Times-Diff-distrib1*: $(A - B) <*> C = (A <*> C) - (B <*> C)$
⟨*proof*⟩

lemma *Times-empty*[*simp*]: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
⟨*proof*⟩

lemma *fst-image-times*[*simp*]: $fst\ `\ (A \times B) = (if\ B = \{\}\ then\ \{\}\ else\ A)$
⟨*proof*⟩

lemma *snd-image-times*[*simp*]: $snd\ `\ (A \times B) = (if\ A = \{\}\ then\ \{\}\ else\ B)$
⟨*proof*⟩

lemma *insert-times-insert*[*simp*]:
$insert\ a\ A \times insert\ b\ B =$

insert (*a*,*b*) (*A* × *insert b B* ∪ *insert a A* × *B*)
⟨*proof*⟩

lemma *vimage-Times*: *f* −‘ (*A* × *B*) = ((*fst* ∘ *f*) −‘ *A*) ∩ ((*snd* ∘ *f*) −‘ *B*)
⟨*proof*⟩

The following *prod-fun* lemmas are due to Joachim Breitner:

lemma *prod-fun-inj-on*:
assumes *inj-on f A* **and** *inj-on g B*
shows *inj-on* (*prod-fun f g*) (*A* × *B*)
⟨*proof*⟩

lemma *prod-fun-surj*:
assumes *surj f* **and** *surj g*
shows *surj* (*prod-fun f g*)
⟨*proof*⟩

lemma *prod-fun-surj-on*:
assumes *f* ‘ *A* = *A*′ **and** *g* ‘ *B* = *B*′
shows *prod-fun f g* ‘ (*A* × *B*) = *A*′ × *B*′
⟨*proof*⟩

lemma *swap-inj-on*:
inj-on (λ(*i*, *j*). (*j*, *i*)) *A*
⟨*proof*⟩

lemma *swap-product*:
(%(*i*, *j*). (*j*, *i*)) ‘ (*A* × *B*) = *B* × *A*
⟨*proof*⟩

lemma *image-split-eq-Sigma*:
(λ*x*. (*f x*, *g x*)) ‘ *A* = *Sigma* (*f* ‘ *A*) (λ*x*. *g* ‘ (*f* −‘ {*x*} ∩ *A*))
⟨*proof*⟩

11.4 Inductively defined sets

⟨*ML*⟩

11.5 Legacy theorem bindings and duplicates

lemma *PairE*:
obtains *x y* **where** *p* = (*x*, *y*)
⟨*proof*⟩

lemma *Pair-inject*:
assumes (*a*, *b*) = (*a*′, *b*′)
and *a* = *a*′ ==> *b* = *b*′ ==> *R*
shows *R*
⟨*proof*⟩

lemmas *Pair-eq* = *prod.inject*

lemmas *split* = *split-conv* — for backwards compatibility

end

12 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*
imports *Typedef Inductive Fun*
begin

12.1 Construction of the sum type and its basic abstract operations

definition *Inl-Rep* :: $'a \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \Rightarrow bool$ **where**
Inl-Rep a x y p $\longleftrightarrow x = a \wedge p$

definition *Inr-Rep* :: $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \Rightarrow bool$ **where**
Inr-Rep b x y p $\longleftrightarrow y = b \wedge \neg p$

global

typedef (*Sum*) $('a, 'b)$ + (**infixr** + *10*) = $\{f.\ (\exists\, a.\ f = \textit{Inl-Rep}\ (a::'a)) \vee (\exists\, b.\ f = \textit{Inr-Rep}\ (b::'b))\}$
⟨*proof*⟩

local

lemma *Inl-RepI*: *Inl-Rep a* ∈ *Sum*
⟨*proof*⟩

lemma *Inr-RepI*: *Inr-Rep b* ∈ *Sum*
⟨*proof*⟩

lemma *inj-on-Abs-Sum*: *A* ⊆ *Sum* ⟹ *inj-on Abs-Sum A*
⟨*proof*⟩

lemma *Inl-Rep-inject*: *inj-on Inl-Rep A*
⟨*proof*⟩

lemma *Inr-Rep-inject*: *inj-on Inr-Rep A*
⟨*proof*⟩

lemma *Inl-Rep-not-Inr-Rep*: *Inl-Rep a* ≠ *Inr-Rep b*
⟨*proof*⟩

definition *Inl* :: $'a \Rightarrow 'a + 'b$ **where**
Inl = *Abs-Sum* ∘ *Inl-Rep*

definition *Inr* :: $'b \Rightarrow 'a + 'b$ **where**
Inr = *Abs-Sum* ∘ *Inr-Rep*

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
⟨*proof*⟩

lemma *Inl-inject*: *Inl x* = *Inl y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
⟨*proof*⟩

lemma *Inr-inject*: *Inr x* = *Inr y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *Inl-not-Inr*: *Inl a* ≠ *Inr b*
⟨*proof*⟩

lemma *Inr-not-Inl*: *Inr b* ≠ *Inl a*
⟨*proof*⟩

lemma *sumE*:
assumes ⋀*x*::'*a*. *s* = *Inl x* ⟹ *P*
and ⋀*y*::'*b*. *s* = *Inr y* ⟹ *P*
shows *P*
⟨*proof*⟩

rep-datatype (*sum*) *Inl Inr*
⟨*proof*⟩

12.2 Projections

lemma *sum-case-KK* [*simp*]: *sum-case* (λ*x*. *a*) (λ*x*. *a*) = (λ*x*. *a*)
⟨*proof*⟩

lemma *surjective-sum*: *sum-case* (λ*x*::'*a*. *f* (*Inl x*)) (λ*y*::'*b*. *f* (*Inr y*)) = *f*
⟨*proof*⟩

lemma *sum-case-inject*:
assumes *a*: *sum-case f1 f2* = *sum-case g1 g2*
assumes *r*: *f1* = *g1* ⟹ *f2* = *g2* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *sum-case-weak-cong*:
s = *t* ⟹ *sum-case f g s* = *sum-case f g t*

— Prevents simplification of f and g: much faster.
⟨*proof*⟩

primrec *Projl* :: $'a + 'b \Rightarrow 'a$ **where**
Projl-Inl: $Projl\ (Inl\ x) = x$

primrec *Projr* :: $'a + 'b \Rightarrow 'b$ **where**
Projr-Inr: $Projr\ (Inr\ x) = x$

primrec *Suml* :: $('a \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$ **where**
$Suml\ f\ (Inl\ x) = f\ x$

primrec *Sumr* :: $('b \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$ **where**
$Sumr\ f\ (Inr\ x) = f\ x$

lemma *Suml-inject*:
assumes $Suml\ f = Suml\ g$ **shows** $f = g$
⟨*proof*⟩

lemma *Sumr-inject*:
assumes $Sumr\ f = Sumr\ g$ **shows** $f = g$
⟨*proof*⟩

12.3 The Disjoint Sum of Sets

definition *Plus* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a + 'b)\ set$ (**infixr** <+> *65*) **where**
$A <+> B = Inl\ `\ A \cup Inr\ `\ B$

lemma *InlI* [*intro*!]: $a \in A \Longrightarrow Inl\ a \in A <+> B$
⟨*proof*⟩

lemma *InrI* [*intro*!]: $b \in B \Longrightarrow Inr\ b \in A <+> B$
⟨*proof*⟩

Exhaustion rule for sums, a degenerate form of induction

lemma *PlusE* [*elim*!]:
$u \in A <+> B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow u = Inl\ x \Longrightarrow P) \Longrightarrow (\bigwedge y.\ y \in B \Longrightarrow u = Inr\ y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Plus-eq-empty-conv* [*simp*]: $A <+> B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
⟨*proof*⟩

lemma *UNIV-Plus-UNIV* [*simp*]: $UNIV <+> UNIV = UNIV$
⟨*proof*⟩

hide-const (**open**) *Suml Sumr Projl Projr*

end

13 Rings: Rings

theory *Rings*
imports *Groups*
begin

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
  assumes *left-distrib*[*algebra-simps*, *field-simps*]: $(a + b) * c = a * c + b * c$
  assumes *right-distrib*[*algebra-simps*, *field-simps*]: $a * (b + c) = a * b + a * c$
begin

For the *combine-numerals* simproc

lemma *combine-common-factor*:
  $a * e + (b * e + c) = (a + b) * e + c$
⟨*proof*⟩

end

class *mult-zero* = *times* + *zero* +
  assumes *mult-zero-left* [*simp*]: $0 * a = 0$
  assumes *mult-zero-right* [*simp*]: $a * 0 = 0$

class *semiring-0* = *semiring* + *comm-monoid-add* + *mult-zero*

class *semiring-0-cancel* = *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0*
⟨*proof*⟩

end

class *comm-semiring* = *ab-semigroup-add* + *ab-semigroup-mult* +
  assumes *distrib*: $(a + b) * c = a * c + b * c$
begin

subclass *semiring*
⟨*proof*⟩

end

class *comm-semiring-0* = *comm-semiring* + *comm-monoid-add* + *mult-zero*
begin

subclass *semiring-0* ⟨*proof*⟩

end

class *comm-semiring-0-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *comm-semiring-0* ⟨*proof*⟩

end

class *zero-neq-one* = *zero* + *one* +
  assumes *zero-neq-one* [*simp*]: *0* ≠ *1*
begin

lemma *one-neq-zero* [*simp*]: *1* ≠ *0*
⟨*proof*⟩

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** *dvd 50*) **where**
  [*code del*]: *b dvd a* ⟷ (∃ *k*. *a* = *b* ∗ *k*)

lemma *dvdI* [*intro?*]: *a* = *b* ∗ *k* ⟹ *b dvd a*
  ⟨*proof*⟩

lemma *dvdE* [*elim?*]: *b dvd a* ⟹ (⋀*k*. *a* = *b* ∗ *k* ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

end

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*
+ *dvd*

begin

subclass *semiring-1* ⟨*proof*⟩

lemma *dvd-refl*[*simp*]: *a dvd a*
⟨*proof*⟩

lemma *dvd-trans*:
  assumes *a dvd b* **and** *b dvd c*
  shows *a dvd c*

⟨*proof*⟩

lemma *dvd-0-left-iff* [*no-atp*, *simp*]: *0 dvd a* ⟷ *a = 0*
⟨*proof*⟩

lemma *dvd-0-right* [*iff*]: *a dvd 0*
⟨*proof*⟩

lemma *one-dvd* [*simp*]: *1 dvd a*
⟨*proof*⟩

lemma *dvd-mult*[*simp*]: *a dvd c* ⟹ *a dvd* (*b* ∗ *c*)
⟨*proof*⟩

lemma *dvd-mult2*[*simp*]: *a dvd b* ⟹ *a dvd* (*b* ∗ *c*)
⟨*proof*⟩

lemma *dvd-triv-right* [*simp*]: *a dvd b* ∗ *a*
⟨*proof*⟩

lemma *dvd-triv-left* [*simp*]: *a dvd a* ∗ *b*
⟨*proof*⟩

lemma *mult-dvd-mono*:
assumes *a dvd b*
and *c dvd d*
shows *a* ∗ *c dvd b* ∗ *d*
⟨*proof*⟩

lemma *dvd-mult-left*: *a* ∗ *b dvd c* ⟹ *a dvd c*
⟨*proof*⟩

lemma *dvd-mult-right*: *a* ∗ *b dvd c* ⟹ *b dvd c*
⟨*proof*⟩

lemma *dvd-0-left*: *0 dvd a* ⟹ *a = 0*
⟨*proof*⟩

lemma *dvd-add*[*simp*]:
assumes *a dvd b* **and** *a dvd c* **shows** *a dvd* (*b* + *c*)
⟨*proof*⟩

end

class *no-zero-divisors = zero + times +*
assumes *no-zero-divisors*: *a* ≠ *0* ⟹ *b* ≠ *0* ⟹ *a* ∗ *b* ≠ *0*
begin

lemma *divisors-zero*:

assumes $a * b = 0$
shows $a = 0 \vee b = 0$
⟨*proof*⟩

end

class *semiring-1-cancel* = *semiring* + *cancel-comm-monoid-add*
+ *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *semiring-1* ⟨*proof*⟩

end

class *comm-semiring-1-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
+ *zero-neq-one* + *comm-monoid-mult*
begin

subclass *semiring-1-cancel* ⟨*proof*⟩
subclass *comm-semiring-0-cancel* ⟨*proof*⟩
subclass *comm-semiring-1* ⟨*proof*⟩

end

class *ring* = *semiring* + *ab-group-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

Distribution rules

lemma *minus-mult-left*: $- (a * b) = - a * b$
⟨*proof*⟩

lemma *minus-mult-right*: $- (a * b) = a * - b$
⟨*proof*⟩

Extract signs from products

lemmas *mult-minus-left* [*simp*, *no-atp*] = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* [*simp*,*no-atp*] = *minus-mult-right* [*symmetric*]

lemma *minus-mult-minus* [*simp*]: $- a * - b = a * b$
⟨*proof*⟩

lemma *minus-mult-commute*: $- a * b = a * - b$
⟨*proof*⟩

lemma *right-diff-distrib*[*algebra-simps*, *field-simps*]: $a * (b - c) = a * b - a * c$

⟨*proof*⟩

lemma *left-diff-distrib*[*algebra-simps*, *field-simps*]: $(a - b) * c = a * c - b * c$
⟨*proof*⟩

lemmas *ring-distribs*[*no-atp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

lemma *eq-add-iff1*:
$a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
⟨*proof*⟩

lemma *eq-add-iff2*:
$a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$
⟨*proof*⟩

end

lemmas *ring-distribs*[*no-atp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* ⟨*proof*⟩
subclass *comm-semiring-0-cancel* ⟨*proof*⟩

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* ⟨*proof*⟩

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*

begin

subclass *ring-1* ⟨*proof*⟩
subclass *comm-semiring-1-cancel* ⟨*proof*⟩

lemma *dvd-minus-iff* [*simp*]: *x dvd* − *y* ⟷ *x dvd y*
⟨*proof*⟩

lemma *minus-dvd-iff* [*simp*]: − *x dvd y* ⟷ *x dvd y*
⟨*proof*⟩

lemma *dvd-diff*[*simp*]: $x\ dvd\ y \Longrightarrow x\ dvd\ z \Longrightarrow x\ dvd\ (y - z)$
⟨*proof*⟩

end

class *ring-no-zero-divisors* = *ring* + *no-zero-divisors*
begin

lemma *mult-eq-0-iff* [*simp*]:
shows $a * b = 0 \longleftrightarrow (a = 0 \vee b = 0)$
⟨*proof*⟩

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*, *no-atp*]:
$a * c = b * c \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

lemma *mult-cancel-left* [*simp*, *no-atp*]:
$c * a = c * b \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *square-eq-1-iff*:
$x * x = 1 \longleftrightarrow x = 1 \vee x = -1$
⟨*proof*⟩

lemma *mult-cancel-right1* [*simp*]:
$c = b * c \longleftrightarrow c = 0 \vee b = 1$
⟨*proof*⟩

lemma *mult-cancel-right2* [*simp*]:
$a * c = c \longleftrightarrow c = 0 \vee a = 1$
⟨*proof*⟩

lemma *mult-cancel-left1* [*simp*]:
$c = c * b \longleftrightarrow c = 0 \vee b = 1$
⟨*proof*⟩

lemma *mult-cancel-left2* [*simp*]:
$c * a = c \longleftrightarrow c = 0 \vee a = 1$
⟨*proof*⟩

end

class *idom* = *comm-ring-1* + *no-zero-divisors*

begin

subclass *ring-1-no-zero-divisors* ⟨*proof*⟩

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow (a = b \vee a = - b)$
⟨*proof*⟩

lemma *dvd-mult-cancel-right* [*simp*]:
$a * c$ *dvd* $b * c \longleftrightarrow c = 0 \vee a$ *dvd* b
⟨*proof*⟩

lemma *dvd-mult-cancel-left* [*simp*]:
$c * a$ *dvd* $c * b \longleftrightarrow c = 0 \vee a$ *dvd* b
⟨*proof*⟩

end

class *inverse* =
fixes *inverse* :: $'a \Rightarrow 'a$
and *divide* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $'/$ 70)

class *division-ring* = *ring-1* + *inverse* +
assumes *left-inverse* [*simp*]: $a \neq 0 \Longrightarrow$ *inverse* $a * a = 1$
assumes *right-inverse* [*simp*]: $a \neq 0 \Longrightarrow a *$ *inverse* $a = 1$
assumes *divide-inverse*: a / $b = a *$ *inverse* b
begin

subclass *ring-1-no-zero-divisors*
⟨*proof*⟩

lemma *nonzero-imp-inverse-nonzero*:
$a \neq 0 \Longrightarrow$ *inverse* $a \neq 0$
⟨*proof*⟩

lemma *inverse-zero-imp-zero*:
inverse $a = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inverse-unique*:
assumes *ab*: $a * b = 1$
shows *inverse* $a = b$
⟨*proof*⟩

lemma *nonzero-inverse-minus-eq*:
$a \neq 0 \Longrightarrow$ *inverse* $(- a) = -$ *inverse* a
⟨*proof*⟩

lemma *nonzero-inverse-inverse-eq*:
$a \neq 0 \Longrightarrow$ *inverse* (*inverse* a) $= a$

⟨*proof*⟩

lemma *nonzero-inverse-eq-imp-eq*:
assumes *inverse a* = *inverse b* **and** *a* ≠ *0* **and** *b* ≠ *0*
shows *a* = *b*
⟨*proof*⟩

lemma *inverse-1* [*simp*]: *inverse 1* = *1*
⟨*proof*⟩

lemma *nonzero-inverse-mult-distrib*:
assumes *a* ≠ *0* **and** *b* ≠ *0*
shows *inverse* (*a* ∗ *b*) = *inverse b* ∗ *inverse a*
⟨*proof*⟩

lemma *division-ring-inverse-add*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *inverse a* + *inverse b* = *inverse a* ∗ (*a* + *b*) ∗ *inverse b*
⟨*proof*⟩

lemma *division-ring-inverse-diff*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *inverse a* − *inverse b* = *inverse a* ∗ (*b* − *a*) ∗ *inverse b*
⟨*proof*⟩

lemma *right-inverse-eq*: *b* ≠ *0* ⟹ *a* / *b* = *1* ⟷ *a* = *b*
⟨*proof*⟩

lemma *nonzero-inverse-eq-divide*: *a* ≠ *0* ⟹ *inverse a* = *1* / *a*
⟨*proof*⟩

lemma *divide-self* [*simp*]: *a* ≠ *0* ⟹ *a* / *a* = *1*
⟨*proof*⟩

lemma *divide-zero-left* [*simp*]: *0* / *a* = *0*
⟨*proof*⟩

lemma *inverse-eq-divide*: *inverse a* = *1* / *a*
⟨*proof*⟩

lemma *add-divide-distrib*: (*a*+*b*) / *c* = *a*/*c* + *b*/*c*
⟨*proof*⟩

lemma *divide-1* [*simp*]: *a* / *1* = *a*
⟨*proof*⟩

lemma *times-divide-eq-right* [*simp*]: *a* ∗ (*b* / *c*) = (*a* ∗ *b*) / *c*
⟨*proof*⟩

lemma *minus-divide-left*: − (*a* / *b*) = (−*a*) / *b*
⟨*proof*⟩

lemma *nonzero-minus-divide-right*: $b \neq 0 \Longrightarrow -(a \mathbin{/} b) = a \mathbin{/} (-b)$
⟨*proof*⟩

lemma *nonzero-minus-divide-divide*: $b \neq 0 \Longrightarrow (-a) \mathbin{/} (-b) = a \mathbin{/} b$
⟨*proof*⟩

lemma *divide-minus-left* [*simp*, *no-atp*]: $(-a) \mathbin{/} b = -(a \mathbin{/} b)$
⟨*proof*⟩

lemma *diff-divide-distrib*: $(a - b) \mathbin{/} c = a \mathbin{/} c - b \mathbin{/} c$
⟨*proof*⟩

lemma *nonzero-eq-divide-eq* [*field-simps*]: $c \neq 0 \Longrightarrow a = b \mathbin{/} c \longleftrightarrow a * c = b$
⟨*proof*⟩

lemma *nonzero-divide-eq-eq* [*field-simps*]: $c \neq 0 \Longrightarrow b \mathbin{/} c = a \longleftrightarrow b = a * c$
⟨*proof*⟩

lemma *divide-eq-imp*: $c \neq 0 \Longrightarrow b = a * c \Longrightarrow b \mathbin{/} c = a$
⟨*proof*⟩

lemma *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b \mathbin{/} c$
⟨*proof*⟩

end

class *division-ring-inverse-zero* = *division-ring* +
assumes *inverse-zero* [*simp*]: *inverse* 0 = 0
begin

lemma *divide-zero* [*simp*]:
$a \mathbin{/} 0 = 0$
⟨*proof*⟩

lemma *divide-self-if* [*simp*]:
$a \mathbin{/} a = (\textit{if } a = 0 \textit{ then } 0 \textit{ else } 1)$
⟨*proof*⟩

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
inverse $a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *inverse-minus-eq* [*simp*]:
inverse $(-a) = -$ *inverse* a
⟨*proof*⟩

lemma *inverse-eq-imp-eq*:
inverse $a =$ *inverse* $b \Longrightarrow a = b$

⟨*proof*⟩

lemma *inverse-eq-iff-eq* [*simp*]:
  inverse a = *inverse b* ⟷ *a* = *b*
  ⟨*proof*⟩

lemma *inverse-inverse-eq* [*simp*]:
  inverse (*inverse a*) = *a*
⟨*proof*⟩

end

class *mult-mono* = *times* + *zero* + *ord* +
  assumes *mult-left-mono*: *a* ≤ *b* ⟹ 0 ≤ *c* ⟹ *c* ∗ *a* ≤ *c* ∗ *b*
  assumes *mult-right-mono*: *a* ≤ *b* ⟹ 0 ≤ *c* ⟹ *a* ∗ *c* ≤ *b* ∗ *c*

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

class *ordered-semiring* = *mult-mono* + *semiring-0* + *ordered-ab-semigroup-add*
begin

lemma *mult-mono*:
  a ≤ *b* ⟹ *c* ≤ *d* ⟹ 0 ≤ *b* ⟹ 0 ≤ *c*
    ⟹ *a* ∗ *c* ≤ *b* ∗ *d*
⟨*proof*⟩

lemma *mult-mono′*:
  a ≤ *b* ⟹ *c* ≤ *d* ⟹ 0 ≤ *a* ⟹ 0 ≤ *c*
    ⟹ *a* ∗ *c* ≤ *b* ∗ *d*
⟨*proof*⟩

end

class *ordered-cancel-semiring* = *mult-mono* + *ordered-ab-semigroup-add*
  + *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩
subclass *ordered-semiring* ⟨*proof*⟩

lemma *mult-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * b$
⟨*proof*⟩

lemma *mult-nonneg-nonpos*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow a * b \leq 0$
⟨*proof*⟩

lemma *mult-nonpos-nonneg*: $a \leq 0 \Longrightarrow 0 \leq b \Longrightarrow a * b \leq 0$
⟨*proof*⟩

Legacy - use *mult-nonpos-nonneg*

lemma *mult-nonneg-nonpos2*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow b * a \leq 0$
⟨*proof*⟩

lemma *split-mult-neg-le*: $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \Longrightarrow a * b \leq 0$
⟨*proof*⟩

end

class *linordered-semiring* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add* + *mult-mono*
begin

subclass *ordered-cancel-semiring* ⟨*proof*⟩

subclass *ordered-comm-monoid-add* ⟨*proof*⟩

lemma *mult-left-less-imp-less*:
$c * a < c * b \Longrightarrow 0 \leq c \Longrightarrow a < b$
⟨*proof*⟩

lemma *mult-right-less-imp-less*:
$a * c < b * c \Longrightarrow 0 \leq c \Longrightarrow a < b$
⟨*proof*⟩

end

class *linordered-semiring-1* = *linordered-semiring* + *semiring-1*
begin

lemma *convex-bound-le*:
assumes $x \leq a$ $y \leq a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
shows $u * x + v * y \leq a$
⟨*proof*⟩

end

class *linordered-semiring-strict* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add* +

assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *linordered-semiring*
⟨*proof*⟩

lemma *mult-left-le-imp-le*:
$c * a \leq c * b \Longrightarrow 0 < c \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *mult-right-le-imp-le*:
$a * c \leq b * c \Longrightarrow 0 < c \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *mult-pos-pos*: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
⟨*proof*⟩

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
⟨*proof*⟩

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$
⟨*proof*⟩

Legacy - use *mult-neg-pos*

lemma *mult-pos-neg2*: $0 < a \Longrightarrow b < 0 \Longrightarrow b * a < 0$
⟨*proof*⟩

lemma *zero-less-mult-pos*:
$0 < a * b \Longrightarrow 0 < a \Longrightarrow 0 < b$
⟨*proof*⟩

lemma *zero-less-mult-pos2*:
$0 < b * a \Longrightarrow 0 < a \Longrightarrow 0 < b$
⟨*proof*⟩

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ **and** $c < d$ **and** $0 < b$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
assumes $a < b$ **and** $c < d$ **and** $0 \leq a$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-less-le-imp-less*:
assumes $a < b$ **and** $c \leq d$ **and** $0 \leq a$ **and** $0 < c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-less-imp-less-left*:
assumes *less*: $c * a < c * b$ **and** *nonneg*: $0 \leq c$
shows $a < b$
⟨*proof*⟩

lemma *mult-less-imp-less-right*:
assumes *less*: $a * c < b * c$ **and** *nonneg*: $0 \leq c$
shows $a < b$
⟨*proof*⟩

end

class *linordered-semiring-1-strict* = *linordered-semiring-strict* + *semiring-1*
begin

subclass *linordered-semiring-1* ⟨*proof*⟩

lemma *convex-bound-lt*:
assumes $x < a$ $y < a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
shows $u * x + v * y < a$
⟨*proof*⟩

end

class *mult-mono1* = *times* + *zero* + *ord* +
assumes *mult-mono1*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$

class *ordered-comm-semiring* = *comm-semiring-0*
+ *ordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *ordered-semiring*
⟨*proof*⟩

end

class *ordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *ordered-ab-semigroup-add* + *mult-mono1*

begin

subclass *ordered-comm-semiring* ⟨*proof*⟩
subclass *ordered-cancel-semiring* ⟨*proof*⟩

end

class *linordered-comm-semiring-strict* = *comm-semiring-0* + *linordered-cancel-ab-semigroup-add* +
  assumes *mult-strict-left-mono-comm*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *linordered-semiring-strict*
⟨*proof*⟩

subclass *ordered-cancel-comm-semiring*
⟨*proof*⟩

end

class *ordered-ring* = *ring* + *ordered-cancel-semiring*
begin

subclass *ordered-ab-group-add* ⟨*proof*⟩

lemma *less-add-iff1*:
  $a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
⟨*proof*⟩

lemma *less-add-iff2*:
  $a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$
⟨*proof*⟩

lemma *le-add-iff1*:
  $a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
⟨*proof*⟩

lemma *le-add-iff2*:
  $a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
⟨*proof*⟩

lemma *mult-left-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow c * a \leq c * b$
  ⟨*proof*⟩

lemma *mult-right-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a * c \leq b * c$
  ⟨*proof*⟩

lemma *mult-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a * b$
⟨*proof*⟩

lemma *split-mult-pos-le*:
$(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a * b$
⟨*proof*⟩

end

class *linordered-ring* = *ring* + *linordered-semiring* + *linordered-ab-group-add* + *abs-if*
begin

subclass *ordered-ring* ⟨*proof*⟩

subclass *ordered-ab-group-add-abs*
⟨*proof*⟩

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
⟨*proof*⟩

lemma *not-square-less-zero* [*simp*]: $\neg (a * a < 0)$
⟨*proof*⟩

end

class *linordered-ring-strict* = *ring* + *linordered-semiring-strict*
+ *ordered-ab-group-add* + *abs-if*
begin

subclass *linordered-ring* ⟨*proof*⟩

lemma *mult-strict-left-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow c * a < c * b$
⟨*proof*⟩

lemma *mult-strict-right-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow a * c < b * c$
⟨*proof*⟩

lemma *mult-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow 0 < a * b$
⟨*proof*⟩

subclass *ring-no-zero-divisors*
⟨*proof*⟩

lemma *zero-less-mult-iff*:
$0 < a * b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
⟨*proof*⟩

lemma *zero-le-mult-iff*:
$0 \leq a * b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$
⟨*proof*⟩

lemma *mult-less-0-iff*:
$a * b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
⟨*proof*⟩

lemma *mult-le-0-iff*:
$a * b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
⟨*proof*⟩

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations $\leq$ and equality.

These "disjunction" versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

lemma *mult-less-cancel-right-disj*:
$a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
⟨*proof*⟩

lemma *mult-less-cancel-left-disj*:
$c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
⟨*proof*⟩

The "conjunction of implication" lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*:
$a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
⟨*proof*⟩

lemma *mult-less-cancel-left*:
$c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
⟨*proof*⟩

lemma *mult-le-cancel-right*:
$a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
⟨*proof*⟩

lemma *mult-le-cancel-left*:
$c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
⟨*proof*⟩

lemma *mult-le-cancel-left-pos*:
$0 < c \Longrightarrow c * a \leq c * b \longleftrightarrow a \leq b$
⟨*proof*⟩

lemma *mult-le-cancel-left-neg*:
$c < 0 \Longrightarrow c * a \leq c * b \longleftrightarrow b \leq a$

⟨*proof*⟩

lemma *mult-less-cancel-left-pos*:
  $0 < c \Longrightarrow c * a < c * b \longleftrightarrow a < b$
⟨*proof*⟩

lemma *mult-less-cancel-left-neg*:
  $c < 0 \Longrightarrow c * a < c * b \longleftrightarrow b < a$
⟨*proof*⟩

end

lemmas *mult-sign-intros* =
  mult-nonneg-nonneg mult-nonneg-nonpos
  mult-nonpos-nonneg mult-nonpos-nonpos
  mult-pos-pos mult-pos-neg
  mult-neg-pos mult-neg-neg

class *ordered-comm-ring* = *comm-ring* + *ordered-comm-semiring*
begin

subclass *ordered-ring* ⟨*proof*⟩
subclass *ordered-cancel-comm-semiring* ⟨*proof*⟩

end

class *linordered-semidom* = *comm-semiring-1-cancel* + *linordered-comm-semiring-strict*
+

  assumes *zero-less-one* [*simp*]: $0 < 1$
begin

lemma *pos-add-strict*:
  shows $0 < a \Longrightarrow b < c \Longrightarrow b < a + c$
  ⟨*proof*⟩

lemma *zero-le-one* [*simp*]: $0 \leq 1$
⟨*proof*⟩

lemma *not-one-le-zero* [*simp*]: $\neg\ 1 \leq 0$
⟨*proof*⟩

lemma *not-one-less-zero* [*simp*]: $\neg\ 1 < 0$
⟨*proof*⟩

lemma *less-1-mult*:
  assumes $1 < m$ **and** $1 < n$
  shows $1 < m * n$
  ⟨*proof*⟩

end

class *linordered-idom* = *comm-ring-1* +
linordered-comm-semiring-strict + *ordered-ab-group-add* +
abs-if + *sgn-if*

begin

subclass *linordered-semiring-1-strict* ⟨*proof*⟩
subclass *linordered-ring-strict* ⟨*proof*⟩
subclass *ordered-comm-ring* ⟨*proof*⟩
subclass *idom* ⟨*proof*⟩

subclass *linordered-semidom*
⟨*proof*⟩

lemma *linorder-neqE-linordered-idom*:
assumes $x \neq y$ **obtains** $x < y \mid y < x$
⟨*proof*⟩

These cancellation simprules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
$c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
⟨*proof*⟩

lemma *mult-le-cancel-right2*:
$a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
⟨*proof*⟩

lemma *mult-le-cancel-left1*:
$c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
⟨*proof*⟩

lemma *mult-le-cancel-left2*:
$c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
⟨*proof*⟩

lemma *mult-less-cancel-right1*:
$c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
⟨*proof*⟩

lemma *mult-less-cancel-right2*:
$a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
⟨*proof*⟩

lemma *mult-less-cancel-left1*:
$c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

⟨*proof*⟩

lemma *mult-less-cancel-left2*:
  $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
⟨*proof*⟩

lemma *sgn-sgn* [*simp*]:
  $sgn\ (sgn\ a) = sgn\ a$
⟨*proof*⟩

lemma *sgn-0-0*:
  $sgn\ a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *sgn-1-pos*:
  $sgn\ a = 1 \longleftrightarrow a > 0$
⟨*proof*⟩

lemma *sgn-1-neg*:
  $sgn\ a = -\ 1 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *sgn-pos* [*simp*]:
  $0 < a \Longrightarrow sgn\ a = 1$
⟨*proof*⟩

lemma *sgn-neg* [*simp*]:
  $a < 0 \Longrightarrow sgn\ a = -\ 1$
⟨*proof*⟩

lemma *sgn-times*:
  $sgn\ (a * b) = sgn\ a * sgn\ b$
⟨*proof*⟩

lemma *abs-sgn*: $|k| = k * sgn\ k$
⟨*proof*⟩

lemma *sgn-greater* [*simp*]:
  $0 < sgn\ a \longleftrightarrow 0 < a$
  ⟨*proof*⟩

lemma *sgn-less* [*simp*]:
  $sgn\ a < 0 \longleftrightarrow a < 0$
  ⟨*proof*⟩

lemma *abs-dvd-iff* [*simp*]: $|m|\ dvd\ k \longleftrightarrow m\ dvd\ k$
  ⟨*proof*⟩

lemma *dvd-abs-iff* [*simp*]: $m\ dvd\ |k| \longleftrightarrow m\ dvd\ k$

⟨*proof*⟩

lemma *dvd-if-abs-eq*:
$|l| = |k| \Longrightarrow l$ *dvd* k
⟨*proof*⟩

end

Simprules for comparisons where common factors can be cancelled.

lemmas *mult-compare-simps*[*no-atp*] =
mult-le-cancel-right mult-le-cancel-left
mult-le-cancel-right1 mult-le-cancel-right2
mult-le-cancel-left1 mult-le-cancel-left2
mult-less-cancel-right mult-less-cancel-left
mult-less-cancel-right1 mult-less-cancel-right2
mult-less-cancel-left1 mult-less-cancel-left2
mult-cancel-right mult-cancel-left
mult-cancel-right1 mult-cancel-right2
mult-cancel-left1 mult-cancel-left2

Reasoning about inequalities with division

context *linordered-semidom*
begin

lemma *less-add-one*: $a < a + 1$
⟨*proof*⟩

lemma *zero-less-two*: $0 < 1 + 1$
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *mult-right-le-one-le*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow x * y \leq x$
⟨*proof*⟩

lemma *mult-left-le-one-le*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow y * x \leq x$
⟨*proof*⟩

end

Absolute Value

context *linordered-idom*
begin

lemma *mult-sgn-abs*:
sgn x ∗ |*x*| = *x*
⟨*proof*⟩

lemma *abs-one* [*simp*]:
|*1*| = *1*
⟨*proof*⟩

end

class *ordered-ring-abs* = *ordered-ring* + *ordered-ab-group-add-abs* +
assumes *abs-eq-mult*:
(*0* ≤ *a* ∨ *a* ≤ *0*) ∧ (*0* ≤ *b* ∨ *b* ≤ *0*) ⟹ |*a* ∗ *b*| = |*a*| ∗ |*b*|

context *linordered-idom*
begin

subclass *ordered-ring-abs* ⟨*proof*⟩

lemma *abs-mult*:
|*a* ∗ *b*| = |*a*| ∗ |*b*|
⟨*proof*⟩

lemma *abs-mult-self*:
|*a*| ∗ |*a*| = *a* ∗ *a*
⟨*proof*⟩

lemma *abs-mult-less*:
|*a*| < *c* ⟹ |*b*| < *d* ⟹ |*a*| ∗ |*b*| < *c* ∗ *d*
⟨*proof*⟩

lemma *less-minus-self-iff*:
a < − *a* ⟷ *a* < *0*
⟨*proof*⟩

lemma *abs-less-iff*:
|*a*| < *b* ⟷ *a* < *b* ∧ − *a* < *b*
⟨*proof*⟩

lemma *abs-mult-pos*:
0 ≤ *x* ⟹ |*y*| ∗ *x* = |*y* ∗ *x*|
⟨*proof*⟩

end

code-modulename *SML*
Rings Arith

code-modulename *OCaml*

Rings Arith

code-modulename *Haskell*
Rings Arith

end

14 Fields: Fields

theory *Fields*
imports *Rings*
begin

class *field* = *comm-ring-1* + *inverse* +
assumes *field-inverse*: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
assumes *field-divide-inverse*: $a\ /\ b = a * inverse\ b$
begin

subclass *division-ring*
⟨*proof*⟩

subclass *idom* ⟨*proof*⟩

There is no slick version using division by zero.

lemma *inverse-add*:
$[|\ a \neq 0;\ \ b \neq 0\ |]$
$\Longrightarrow inverse\ a + inverse\ b = (a + b) * inverse\ a * inverse\ b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*, *no-atp*]:
assumes [*simp*]: $b \neq 0$ **and** [*simp*]: $c \neq 0$ **shows** $(c*a)/(c*b) = a/b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*, *no-atp*]:
$[\![b \neq 0;\ c \neq 0]\!] \Longrightarrow (a * c)\ /\ (b * c) = a\ /\ b$
⟨*proof*⟩

lemma *times-divide-eq-left* [*simp*]: $(b\ /\ c) * a = (b * a)\ /\ c$
⟨*proof*⟩

These are later declared as simp rules.

lemmas *times-divide-eq* [*no-atp*] = *times-divide-eq-right times-divide-eq-left*

lemma *add-frac-eq*:
assumes $y \neq 0$ **and** $z \neq 0$
shows $x\ /\ y + w\ /\ z = (x * z + w * y)\ /\ (y * z)$
⟨*proof*⟩

Special Cancellation Simprules for Division

lemma *nonzero-mult-divide-cancel-right* [*simp*, *no-atp*]:
$b \neq 0 \Longrightarrow a * b \mathbin{/} b = a$
⟨*proof*⟩

lemma *nonzero-mult-divide-cancel-left* [*simp*, *no-atp*]:
$a \neq 0 \Longrightarrow a * b \mathbin{/} a = b$
⟨*proof*⟩

lemma *nonzero-divide-mult-cancel-right* [*simp*, *no-atp*]:
$[\![a \neq 0; b \neq 0]\!] \Longrightarrow b \mathbin{/} (a * b) = 1 \mathbin{/} a$
⟨*proof*⟩

lemma *nonzero-divide-mult-cancel-left* [*simp*, *no-atp*]:
$[\![a \neq 0; b \neq 0]\!] \Longrightarrow a \mathbin{/} (a * b) = 1 \mathbin{/} b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*, *no-atp*]:
$[\![b \neq 0; c \neq 0]\!] \Longrightarrow (c * a) \mathbin{/} (b * c) = a \mathbin{/} b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*, *no-atp*]:
$[\![b \neq 0; c \neq 0]\!] \Longrightarrow (a * c) \mathbin{/} (c * b) = a \mathbin{/} b$
⟨*proof*⟩

lemma *add-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x + y \mathbin{/} z = (z * x + y) \mathbin{/} z$
⟨*proof*⟩

lemma *divide-add-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x \mathbin{/} z + y = (x + z * y) \mathbin{/} z$
⟨*proof*⟩

lemma *diff-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x - y \mathbin{/} z = (z * x - y) \mathbin{/} z$
⟨*proof*⟩

lemma *divide-diff-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x \mathbin{/} z - y = (x - z * y) \mathbin{/} z$
⟨*proof*⟩

lemma *diff-frac-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y - w \mathbin{/} z = (x * z - w * y) \mathbin{/} (y * z)$
⟨*proof*⟩

lemma *frac-eq-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow (x \mathbin{/} y = w \mathbin{/} z) = (x * z = w * y)$
⟨*proof*⟩

end

class *field-inverse-zero* = *field* +
  assumes *field-inverse-zero*: *inverse 0 = 0*
begin

subclass *division-ring-inverse-zero* ⟨*proof*⟩

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
  inverse (*a* ∗ *b*) = *inverse a* ∗ *inverse b*
⟨*proof*⟩

lemma *inverse-divide* [*simp*]:
  inverse (*a* / *b*) = *b* / *a*
  ⟨*proof*⟩

Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemma *mult-divide-mult-cancel-left*:
  c ≠ *0* ⟹ (*c* ∗ *a*) / (*c* ∗ *b*) = *a* / *b*
⟨*proof*⟩

lemma *mult-divide-mult-cancel-right*:
  c ≠ *0* ⟹ (*a* ∗ *c*) / (*b* ∗ *c*) = *a* / *b*
⟨*proof*⟩

lemma *divide-divide-eq-right* [*simp*, *no-atp*]:
  a / (*b* / *c*) = (*a* ∗ *c*) / *b*
  ⟨*proof*⟩

lemma *divide-divide-eq-left* [*simp*, *no-atp*]:
  (*a* / *b*) / *c* = *a* / (*b* ∗ *c*)
  ⟨*proof*⟩

Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if* [*simp*,*no-atp*]:
  shows (*c* ∗ *a*) / (*c* ∗ *b*) = (*if c = 0 then 0 else a* / *b*)
  ⟨*proof*⟩

Division and Unary Minus

lemma *minus-divide-right*:
  − (*a* / *b*) = *a* / − *b*
  ⟨*proof*⟩

lemma *divide-minus-right* [*simp*, *no-atp*]:
$a \mathbin{/} - b = - (a \mathbin{/} b)$
⟨*proof*⟩

lemma *minus-divide-divide*:
$(- a) \mathbin{/} (- b) = a \mathbin{/} b$
⟨*proof*⟩

lemma *eq-divide-eq*:
$a = b \mathbin{/} c \longleftrightarrow (\textit{if } c \neq 0 \textit{ then } a * c = b \textit{ else } a = 0)$
⟨*proof*⟩

lemma *divide-eq-eq*:
$b \mathbin{/} c = a \longleftrightarrow (\textit{if } c \neq 0 \textit{ then } b = a * c \textit{ else } a = 0)$
⟨*proof*⟩

lemma *inverse-eq-1-iff* [*simp*]:
$\textit{inverse } x = 1 \longleftrightarrow x = 1$
⟨*proof*⟩

lemma *divide-eq-0-iff* [*simp*, *no-atp*]:
$a \mathbin{/} b = 0 \longleftrightarrow a = 0 \vee b = 0$
⟨*proof*⟩

lemma *divide-cancel-right* [*simp*, *no-atp*]:
$a \mathbin{/} c = b \mathbin{/} c \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

lemma *divide-cancel-left* [*simp*, *no-atp*]:
$c \mathbin{/} a = c \mathbin{/} b \longleftrightarrow c = 0 \vee a = b$
⟨*proof*⟩

lemma *divide-eq-1-iff* [*simp*, *no-atp*]:
$a \mathbin{/} b = 1 \longleftrightarrow b \neq 0 \wedge a = b$
⟨*proof*⟩

lemma *one-eq-divide-iff* [*simp*, *no-atp*]:
$1 = a \mathbin{/} b \longleftrightarrow b \neq 0 \wedge a = b$
⟨*proof*⟩

lemma *times-divide-times-eq*:
$(x \mathbin{/} y) * (z \mathbin{/} w) = (x * z) \mathbin{/} (y * w)$
⟨*proof*⟩

lemma *add-frac-num*:
$y \neq 0 \Longrightarrow x \mathbin{/} y + z = (x + z * y) \mathbin{/} y$
⟨*proof*⟩

lemma *add-num-frac*:

$y \neq 0 \Longrightarrow z + x \mathbin{/} y = (x + z * y) \mathbin{/} y$
⟨*proof*⟩

end

Ordered Fields

class *linordered-field* = *field* + *linordered-idom*
begin

lemma *positive-imp-inverse-positive*:
assumes *a-gt-0*: $0 < a$
shows $0 <$ *inverse* a
⟨*proof*⟩

lemma *negative-imp-inverse-negative*:
$a < 0 \Longrightarrow$ *inverse* $a < 0$
⟨*proof*⟩

lemma *inverse-le-imp-le*:
assumes *invle*: *inverse* $a \leq$ *inverse* b **and** *apos*: $0 < a$
shows $b \leq a$
⟨*proof*⟩

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 <$ *inverse* a **and** *nz*: $a \neq 0$
shows $0 < a$
⟨*proof*⟩

lemma *inverse-negative-imp-negative*:
assumes *inv-less-0*: *inverse* $a < 0$ **and** *nz*: $a \neq 0$
shows $a < 0$
⟨*proof*⟩

lemma *linordered-field-no-lb*:
$\forall x.\ \exists y.\ y < x$
⟨*proof*⟩

lemma *linordered-field-no-ub*:
$\forall\ x.\ \exists y.\ y > x$
⟨*proof*⟩

lemma *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows *inverse* $b <$ *inverse* a
⟨*proof*⟩

lemma *inverse-less-imp-less*:
inverse $a <$ *inverse* $b \Longrightarrow 0 < a \Longrightarrow b < a$
⟨*proof*⟩

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*,*no-atp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a < inverse\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *le-imp-inverse-le*:
$a \leq b \Longrightarrow 0 < a \Longrightarrow inverse\ b \leq inverse\ a$
⟨*proof*⟩

lemma *inverse-le-iff-le* [*simp*,*no-atp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
$inverse\ a \leq inverse\ b \Longrightarrow b < 0 \Longrightarrow b \leq a$
⟨*proof*⟩

lemma *less-imp-inverse-less-neg*:
$a < b \Longrightarrow b < 0 \Longrightarrow inverse\ b < inverse\ a$
⟨*proof*⟩

lemma *inverse-less-imp-less-neg*:
$inverse\ a < inverse\ b \Longrightarrow b < 0 \Longrightarrow b < a$
⟨*proof*⟩

lemma *inverse-less-iff-less-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow inverse\ a < inverse\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *le-imp-inverse-le-neg*:
$a \leq b \Longrightarrow b < 0 \Longrightarrow inverse\ b \leq inverse\ a$
⟨*proof*⟩

lemma *inverse-le-iff-le-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

lemma *one-less-inverse*:
$0 < a \Longrightarrow a < 1 \Longrightarrow 1 < inverse\ a$
⟨*proof*⟩

lemma *one-le-inverse*:
$0 < a \Longrightarrow a \leq 1 \Longrightarrow 1 \leq inverse\ a$
⟨*proof*⟩

lemma *pos-le-divide-eq* [*field-simps*]: $0 < c \Longrightarrow (a \leq b/c) = (a*c \leq b)$
⟨*proof*⟩

lemma *neg-le-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow (a \leq b/c) = (b \leq a*c)$
⟨*proof*⟩

lemma *pos-less-divide-eq* [*field-simps*]:
$0 < c \Longrightarrow (a < b/c) = (a*c < b)$
⟨*proof*⟩

lemma *neg-less-divide-eq* [*field-simps*]:
$c < 0 \Longrightarrow (a < b/c) = (b < a*c)$
⟨*proof*⟩

lemma *pos-divide-less-eq* [*field-simps*]:
$0 < c \Longrightarrow (b/c < a) = (b < a*c)$
⟨*proof*⟩

lemma *neg-divide-less-eq* [*field-simps*]:
$c < 0 \Longrightarrow (b/c < a) = (a*c < b)$
⟨*proof*⟩

lemma *pos-divide-le-eq* [*field-simps*]: $0 < c \Longrightarrow (b/c \leq a) = (b \leq a*c)$
⟨*proof*⟩

lemma *neg-divide-le-eq* [*field-simps*]: $c < 0 \Longrightarrow (b/c \leq a) = (a*c \leq b)$
⟨*proof*⟩

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

lemmas *sign-simps* [*no-atp*] = *algebra-simps*
zero-less-mult-iff mult-less-0-iff

lemmas (**in** −) *sign-simps* [*no-atp*] = *algebra-simps*
zero-less-mult-iff mult-less-0-iff

lemma *divide-pos-pos*:
$0 < x \Longrightarrow 0 < y \Longrightarrow 0 < x \mathbin{/} y$
⟨*proof*⟩

lemma *divide-nonneg-pos*:
$0 <= x \Longrightarrow 0 < y \Longrightarrow 0 <= x \mathbin{/} y$
⟨*proof*⟩

lemma *divide-neg-pos*:
$x < 0 \Longrightarrow 0 < y \Longrightarrow x \mathbin{/} y < 0$
⟨*proof*⟩

lemma *divide-nonpos-pos*:
$x <= 0 ==> 0 < y ==> x / y <= 0$
⟨*proof*⟩

lemma *divide-pos-neg*:
$0 < x ==> y < 0 ==> x / y < 0$
⟨*proof*⟩

lemma *divide-nonneg-neg*:
$0 <= x ==> y < 0 ==> x / y <= 0$
⟨*proof*⟩

lemma *divide-neg-neg*:
$x < 0 ==> y < 0 ==> 0 < x / y$
⟨*proof*⟩

lemma *divide-nonpos-neg*:
$x <= 0 ==> y < 0 ==> 0 <= x / y$
⟨*proof*⟩

lemma *divide-strict-right-mono*:
$[|a < b; 0 < c|] ==> a / c < b / c$
⟨*proof*⟩

lemma *divide-strict-right-mono-neg*:
$[|b < a; c < 0|] ==> a / c < b / c$
⟨*proof*⟩

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
$[|b < a; 0 < c; 0 < a*b|] ==> c / a < c / b$
⟨*proof*⟩

lemma *divide-left-mono*:
$[|b \leq a; 0 \leq c; 0 < a*b|] ==> c / a \leq c / b$
⟨*proof*⟩

lemma *divide-strict-left-mono-neg*:
$[|a < b; c < 0; 0 < a*b|] ==> c / a < c / b$
⟨*proof*⟩

lemma *mult-imp-div-pos-le*: $0 < y ==> x <= z * y ==>$
$x / y <= z$
⟨*proof*⟩

lemma *mult-imp-le-div-pos*: $0 < y ==> z * y <= x ==>$
$z <= x / y$
⟨*proof*⟩

lemma *mult-imp-div-pos-less*: *0 < y ==> x < z * y ==>*
x / y < z
⟨*proof*⟩

lemma *mult-imp-less-div-pos*: *0 < y ==> z * y < x ==>*
z < x / y
⟨*proof*⟩

lemma *frac-le*: *0 <= x ==>*
x <= y ==> 0 < w ==> w <= z ==> x / z <= y / w
⟨*proof*⟩

lemma *frac-less*: *0 <= x ==>*
x < y ==> 0 < w ==> w <= z ==> x / z < y / w
⟨*proof*⟩

lemma *frac-less2*: *0 < x ==>*
x <= y ==> 0 < w ==> w < z ==> x / z < y / w
⟨*proof*⟩

It's not obvious whether these should be simprules or not. Their effect is to gather terms into one big fraction, like a*b*c / x*y*z. The rationale for that is unclear, but many proofs seem to need them.

lemma *less-half-sum*: *a < b ==> a < (a+b) / (1+1)*
⟨*proof*⟩

lemma *gt-half-sum*: *a < b ==> (a+b)/(1+1) < b*
⟨*proof*⟩

subclass *dense-linorder*
⟨*proof*⟩

lemma *nonzero-abs-inverse*:
a ≠ 0 ==> |inverse a| = inverse |a|
⟨*proof*⟩

lemma *nonzero-abs-divide*:
b ≠ 0 ==> |a / b| = |a| / |b|
⟨*proof*⟩

lemma *field-le-epsilon*:
assumes *e*: $\bigwedge e.\ 0 < e \Longrightarrow x \leq y + e$
shows $x \leq y$
⟨*proof*⟩

end

class *linordered-field-inverse-zero = linordered-field + field-inverse-zero*

begin

lemma *le-divide-eq*:
$(a \leq b/c) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ a*c \leq b$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ b \leq a*c$
$\mathit{else}\ \ a \leq 0)$
⟨*proof*⟩

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 < \mathit{inverse}\ a) = (0 < a)$
⟨*proof*⟩

lemma *inverse-negative-iff-negative* [*simp*]:
$(\mathit{inverse}\ a < 0) = (a < 0)$
⟨*proof*⟩

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
$0 \leq \mathit{inverse}\ a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
$\mathit{inverse}\ a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *one-less-inverse-iff*:
$1 < \mathit{inverse}\ x \longleftrightarrow 0 < x \wedge x < 1$
⟨*proof*⟩

lemma *one-le-inverse-iff*:
$1 \leq \mathit{inverse}\ x \longleftrightarrow 0 < x \wedge x \leq 1$
⟨*proof*⟩

lemma *inverse-less-1-iff*:
$\mathit{inverse}\ x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$
⟨*proof*⟩

lemma *inverse-le-1-iff*:
$\mathit{inverse}\ x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$
⟨*proof*⟩

lemma *divide-le-eq*:
$(b/c \leq a) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ b \leq a*c$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ a*c \leq b$
$\mathit{else}\ 0 \leq a)$
⟨*proof*⟩

lemma *less-divide-eq*:

$(a < b/c) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ a*c < b$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ b < a*c$
$\mathit{else}\ \ a < 0)$
⟨*proof*⟩

lemma *divide-less-eq*:
$(b/c < a) =$
$(\mathit{if}\ 0 < c\ \mathit{then}\ b < a*c$
$\mathit{else\ if}\ c < 0\ \mathit{then}\ a*c < b$
$\mathit{else}\ 0 < a)$
⟨*proof*⟩

Division and Signs

lemma *zero-less-divide-iff*:
$(0 < a/b) = (0 < a\ \&\ 0 < b \mid a < 0\ \&\ b < 0)$
⟨*proof*⟩

lemma *divide-less-0-iff*:
$(a/b < 0) =$
$(0 < a\ \&\ b < 0 \mid a < 0\ \&\ 0 < b)$
⟨*proof*⟩

lemma *zero-le-divide-iff*:
$(0 \leq a/b) =$
$(0 \leq a\ \&\ 0 \leq b \mid a \leq 0\ \&\ b \leq 0)$
⟨*proof*⟩

lemma *divide-le-0-iff*:
$(a/b \leq 0) =$
$(0 \leq a\ \&\ b \leq 0 \mid a \leq 0\ \&\ 0 \leq b)$
⟨*proof*⟩

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp*,*no-atp*]:
$(0 = 1/a) = (a = 0)$
⟨*proof*⟩

lemma *one-divide-eq-0-iff* [*simp*,*no-atp*]:
$(1/a = 0) = (a = 0)$
⟨*proof*⟩

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma *zero-le-divide-1-iff* [*simp*, *no-atp*]:
$0 \leq 1\ /\ a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-less-divide-1-iff* [*simp*, *no-atp*]:
$0 < 1 \,/\, a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *divide-le-0-1-iff* [*simp*, *no-atp*]:
$1 \,/\, a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *divide-less-0-1-iff* [*simp*, *no-atp*]:
$1 \,/\, a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *divide-right-mono*:
$[|a \leq b; 0 \leq c|] ==> a/c \leq b/c$
⟨*proof*⟩

lemma *divide-right-mono-neg*: $a <= b$
$==> c <= 0 ==> b \,/\, c <= a \,/\, c$
⟨*proof*⟩

lemma *divide-left-mono-neg*: $a <= b$
$==> c <= 0 ==> 0 < a * b ==> c \,/\, a <= c \,/\, b$
⟨*proof*⟩

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*no-atp*]:
$(1 \leq b \,/\, a) = ((0 < a \,\&\, a \leq b) \mid (a < 0 \,\&\, b \leq a))$
⟨*proof*⟩

lemma *divide-le-eq-1* [*no-atp*]:
$(b \,/\, a \leq 1) = ((0 < a \,\&\, b \leq a) \mid (a < 0 \,\&\, a \leq b) \mid a=0)$
⟨*proof*⟩

lemma *less-divide-eq-1* [*no-atp*]:
$(1 < b \,/\, a) = ((0 < a \,\&\, a < b) \mid (a < 0 \,\&\, b < a))$
⟨*proof*⟩

lemma *divide-less-eq-1* [*no-atp*]:
$(b \,/\, a < 1) = ((0 < a \,\&\, b < a) \mid (a < 0 \,\&\, a < b) \mid a=0)$
⟨*proof*⟩

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
⟨*proof*⟩

lemma *le-divide-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
⟨*proof*⟩

lemma *less-divide-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (1 < b/a) = (a < b)$
⟨*proof*⟩

lemma *less-divide-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (1 < b/a) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (b/a < 1) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b/a < 1 <-> a < b$
⟨*proof*⟩

lemma *eq-divide-eq-1* [*simp*,*no-atp*]:
$(1 = b/a) = ((a \neq 0 \ \&\ a = b))$
⟨*proof*⟩

lemma *divide-eq-eq-1* [*simp*,*no-atp*]:
$(b/a = 1) = ((a \neq 0 \ \&\ a = b))$
⟨*proof*⟩

lemma *abs-inverse* [*simp*]:
$|inverse\ a| =$
$inverse\ |a|$
⟨*proof*⟩

lemma *abs-divide* [*simp*]:
$|a\ /\ b| = |a|\ /\ |b|$
⟨*proof*⟩

lemma *abs-div-pos*: $0 < y ==>$
$|x|\ /\ y = |x\ /\ y|$
⟨*proof*⟩

lemma *field-le-mult-one-interval*:
assumes $*$: $\bigwedge z.\ [\![\ 0 < z\ ;\ z < 1\ ]\!] \Longrightarrow z * x \leq y$
shows $x \leq y$

⟨*proof*⟩

end

code-modulename *SML*
Fields Arith

code-modulename *OCaml*
Fields Arith

code-modulename *Haskell*
Fields Arith

end

15 Nat: Natural numbers

theory *Nat*
imports *Inductive Typedef Fun Fields*
uses
~~/*src*/*Tools*/*rat.ML*
~~/*src*/*Provers*/*Arith*/*cancel-sums.ML*
Tools/*arith-data.ML*
(*Tools*/*nat-arith.ML*)
~~/*src*/*Provers*/*Arith*/*fast-lin-arith.ML*
(*Tools*/*lin-arith.ML*)
begin

15.1 Type *ind*

typedecl *ind*

axiomatization
Zero-Rep :: *ind* **and**
Suc-Rep :: *ind* => *ind*
where
— the axiom of infinity in 2 parts
Suc-Rep-inject: *Suc-Rep x* = *Suc-Rep y* ==> *x* = *y* **and**
Suc-Rep-not-Zero-Rep: *Suc-Rep x* ≠ *Zero-Rep*

15.2 Type nat

Type definition

inductive *Nat* :: *ind* ⇒ *bool*
where
Zero-RepI: *Nat Zero-Rep*
| *Suc-RepI*: *Nat i* ⟹ *Nat* (*Suc-Rep i*)

global

typedef (**open** *Nat*)
nat = *Nat*
⟨*proof*⟩

definition *Suc* :: *nat* => *nat* **where**
Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))

local

instantiation *nat* :: *zero*
begin

definition *Zero-nat-def* [*code del*]:
0 = *Abs-Nat Zero-Rep*

instance ⟨*proof*⟩

end

lemma *Suc-not-Zero*: *Suc m* ≠ *0*
⟨*proof*⟩

lemma *Zero-not-Suc*: *0* ≠ *Suc m*
⟨*proof*⟩

lemma *Suc-Rep-inject'*: *Suc-Rep x* = *Suc-Rep y* ⟷ *x* = *y*
⟨*proof*⟩

rep-datatype *0* :: *nat Suc*
⟨*proof*⟩

lemma *nat-induct* [*case-names 0 Suc*, *induct type*: *nat*]:
— for backward compatibility – names of variables differ
fixes *n*
assumes *P 0*
and ⋀*n*. *P n* ⟹ *P* (*Suc n*)
shows *P n*
⟨*proof*⟩

declare *nat.exhaust* [*case-names 0 Suc*, *cases type*: *nat*]

lemmas *nat-rec-0* = *nat.recs*(*1*)
and *nat-rec-Suc* = *nat.recs*(*2*)

lemmas *nat-case-0* = *nat.cases*(*1*)
and *nat-case-Suc* = *nat.cases*(*2*)

Injectiveness and distinctness lemmas

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
⟨*proof*⟩

lemma *Suc-neq-Zero*: *Suc m = 0 ⟹ R*
⟨*proof*⟩

lemma *Zero-neq-Suc*: *0 = Suc m ⟹ R*
⟨*proof*⟩

lemma *Suc-inject*: *Suc x = Suc y ⟹ x = y*
⟨*proof*⟩

lemma *n-not-Suc-n*: *n ≠ Suc n*
⟨*proof*⟩

lemma *Suc-n-not-n*: *Suc n ≠ n*
⟨*proof*⟩

A special form of induction for reasoning about $m < n$ and $m - n$

lemma *diff-induct*: (!!*x*. *P x 0*) ==> (!!*y*. *P 0* (*Suc y*)) ==>
(!!*x y*. *P x y* ==> *P* (*Suc x*) (*Suc y*)) ==> *P m n*
⟨*proof*⟩

15.3 Arithmetic operators

instantiation *nat* :: {*minus*, *comm-monoid-add*}
begin

primrec *plus-nat*
where
add-0: *0 + n* = (*n*::*nat*)
| *add-Suc*: *Suc m + n* = *Suc* (*m + n*)

lemma *add-0-right* [*simp*]: *m + 0* = (*m*::*nat*)
⟨*proof*⟩

lemma *add-Suc-right* [*simp*]: *m + Suc n* = *Suc* (*m + n*)
⟨*proof*⟩

declare *add-0* [*code*]

lemma *add-Suc-shift* [*code*]: *Suc m + n* = *m + Suc n*
⟨*proof*⟩

primrec *minus-nat*
where
diff-0: *m − 0* = (*m*::*nat*)
| *diff-Suc*: *m − Suc n* = (*case m − n of 0* => *0* | *Suc k* => *k*)

declare *diff-Suc* [*simp del*]
declare *diff-0* [*code*]

lemma *diff-0-eq-0* [*simp*, *code*]: $0 - n = (0::nat)$
⟨*proof*⟩

lemma *diff-Suc-Suc* [*simp*, *code*]: $Suc\ m - Suc\ n = m - n$
⟨*proof*⟩

instance ⟨*proof*⟩

end

hide-fact (**open**) *add-0 add-0-right diff-0*

instantiation *nat* :: *comm-semiring-1-cancel*
begin

definition
One-nat-def [*simp*]: $1 = Suc\ 0$

primrec *times-nat*
where
mult-0: $0 * n = (0::nat)$
| *mult-Suc*: $Suc\ m * n = n + (m * n)$

lemma *mult-0-right* [*simp*]: $(m::nat) * 0 = 0$
⟨*proof*⟩

lemma *mult-Suc-right* [*simp*]: $m * Suc\ n = m + (m * n)$
⟨*proof*⟩

lemma *add-mult-distrib*: $(m + n) * k = (m * k) + ((n * k)::nat)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

15.3.1 Addition

lemma *nat-add-assoc*: $(m + n) + k = m + ((n + k)::nat)$
⟨*proof*⟩

lemma *nat-add-commute*: $m + n = n + (m::nat)$
⟨*proof*⟩

lemma *nat-add-left-commute*: $x + (y + z) = y + ((x + z)::nat)$

⟨*proof*⟩

lemma *nat-add-left-cancel* [*simp*]: $(k + m = k + n) = (m = (n::nat))$
⟨*proof*⟩

lemma *nat-add-right-cancel* [*simp*]: $(m + k = n + k) = (m=(n::nat))$
⟨*proof*⟩

Reasoning about $m + 0 = 0$, etc.

lemma *add-is-0* [*iff*]:
fixes *m n* :: *nat*
shows $(m + n = 0) = (m = 0 \ \& \ n = 0)$
⟨*proof*⟩

lemma *add-is-1*:
$(m+n= Suc\ 0) = (m= Suc\ 0 \ \& \ n=0 \mid m=0 \ \& \ n= Suc\ 0)$
⟨*proof*⟩

lemma *one-is-add*:
$(Suc\ 0 = m + n) = (m = Suc\ 0 \ \& \ n = 0 \mid m = 0 \ \& \ n = Suc\ 0)$
⟨*proof*⟩

lemma *add-eq-self-zero*:
fixes *m n* :: *nat*
shows $m + n = m \Longrightarrow n = 0$
⟨*proof*⟩

lemma *inj-on-add-nat*[*simp*]: *inj-on* $(\%n::nat.\ n+k)\ N$
⟨*proof*⟩

15.3.2 Difference

lemma *diff-self-eq-0* [*simp*]: $(m::nat) - m = 0$
⟨*proof*⟩

lemma *diff-diff-left*: $(i::nat) - j - k = i - (j + k)$
⟨*proof*⟩

lemma *Suc-diff-diff* [*simp*]: $(Suc\ m - n) - Suc\ k = m - n - k$
⟨*proof*⟩

lemma *diff-commute*: $(i::nat) - j - k = i - k - j$
⟨*proof*⟩

lemma *diff-add-inverse*: $(n + m) - n = (m::nat)$
⟨*proof*⟩

lemma *diff-add-inverse2*: $(m + n) - n = (m::nat)$
⟨*proof*⟩

lemma *diff-cancel*: $(k + m) - (k + n) = m - (n::nat)$
⟨*proof*⟩

lemma *diff-cancel2*: $(m + k) - (n + k) = m - (n::nat)$
⟨*proof*⟩

lemma *diff-add-0*: $n - (n + m) = (0::nat)$
⟨*proof*⟩

lemma *diff-Suc-1* [*simp*]: $Suc\ n - 1 = n$
⟨*proof*⟩

Difference distributes over multiplication

lemma *diff-mult-distrib*: $((m::nat) - n) * k = (m * k) - (n * k)$
⟨*proof*⟩

lemma *diff-mult-distrib2*: $k * ((m::nat) - n) = (k * m) - (k * n)$
⟨*proof*⟩

15.3.3 Multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
⟨*proof*⟩

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
⟨*proof*⟩

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
⟨*proof*⟩

lemma *mult-is-0* [*simp*]: $((m::nat) * n = 0) = (m=0 \mid n=0)$
⟨*proof*⟩

lemmas *nat-distrib* =
add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

lemma *mult-eq-1-iff* [*simp*]: $(m * n = Suc\ 0) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
⟨*proof*⟩

lemma *one-eq-mult-iff* [*simp*,*no-atp*]: $(Suc\ 0 = m * n) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
⟨*proof*⟩

lemma *nat-mult-eq-1-iff* [*simp*]: $m * n = (1::nat) \longleftrightarrow m = 1 \wedge n = 1$
⟨*proof*⟩

lemma *nat-1-eq-mult-iff* [*simp*]: $(1::nat) = m * n \longleftrightarrow m = 1 \wedge n = 1$
⟨*proof*⟩

lemma *mult-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
⟨*proof*⟩

lemma *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
⟨*proof*⟩

lemma *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
⟨*proof*⟩

15.4 Orders on *nat*

15.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat* **where**
$(0::nat) \leq n \longleftrightarrow True$
| $Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*]
lemma [*code*]: $(0::nat) \leq n \longleftrightarrow True$ ⟨*proof*⟩
lemma *le0* [*iff*]: $0 \leq (n::nat)$ ⟨*proof*⟩

definition *less-nat* **where**
less-eq-Suc-le: $n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *Suc-le-eq* [*code*]: $Suc\ m \leq n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *le-0-eq* [*iff*]: $(n::nat) \leq 0 \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *not-less0* [*iff*]: $\neg\ n < (0::nat)$
⟨*proof*⟩

lemma *less-nat-zero-code* [*code*]: $n < (0::nat) \longleftrightarrow False$
⟨*proof*⟩

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$

⟨*proof*⟩

lemma *Suc-leD*: *Suc m* ≤ *n* ⟹ *m* ≤ *n*
⟨*proof*⟩

lemma *less-SucI*: *m* < *n* ⟹ *m* < *Suc n*
⟨*proof*⟩

lemma *Suc-lessD*: *Suc m* < *n* ⟹ *m* < *n*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *nat* :: *bot*
begin

definition *bot-nat* :: *nat* **where**
bot-nat = *0*

instance ⟨*proof*⟩

end

15.4.2 Introduction properties

lemma *lessI* [*iff*]: *n* < *Suc n*
⟨*proof*⟩

lemma *zero-less-Suc* [*iff*]: *0* < *Suc n*
⟨*proof*⟩

15.4.3 Elimination properties

lemma *less-not-refl*: ~ *n* < (*n*::*nat*)
⟨*proof*⟩

lemma *less-not-refl2*: *n* < *m* ==> *m* ≠ (*n*::*nat*)
⟨*proof*⟩

lemma *less-not-refl3*: (*s*::*nat*) < *t* ==> *s* ≠ *t*
⟨*proof*⟩

lemma *less-irrefl-nat*: (*n*::*nat*) < *n* ==> *R*
⟨*proof*⟩

lemma *less-zeroE*: (*n*::*nat*) < *0* ==> *R*
⟨*proof*⟩

lemma *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
⟨*proof*⟩

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
⟨*proof*⟩

lemma *less-one* [*iff*, *no-atp*]: $(n < (1::nat)) = (n = 0)$
⟨*proof*⟩

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
⟨*proof*⟩

"Less than" is antisymmetric, sort of

lemma *less-antisym*: $\llbracket \neg\ n < m;\ n < Suc\ m \rrbracket \Longrightarrow m = n$
⟨*proof*⟩

lemma *nat-neq-iff*: $((m::nat) \neq n) = (m < n \mid n < m)$
⟨*proof*⟩

lemma *nat-less-cases*: **assumes** *major*: $(m::nat) < n \Longrightarrow P\ n\ m$
and *eqCase*: $m = n \Longrightarrow P\ n\ m$ **and** *lessCase*: $n<m \Longrightarrow P\ n\ m$
shows $P\ n\ m$
⟨*proof*⟩

15.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \Longrightarrow Suc\ m \neq n \Longrightarrow Suc\ m < n$
⟨*proof*⟩

lemma *lessE*:
assumes *major*: $i < k$
and *p1*: $k = Suc\ i \Longrightarrow P$ **and** *p2*: $!!j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *less-SucE*: **assumes** *major*: $m < Suc\ n$
and *less*: $m < n \Longrightarrow P$ **and** *eq*: $m = n \Longrightarrow P$ **shows** P
⟨*proof*⟩

lemma *Suc-lessE*: **assumes** *major*: $Suc\ i < k$
and *minor*: $!!j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$ **shows** P
⟨*proof*⟩

lemma *Suc-less-SucD*: $Suc\ m < Suc\ n \Longrightarrow m < n$
⟨*proof*⟩

lemma *less-trans-Suc*:
assumes *le*: $i < j$ **shows** $j < k \Longrightarrow Suc\ i < k$

⟨*proof*⟩

Can be used with *less-Suc-eq* to get $n = m \vee n < m$

lemma *not-less-eq*: $\neg\ m < n \longleftrightarrow n < Suc\ m$
⟨*proof*⟩

lemma *not-less-eq-eq*: $\neg\ m \leq n \longleftrightarrow Suc\ n \leq m$
⟨*proof*⟩

Properties of "less than or equal"

lemma *le-imp-less-Suc*: $m \leq n \Longrightarrow m < Suc\ n$
⟨*proof*⟩

lemma *Suc-n-not-le-n*: $\sim\ Suc\ n \leq n$
⟨*proof*⟩

lemma *le-Suc-eq*: $(m \leq Suc\ n) = (m \leq n \mid m = Suc\ n)$
⟨*proof*⟩

lemma *le-SucE*: $m \leq Suc\ n \Longrightarrow (m \leq n \Longrightarrow R) \Longrightarrow (m = Suc\ n \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *Suc-leI*: $m < n \Longrightarrow Suc(m) \leq n$
⟨*proof*⟩

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $Suc\ m \leq n \Longrightarrow m < n$
⟨*proof*⟩

lemma *less-imp-le-nat*: $m < n \Longrightarrow m \leq (n::nat)$
⟨*proof*⟩

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \mid m = n \Longrightarrow m \leq (n::nat)$
⟨*proof*⟩

lemma *le-eq-less-or-eq*: $(m \leq (n::nat)) = (m < n \mid m=n)$
⟨*proof*⟩

Useful with *blast*.

lemma *eq-imp-le*: $(m::nat) = n \Longrightarrow m \leq n$
⟨*proof*⟩

lemma *le-refl*: $n \leq (n::nat)$

⟨*proof*⟩

lemma *le-trans*: [| $i \leq j$; $j \leq k$ |] ==> $i \leq (k::nat)$
⟨*proof*⟩

lemma *le-antisym*: [| $m \leq n$; $n \leq m$ |] ==> $m = (n::nat)$
⟨*proof*⟩

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \;\&\; m \neq n)$
⟨*proof*⟩

lemma *le-neq-implies-less*: $(m::nat) \leq n$ ==> $m \neq n$ ==> $m < n$
⟨*proof*⟩

lemma *nat-le-linear*: $(m::nat) \leq n \mid n \leq m$
⟨*proof*⟩

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = nat$]

lemma *le-less-Suc-eq*: $m \leq n$ ==> $(n < Suc\ m) = (n = m)$
⟨*proof*⟩

lemma *not-less-less-Suc-eq*: ~ $n < m$ ==> $(n < Suc\ m) = (n = m)$
⟨*proof*⟩

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

These two rules ease the use of primitive recursion. NOTE USE OF ==

lemma *def-nat-rec-0*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ 0 = c$
⟨*proof*⟩

lemma *def-nat-rec-Suc*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ (Suc\ n) = h\ n\ (f\ n)$
⟨*proof*⟩

lemma *not0-implies-Suc*: $n \neq 0$ ==> $\exists m.\ n = Suc\ m$
⟨*proof*⟩

lemma *gr0-implies-Suc*: $n > 0$ ==> $\exists m.\ n = Suc\ m$
⟨*proof*⟩

lemma *gr-implies-not0*: **fixes** $n :: nat$ **shows** $m<n$ ==> $n \neq 0$
⟨*proof*⟩

lemma *neq0-conv*[*iff*]: **fixes** $n :: nat$ **shows** $(n \neq 0) = (0 < n)$
⟨*proof*⟩

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0$ ==> $False)$ ==> $0 < n$
⟨*proof*⟩

lemma *gr0-conv-Suc*: $(0 < n) = (\exists m.\ n = Suc\ m)$
⟨*proof*⟩

lemma *not-gr0* [*iff*,*no-atp*]: $!!n{::}nat.\ (\sim\ (0 < n)) = (n = 0)$
⟨*proof*⟩

lemma *Suc-le-D*: $(Suc\ n \leq m') ==> (?\ m.\ m' = Suc\ m)$
⟨*proof*⟩

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $(m < Suc\ n) = (m = 0 \mid (\exists j.\ m = Suc\ j\ \&\ j < n))$
⟨*proof*⟩

15.4.5 *min* and *max*

lemma *mono-Suc*: *mono Suc*
⟨*proof*⟩

lemma *min-0L* [*simp*]: $min\ 0\ n = (0{::}nat)$
⟨*proof*⟩

lemma *min-0R* [*simp*]: $min\ n\ 0 = (0{::}nat)$
⟨*proof*⟩

lemma *min-Suc-Suc* [*simp*]: $min\ (Suc\ m)\ (Suc\ n) = Suc\ (min\ m\ n)$
⟨*proof*⟩

lemma *min-Suc1*:
$min\ (Suc\ n)\ m = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc(min\ n\ m'))$
⟨*proof*⟩

lemma *min-Suc2*:
$min\ m\ (Suc\ n) = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc(min\ m'\ n))$
⟨*proof*⟩

lemma *max-0L* [*simp*]: $max\ 0\ n = (n{::}nat)$
⟨*proof*⟩

lemma *max-0R* [*simp*]: $max\ n\ 0 = (n{::}nat)$
⟨*proof*⟩

lemma *max-Suc-Suc* [*simp*]: $max\ (Suc\ m)\ (Suc\ n) = Suc(max\ m\ n)$
⟨*proof*⟩

lemma *max-Suc1*:
$max\ (Suc\ n)\ m = (case\ m\ of\ 0 => Suc\ n \mid Suc\ m' => Suc(max\ n\ m'))$
⟨*proof*⟩

lemma *max-Suc2*:
max m (Suc n) = (case m of 0 => Suc n | Suc m' => Suc(max m' n))
⟨*proof*⟩

15.4.6 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: *n>0 ==> Suc (n - Suc 0) = n*
⟨*proof*⟩

lemma *Suc-diff-1* [*simp*]: *0 < n ==> Suc (n - 1) = n*
⟨*proof*⟩

lemma *nat-add-left-cancel-le* [*simp*]: *(k + m ≤ k + n) = (m≤(n::nat))*
⟨*proof*⟩

lemma *nat-add-left-cancel-less* [*simp*]: *(k + m < k + n) = (m<(n::nat))*
⟨*proof*⟩

lemma *add-gr-0* [*iff*]: *!!m::nat. (m + n > 0) = (m>0 | n>0)*
⟨*proof*⟩

strict, in 1st argument

lemma *add-less-mono1*: *i < j ==> i + k < j + (k::nat)*
⟨*proof*⟩

strict, in both arguments

lemma *add-less-mono*: *[|i < j; k < l|] ==> i + k < j + (l::nat)*
⟨*proof*⟩

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

lemma *less-imp-Suc-add*: *m < n ==> (∃k. n = Suc (m + k))*
⟨*proof*⟩

strict, in 1st argument; proof is by induction on *k > 0*

lemma *mult-less-mono2*: *(i::nat) < j ==> 0<k ==> k * i < k * j*
⟨*proof*⟩

The naturals form an ordered *comm-semiring-1-cancel*

instance *nat* :: *linordered-semidom*
⟨*proof*⟩

instance *nat* :: *no-zero-divisors*
⟨*proof*⟩

lemma *nat-mult-1*: *(1::nat) * n = n*
⟨*proof*⟩

lemma *nat-mult-1-right*: *n * (1::nat) = n*
⟨*proof*⟩

15.4.7 Additional theorems about *op* ≤

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder* ⟨*proof*⟩

lemma *Least-Suc*:
[| *P n*; ~ *P 0* |] ==> (*LEAST n*. *P n*) = *Suc* (*LEAST m*. *P*(*Suc m*))
⟨*proof*⟩

lemma *Least-Suc2*:
[|*P n*; *Q m*; ~*P 0*; !*k*. *P* (*Suc k*) = *Q k*|] ==> *Least P* = *Suc* (*Least Q*)
⟨*proof*⟩

lemma *ex-least-nat-le*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃ *k*≤*n*. (∀ *i*<*k*. ¬*P i*) & *P*(*k*)
⟨*proof*⟩

lemma *ex-least-nat-less*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃ *k*<*n*. (∀ *i*≤*k*. ¬*P i*) & *P*(*k*+*1*)
⟨*proof*⟩

lemma *nat-less-induct*:
assumes !!*n*. ∀ *m*::*nat*. *m* < *n* −−> *P m* ==> *P n* **shows** *P n*
⟨*proof*⟩

lemma *measure-induct-rule* [*case-names less*]:
fixes *f* :: *'a* ⇒ *nat*
assumes *step*: ⋀*x*. (⋀*y*. *f y* < *f x* ⟹ *P y*) ⟹ *P x*
shows *P a*
⟨*proof*⟩

old style induction rules:

lemma *measure-induct*:
fixes *f* :: *'a* ⇒ *nat*
shows (⋀*x*. ∀ *y*. *f y* < *f x* ⟶ *P y* ⟹ *P x*) ⟹ *P a*
⟨*proof*⟩

lemma *full-nat-induct*:
assumes *step*: (!!*n*. (*ALL m*. *Suc m* <= *n* −−> *P m*) ==> *P n*)
shows *P n*
⟨*proof*⟩

An induction rule for estabilishing binary relations

lemma *less-Suc-induct*:
assumes *less*: *i* < *j*
and *step*: !!*i*. *P i* (*Suc i*)
and *trans*: !!*i j k*. *i* < *j* ==> *j* < *k* ==> *P i j* ==> *P j k* ==> *P i k*
shows *P i j*
⟨*proof*⟩

The method of infinite descent, frequently used in number theory. Provided

by Roelof Oosterhuis. $P(n)$ is true for all $n \in \mathbb{N}$ if

- case "0": given $n = 0$ prove $P(n)$,
- case "smaller": given $n > 0$ and $\neg P(n)$ prove there exists a smaller integer m such that $\neg P(m)$.

A compact version without explicit base case:

lemma *infinite-descent*:
$[\![\; !!n{::}nat.\ \neg\ P\ n \Longrightarrow\ \exists\, m{<}n.\ \neg\ \ P\ m \;]\!] \Longrightarrow\ \ P\ n$
⟨*proof*⟩

lemma *infinite-descent0*[*case-names 0 smaller*]:
$[\![\; P\ 0;\ !!n.\ n{>}0 \Longrightarrow \neg\ P\ n \Longrightarrow (\exists\, m{::}nat.\ m < n \wedge \neg P\ m) \;]\!] \Longrightarrow P\ n$
⟨*proof*⟩

Infinite descent using a mapping to $\mathbb{N}$: $P(x)$ is true for all $x \in D$ if there exists a $V : D \to \mathbb{N}$ and

- case "0": given $V(x) = 0$ prove $P(x)$,
- case "smaller": given $V(x) > 0$ and $\neg P(x)$ prove there exists a $y \in D$ such that $V(y) < V(x)$ and $\neg P(y)$.

NB: the proof also shows how to use the previous lemma.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
assumes *A0*: $!!x.\ V\ x = (0{::}nat) \Longrightarrow P\ x$
and *A1*: $!!x.\ V\ x > 0 \Longrightarrow \neg P\ x \Longrightarrow (\exists\, y.\ V\ y < V\ x \wedge \neg P\ y)$
shows $P\ x$
⟨*proof*⟩

Again, without explicit base case:

lemma *infinite-descent-measure*:
assumes $!!x.\ \neg\ P\ x \Longrightarrow \exists\, y.\ (V{::}'a{\Rightarrow}nat)\ y < V\ x \wedge \neg\ P\ y$ **shows** $P\ x$
⟨*proof*⟩

A [clumsy] way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
$[\![\; !!i\ j{::}nat.\ i < j \Longrightarrow f\ i < f\ j;\ i \leq j \;]\!] \Longrightarrow f\ i \leq ((f\ j){::}nat)$
⟨*proof*⟩

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j$ ==> $i + k \leq j + (k{::}nat)$
⟨*proof*⟩

non-strict, in both arguments

lemma *add-le-mono*: $[|\ i \leq j;\ \ k \leq l\ |]$ ==> $i + k \leq j + (l{::}nat)$

⟨*proof*⟩

lemma *le-add2*: $n \leq ((m + n)::nat)$
⟨*proof*⟩

lemma *le-add1*: $n \leq ((n + m)::nat)$
⟨*proof*⟩

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
⟨*proof*⟩

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
⟨*proof*⟩

lemma *less-iff-Suc-add*: $(m < n) = (\exists k.\ n = Suc\ (m + k))$
⟨*proof*⟩

lemma *trans-le-add1*: $(i::nat) \leq j \Longrightarrow i \leq j + m$
⟨*proof*⟩

lemma *trans-le-add2*: $(i::nat) \leq j \Longrightarrow i \leq m + j$
⟨*proof*⟩

lemma *trans-less-add1*: $(i::nat) < j \Longrightarrow i < j + m$
⟨*proof*⟩

lemma *trans-less-add2*: $(i::nat) < j \Longrightarrow i < m + j$
⟨*proof*⟩

lemma *add-lessD1*: $i + j < (k::nat) \Longrightarrow i < k$
⟨*proof*⟩

lemma *not-add-less1* [*iff*]: $\sim (i + j < (i::nat))$
⟨*proof*⟩

lemma *not-add-less2* [*iff*]: $\sim (j + i < (i::nat))$
⟨*proof*⟩

lemma *add-leD1*: $m + k \leq n \Longrightarrow m \leq (n::nat)$
⟨*proof*⟩

lemma *add-leD2*: $m + k \leq n \Longrightarrow k \leq (n::nat)$
⟨*proof*⟩

lemma *add-leE*: $(m::nat) + k \leq n \Longrightarrow (m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

needs $!!k$ for *add-ac* to work

lemma *less-add-eq-less*: $!!k::nat.\ k < l \Longrightarrow m + l = k + n \Longrightarrow m < n$

⟨*proof*⟩

15.4.8 More results about difference

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse*: ~ $m < n \Longrightarrow n + (m - n) = (m::nat)$
⟨*proof*⟩

lemma *le-add-diff-inverse* [*simp*]: $n \leq m \Longrightarrow n + (m - n) = (m::nat)$
⟨*proof*⟩

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m \Longrightarrow (m - n) + n = (m::nat)$
⟨*proof*⟩

lemma *Suc-diff-le*: $n \leq m \Longrightarrow Suc\ m - n = Suc\ (m - n)$
⟨*proof*⟩

lemma *diff-less-Suc*: $m - n < Suc\ m$
⟨*proof*⟩

lemma *diff-le-self* [*simp*]: $m - n \leq (m::nat)$
⟨*proof*⟩

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k.\ n = m + k)$
⟨*proof*⟩

lemma *less-imp-diff-less*: $(j::nat) < k \Longrightarrow j - n < k$
⟨*proof*⟩

lemma *diff-Suc-less* [*simp*]: $0<n \Longrightarrow n - Suc\ i < n$
⟨*proof*⟩

lemma *diff-add-assoc*: $k \leq (j::nat) \Longrightarrow (i + j) - k = i + (j - k)$
⟨*proof*⟩

lemma *diff-add-assoc2*: $k \leq (j::nat) \Longrightarrow (j + i) - k = (j - k) + i$
⟨*proof*⟩

lemma *le-imp-diff-is-add*: $i \leq (j::nat) \Longrightarrow (j - i = k) = (j = k + i)$
⟨*proof*⟩

lemma *diff-is-0-eq* [*simp*]: $((m::nat) - n = 0) = (m \leq n)$
⟨*proof*⟩

lemma *diff-is-0-eq'* [*simp*]: $m \leq n \Longrightarrow (m::nat) - n = 0$
⟨*proof*⟩

lemma *zero-less-diff* [*simp*]: $(0 < n - (m::nat)) = (m < n)$
⟨*proof*⟩

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k::nat.\ 0 < k\ \&\ i + k = j$
⟨*proof*⟩

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*:
fixes $n\ m :: nat$
shows $n - m + m = max\ n\ m$
⟨*proof*⟩

lemma *nat-diff-split*:
$P(a - b::nat) = ((a<b \longrightarrow P\ 0)\ \&\ (ALL\ d.\ a = b + d \longrightarrow P\ d))$
— elimination of $-$ on *nat*
⟨*proof*⟩

lemma *nat-diff-split-asm*:
$P(a - b::nat) = (\sim\ (a < b\ \&\ \sim\ P\ 0 \mid (EX\ d.\ a = b + d\ \&\ \sim\ P\ d)))$
— elimination of $-$ on *nat* in assumptions
⟨*proof*⟩

15.4.9 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \le (j::nat) \Longrightarrow i * k \le j * k$
⟨*proof*⟩

lemma *mult-le-mono2*: $i \le (j::nat) \Longrightarrow k * i \le k * j$
⟨*proof*⟩

$\le$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \le (j::nat) \Longrightarrow k \le l \Longrightarrow i * k \le j * l$
⟨*proof*⟩

lemma *mult-less-mono1*: $(i::nat) < j \Longrightarrow 0 < k \Longrightarrow i * k < j * k$
⟨*proof*⟩

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $(0 < (m::nat) * n) = (0 < m\ \&\ 0 < n)$
⟨*proof*⟩

lemma *one-le-mult-iff* [*simp*]: $(Suc\ 0 \le m * n) = (Suc\ 0 \le m\ \&\ Suc\ 0 \le n)$
⟨*proof*⟩

lemma *mult-less-cancel2* [*simp*]: $((m::nat) * k < n * k) = (0 < k\ \&\ m < n)$
⟨*proof*⟩

lemma *mult-less-cancel1* [*simp*]: $(k * (m::nat) < k * n) = (0 < k\ \&\ m < n)$

⟨*proof*⟩

lemma *mult-le-cancel1* [*simp*]: $(k * (m::nat) \leq k * n) = (0 < k \longrightarrow m \leq n)$
⟨*proof*⟩

lemma *mult-le-cancel2* [*simp*]: $((m::nat) * k \leq n * k) = (0 < k \longrightarrow m \leq n)$
⟨*proof*⟩

lemma *Suc-mult-less-cancel1*: $(Suc\ k * m < Suc\ k * n) = (m < n)$
⟨*proof*⟩

lemma *Suc-mult-le-cancel1*: $(Suc\ k * m \leq Suc\ k * n) = (m \leq n)$
⟨*proof*⟩

lemma *le-square*: $m \leq m * (m::nat)$
⟨*proof*⟩

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
⟨*proof*⟩

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n \Longrightarrow n = 1 \mid m = 0$
⟨*proof*⟩

the lattice order on *nat*

instantiation *nat* :: *distrib-lattice*
begin

definition
$(inf :: nat \Rightarrow nat \Rightarrow nat) = min$

definition
$(sup :: nat \Rightarrow nat \Rightarrow nat) = max$

instance ⟨*proof*⟩

end

15.5 Natural operation of natural numbers on functions

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

consts *compow* :: $nat \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

abbreviation *compower* :: $('a \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a \Rightarrow 'b$ (**infixr** $\hat{}\hat{}$ *80*) **where**
$f\ \hat{}\hat{}\ n \equiv compow\ n\ f$

notation (*latex* **output**)

compower ((-[^]) [*1000*] *1000*)

notation (*HTML* **output**)
compower ((-[^]) [*1000*] *1000*)

f ^^ *n* = *f o ... o f*, the n-fold composition of *f*

overloading
funpow == *compow* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ ('*a* ⇒ '*a*)
begin

primrec *funpow* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a* **where**
  funpow 0 f = *id*
| *funpow* (*Suc n*) *f* = *f o funpow n f*

end

for code generation

definition *funpow* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a* **where**
funpow-code-def [*code-post*]: *funpow* = *compow*

lemmas [*code-unfold*] = *funpow-code-def* [*symmetric*]

lemma [*code*]:
funpow 0 f = *id*
funpow (*Suc n*) *f* = *f o funpow n f*
⟨*proof*⟩

hide-const (**open**) *funpow*

lemma *funpow-add*:
f ^^ (*m* + *n*) = *f* ^^ *m* ∘ *f* ^^ *n*
⟨*proof*⟩

lemma *funpow-swap1*:
f ((*f* ^^ *n*) *x*) = (*f* ^^ *n*) (*f x*)
⟨*proof*⟩

15.6 Embedding of the Naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

primrec
of-nat :: *nat* ⇒ '*a*
where
of-nat-0: *of-nat 0* = *0*
| *of-nat-Suc*: *of-nat* (*Suc m*) = *1* + *of-nat m*

lemma *of-nat-1* [*simp*]: *of-nat 1* = *1*

⟨*proof*⟩

lemma *of-nat-add* [*simp*]: *of-nat* (*m* + *n*) = *of-nat m* + *of-nat n*
⟨*proof*⟩

lemma *of-nat-mult*: *of-nat* (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
⟨*proof*⟩

primrec *of-nat-aux* :: ('*a* ⇒ '*a*) ⇒ *nat* ⇒ '*a* ⇒ '*a* **where**
of-nat-aux inc 0 i = *i*
| *of-nat-aux inc* (*Suc n*) *i* = *of-nat-aux inc n* (*inc i*) — tail recursive

lemma *of-nat-code*:
of-nat n = *of-nat-aux* (λ*i*. *i* + *1*) *n 0*
⟨*proof*⟩

end

declare *of-nat-code* [*code*, *code-unfold*, *code-inline del*]

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0* = *semiring-1* +
assumes *of-nat-eq-iff* [*simp*]: *of-nat m* = *of-nat n* ⟷ *m* = *n*
begin

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*, *no-atp*]: *0* = *of-nat n* ⟷ *0* = *n*
⟨*proof*⟩

lemma *of-nat-eq-0-iff* [*simp*, *no-atp*]: *of-nat m* = *0* ⟷ *m* = *0*
⟨*proof*⟩

lemma *inj-of-nat*: *inj of-nat*
⟨*proof*⟩

end

context *linordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0* ≤ *of-nat m*
⟨*proof*⟩

lemma *less-imp-of-nat-less*: *m* < *n* ⟹ *of-nat m* < *of-nat n*
⟨*proof*⟩

lemma *of-nat-less-imp-less*: *of-nat m* < *of-nat n* ⟹ *m* < *n*
⟨*proof*⟩

lemma *of-nat-less-iff* [*simp*]: *of-nat m* < *of-nat n* ⟷ *m* < *n*
⟨*proof*⟩

lemma *of-nat-le-iff* [*simp*]: *of-nat m* ≤ *of-nat n* ⟷ *m* ≤ *n*
⟨*proof*⟩

Every *linordered-semidom* has characteristic zero.

subclass *semiring-char-0*
⟨*proof*⟩

Special cases where either operand is zero

lemma *of-nat-0-le-iff* [*simp*]: *0* ≤ *of-nat n*
⟨*proof*⟩

lemma *of-nat-le-0-iff* [*simp*, *no-atp*]: *of-nat m* ≤ *0* ⟷ *m* = *0*
⟨*proof*⟩

lemma *of-nat-0-less-iff* [*simp*]: *0* < *of-nat n* ⟷ *0* < *n*
⟨*proof*⟩

lemma *of-nat-less-0-iff* [*simp*]: ¬ *of-nat m* < *0*
⟨*proof*⟩

end

context *ring-1*
begin

lemma *of-nat-diff*: *n* ≤ *m* ⟹ *of-nat* (*m* − *n*) = *of-nat m* − *of-nat n*
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *abs-of-nat* [*simp*]: |*of-nat n*| = *of-nat n*
⟨*proof*⟩

end

lemma *of-nat-id* [*simp*]: *of-nat n* = *n*
⟨*proof*⟩

lemma *of-nat-eq-id* [*simp*]: *of-nat* = *id*
⟨*proof*⟩

15.7 The Set of Natural Numbers

context *semiring-1*
begin

definition
Nats :: *'a set* **where**
[*code del*]: *Nats* = *range of-nat*

notation (*xsymbols*)
Nats (ℕ)

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ ℕ
⟨*proof*⟩

lemma *Nats-0* [*simp*]: *0* ∈ ℕ
⟨*proof*⟩

lemma *Nats-1* [*simp*]: *1* ∈ ℕ
⟨*proof*⟩

lemma *Nats-add* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* + *b* ∈ ℕ
⟨*proof*⟩

lemma *Nats-mult* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* ∗ *b* ∈ ℕ
⟨*proof*⟩

lemma *Nats-cases* [*cases set*: *Nats*]:
assumes *x* ∈ ℕ
obtains (*of-nat*) *n* **where** *x* = *of-nat n*
⟨*proof*⟩

lemma *Nats-induct* [*case-names of-nat*, *induct set*: *Nats*]:
x ∈ ℕ ⟹ (⋀*n*. *P* (*of-nat n*)) ⟹ *P x*
⟨*proof*⟩

end

15.8 Further Arithmetic Facts Concerning the Natural Numbers

lemma *subst-equals*:
assumes *1*: *t* = *s* **and** *2*: *u* = *t*
shows *u* = *s*
⟨*proof*⟩

⟨*ML*⟩

lemmas [*arith-split*] = *nat-diff-split split-min split-max*

context *order*
begin

lemma *lift-Suc-mono-le*:
  assumes *mono*: !!n. $f\ n \leq f(Suc\ n)$ **and** $n \leq n'$
  shows $f\ n \leq f\ n'$
⟨*proof*⟩

lemma *lift-Suc-mono-less*:
  assumes *mono*: !!n. $f\ n < f(Suc\ n)$ **and** $n < n'$
  shows $f\ n < f\ n'$
⟨*proof*⟩

lemma *lift-Suc-mono-less-iff*:
  $(!!n.\ f\ n < f(Suc\ n)) \Longrightarrow f(n) < f(m) \longleftrightarrow n<m$
⟨*proof*⟩

end

lemma *mono-iff-le-Suc*: $mono\ f = (\forall\, n.\ f\ n \leq f\ (Suc\ n))$
⟨*proof*⟩

lemma *mono-nat-linear-lb*:
  $(!!m\ n{::}nat.\ m<n \Longrightarrow f\ m < f\ n) \Longrightarrow f(m)+k \leq f(m+k)$
⟨*proof*⟩

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: $[|\ a < (b{::}nat);\ c \leq a\ |] ==> a-c < b-c$
⟨*proof*⟩

lemma *less-diff-conv*: $(i < j-k) = (i+k < (j{::}nat))$
⟨*proof*⟩

lemma *le-diff-conv*: $(j-k \leq (i{::}nat)) = (j \leq i+k)$
⟨*proof*⟩

lemma *le-diff-conv2*: $k \leq j ==> (i \leq j-k) = (i+k \leq (j{::}nat))$
⟨*proof*⟩

lemma *diff-diff-cancel* [*simp*]: $i \leq (n{::}nat) ==> n - (n - i) = i$
⟨*proof*⟩

lemma *le-add-diff*: $k \leq (n{::}nat) ==> m \leq n + m - k$
⟨*proof*⟩

lemma *diff-less*[*simp*]: !!$m{::}nat$. $[|\ 0<n;\ 0<m\ |] ==> m - n < m$
⟨*proof*⟩

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $((m-k) - (n-k)) = (m-n)$
⟨*proof*⟩

hide-fact (**open**) *diff-diff-eq*

lemma *eq-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k = n-k) = (m=n)$
⟨*proof*⟩

lemma *less-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k < n-k) = (m<n)$
⟨*proof*⟩

lemma *le-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k \leq n-k) = (m \leq n)$
⟨*proof*⟩

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat)$ ==> $(m-l) \leq (n-l)$
⟨*proof*⟩

lemma *diff-le-mono2*: $m \leq (n::nat)$ ==> $(l-n) \leq (l-m)$
⟨*proof*⟩

lemma *diff-less-mono2*: [| $m < (n::nat)$; $m<l$ |] ==> $(l-n) < (l-m)$
⟨*proof*⟩

lemma *diffs0-imp-equal*: !!$m::nat$. [| $m-n = 0$; $n-m = 0$ |] ==> $m=n$
⟨*proof*⟩

lemma *min-diff*: $min\ (m - (i::nat))\ (n - i) = min\ m\ n - i$
⟨*proof*⟩

lemma *inj-on-diff-nat*:
assumes *k-le-n*: $\forall n \in N.\ k \leq (n::nat)$
shows *inj-on* $(\lambda n.\ n - k)\ N$
⟨*proof*⟩

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j$ --> $i - (j - k) = i + (k::nat) - j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j$ ==> $m - Suc\ (j - k) = m + k - Suc\ j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j$ ==> $Suc\ (j - k) - m = Suc\ j - (k + m)$
⟨*proof*⟩

Lemmas for ex/Factorization

lemma *one-less-mult*: [| $Suc\ 0 < n$; $Suc\ 0 < m$ |] ==> $Suc\ 0 < m*n$
⟨*proof*⟩

lemma *n-less-m-mult-n*: [| *Suc 0* < *n*; *Suc 0* < *m* |] ==> *n*<*m*∗*n*
⟨*proof*⟩

lemma *n-less-n-mult-m*: [| *Suc 0* < *n*; *Suc 0* < *m* |] ==> *n*<*n*∗*m*
⟨*proof*⟩

Specialized induction principles that work "backwards":

lemma *inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: *i* <= *j*
assumes *base*: *P j*
assumes *step*: !!*i*. [| *i* < *j*; *P* (*Suc i*) |] ==> *P i*
shows *P i*
⟨*proof*⟩

lemma *strict-inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: *i* < *j*
assumes *base*: !!*i*. *j* = *Suc i* ==> *P i*
assumes *step*: !!*i*. [| *i* < *j*; *P* (*Suc i*) |] ==> *P i*
shows *P i*
⟨*proof*⟩

lemma *zero-induct-lemma*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P* (*k* − *i*)
⟨*proof*⟩

lemma *zero-induct*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P 0*
⟨*proof*⟩

lemmas *add-diff-assoc* = *diff-add-assoc* [*symmetric*]
lemmas *add-diff-assoc2* = *diff-add-assoc2*[*symmetric*]
declare *diff-diff-left* [*simp*] *add-diff-assoc* [*simp*] *add-diff-assoc2*[*simp*]

At present we prove no analogue of *not-less-Least* or *Least-Suc*, since there appears to be no need.

15.9 The divides relation on *nat*

lemma *dvd-1-left* [*iff*]: *Suc 0 dvd k*
⟨*proof*⟩

lemma *dvd-1-iff-1* [*simp*]: (*m dvd Suc 0*) = (*m* = *Suc 0*)
⟨*proof*⟩

lemma *nat-dvd-1-iff-1* [*simp*]: *m dvd* (*1*::*nat*) ⟷ *m* = *1*
⟨*proof*⟩

lemma *dvd-antisym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
⟨*proof*⟩

op dvd is a partial order

interpretation *dvd*: *order op dvd* $\lambda n\ m :: nat.\ n\ dvd\ m \wedge \neg\ m\ dvd\ n$
⟨*proof*⟩

lemma *dvd-diff-nat*[*simp*]: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m*−*n* :: *nat*)
⟨*proof*⟩

lemma *dvd-diffD*: [| *k dvd m*−*n*; *k dvd n*; *n*≤*m* |] ==> *k dvd* (*m*::*nat*)
⟨*proof*⟩

lemma *dvd-diffD1*: [| *k dvd m*−*n*; *k dvd m*; *n*≤*m* |] ==> *k dvd* (*n*::*nat*)
⟨*proof*⟩

lemma *dvd-reduce*: (*k dvd n* + *k*) = (*k dvd* (*n*::*nat*))
⟨*proof*⟩

lemma *dvd-mult-cancel*: !!*k*::*nat*. [| *k*∗*m dvd k*∗*n*; *0*<*k* |] ==> *m dvd n*
⟨*proof*⟩

lemma *dvd-mult-cancel1*: *0*<*m* ==> (*m*∗*n dvd m*) = (*n* = (*1*::*nat*))
⟨*proof*⟩

lemma *dvd-mult-cancel2*: *0*<*m* ==> (*n*∗*m dvd m*) = (*n* = (*1*::*nat*))
⟨*proof*⟩

lemma *dvd-imp-le*: [| *k dvd n*; *0* < *n* |] ==> *k* ≤ (*n*::*nat*)
⟨*proof*⟩

lemma *nat-dvd-not-less*:
fixes *m n* :: *nat*
shows *0* < *m* ⟹ *m* < *n* ⟹ ¬ *n dvd m*
⟨*proof*⟩

15.10 size of a datatype value

class *size* =
fixes *size* :: *'a* ⇒ *nat* — see further theory *Wellfounded*

15.11 code module namespace

code-modulename *SML*
Nat Arith

code-modulename *OCaml*
Nat Arith

code-modulename *Haskell*
Nat Arith

end

16 Datatype: Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory *Datatype*
imports *Product-Type Sum-Type Nat*
uses
(*Tools/Datatype/datatype.ML*)
(*Tools/inductive-realizer.ML*)
(*Tools/Datatype/datatype-realizer.ML*)
begin

16.1 The datatype universe

typedef (*Node*)
(*'a*,*'b*) *node* = {*p*. *EX f x k*. *p* = (*f*::*nat*=>*'b*+*nat*, *x*::*'a*+*nat*) & *f k* = *Inr 0*}
— it is a subtype of (*nat*=>*'b*+*nat*) * (*'a*+*nat*)
⟨*proof*⟩

Datatypes will be represented by sets of type *node*

types *'a item* = (*'a*, *unit*) *node set*
(*'a*, *'b*) *dtree* = (*'a*, *'b*) *node set*

consts
Push :: [(*'b* + *nat*), *nat* => (*'b* + *nat*)] => (*nat* => (*'b* + *nat*))

Push-Node :: [(*'b* + *nat*), (*'a*, *'b*) *node*] => (*'a*, *'b*) *node*
ndepth :: (*'a*, *'b*) *node* => *nat*

Atom :: (*'a* + *nat*) => (*'a*, *'b*) *dtree*
Leaf :: *'a* => (*'a*, *'b*) *dtree*
Numb :: *nat* => (*'a*, *'b*) *dtree*
Scons :: [(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*
In0 :: (*'a*, *'b*) *dtree* => (*'a*, *'b*) *dtree*
In1 :: (*'a*, *'b*) *dtree* => (*'a*, *'b*) *dtree*
Lim :: (*'b* => (*'a*, *'b*) *dtree*) => (*'a*, *'b*) *dtree*

ntrunc :: [*nat*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*

uprod :: [(*'a*, *'b*) *dtree set*, (*'a*, *'b*) *dtree set*]=> (*'a*, *'b*) *dtree set*
usum :: [(*'a*, *'b*) *dtree set*, (*'a*, *'b*) *dtree set*]=> (*'a*, *'b*) *dtree set*

Split :: [[(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*]=>*'c*, (*'a*, *'b*) *dtree*] => *'c*
Case :: [[(*'a*, *'b*) *dtree*]=>*'c*, [(*'a*, *'b*) *dtree*]=>*'c*, (*'a*, *'b*) *dtree*] => *'c*

dprod :: [((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*, ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*]
=> ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*
dsum :: [((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*, ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*]
=> ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*

defs

Push-Node-def: *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

Push-def: *Push* == (%*b h*. *nat-case b h*)

Atom-def: *Atom* == (%*x*. {*Abs-Node*((%*k*. *Inr 0*, *x*))})
Scons-def: *Scons M N* == (*Push-Node* (*Inr 1*) ‘ *M*) *Un* (*Push-Node* (*Inr* (*Suc 1*)) ‘ *N*)

Leaf-def: *Leaf* == *Atom o Inl*
Numb-def: *Numb* == *Atom o Inr*

In0-def: *In0*(*M*) == *Scons* (*Numb 0*) *M*
In1-def: *In1*(*M*) == *Scons* (*Numb 1*) *M*

Lim-def: *Lim f* == *Union* {*z*. *? x*. *z* = *Push-Node* (*Inl x*) ‘ (*f x*)}

ndepth-def: *ndepth*(*n*) == (%(*f*,*x*). *LEAST k*. *f k* = *Inr 0*) (*Rep-Node n*)
ntrunc-def: *ntrunc k N* == {*n*. *n*:*N* & *ndepth*(*n*)<*k*}

uprod-def: *uprod A B* == *UN x*:*A*. *UN y*:*B*. { *Scons x y* }
usum-def: *usum A B* == *In0*‘*A Un In1*‘*B*

Split-def: *Split c M* == *THE u*. *EX x y*. *M* = *Scons x y* & *u* = *c x y*

Case-def: *Case c d M* == *THE u*. (*EX x* . *M* = *In0*(*x*) & *u* = *c*(*x*))
| (*EX y* . *M* = *In1*(*y*) & *u* = *d*(*y*))

dprod-def: *dprod r s* == *UN* (*x*,*x*′):*r*. *UN* (*y*,*y*′):*s*. {(*Scons x y*, *Scons x*′ *y*′)}

dsum-def: *dsum r s* == (*UN* (*x*,*x*′):*r*. {(*In0*(*x*),*In0*(*x*′))}) *Un*
(*UN* (*y*,*y*′):*s*. {(*In1*(*y*),*In1*(*y*′))})

lemma *apfst-convE*:
[| $q = apfst\ f\ p$; !!$x\ y$. [| $p = (x,y)$; $q = (f(x),y)$ |] ==> R
|] ==> R
⟨*proof*⟩

lemma *Push-inject1*: $Push\ i\ f = Push\ j\ g$ ==> $i=j$
⟨*proof*⟩

lemma *Push-inject2*: $Push\ i\ f = Push\ j\ g$ ==> $f=g$
⟨*proof*⟩

lemma *Push-inject*:
[| $Push\ i\ f = Push\ j\ g$; [| $i=j$; $f=g$ |] ==> P |] ==> P
⟨*proof*⟩

lemma *Push-neq-K0*: $Push\ (Inr\ (Suc\ k))\ f = (\%z.\ Inr\ 0)$ ==> P
⟨*proof*⟩

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*, *standard*]

lemma *Node-K0-I*: $(\%k.\ Inr\ 0,\ a) : Node$
⟨*proof*⟩

lemma *Node-Push-I*: p: *Node* ==> $apfst\ (Push\ i)\ p : Node$
⟨*proof*⟩

16.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: $Scons\ M\ N \neq Atom(a)$
⟨*proof*⟩

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*, *standard*]

lemma *inj-Atom*: $inj(Atom)$
⟨*proof*⟩
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*, *standard*]

lemma *Atom-Atom-eq* [*iff*]: $(Atom(a)=Atom(b)) = (a=b)$
⟨*proof*⟩

lemma *inj-Leaf*: *inj*(*Leaf*)
⟨*proof*⟩

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*, *standard*]

lemma *inj-Numb*: *inj*(*Numb*)
⟨*proof*⟩

lemmas *Numb-inject* [*dest*!] = *inj-Numb* [*THEN injD*, *standard*]

lemma *Push-Node-inject*:
[| *Push-Node i m* =*Push-Node j n*; [| *i*=*j*; *m*=*n* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M*<=*M'*
⟨*proof*⟩

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M' N'* ==> *N*<=*N'*
⟨*proof*⟩

lemma *Scons-inject1*: *Scons M N* = *Scons M' N'* ==> *M*=*M'*
⟨*proof*⟩

lemma *Scons-inject2*: *Scons M N* = *Scons M' N'* ==> *N*=*N'*
⟨*proof*⟩

lemma *Scons-inject*:
[| *Scons M N* = *Scons M' N'*; [| *M*=*M'*; *N*=*N'* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N* = *Scons M' N'*) = (*M*=*M'* & *N*=*N'*)
⟨*proof*⟩

lemma *Scons-not-Leaf* [*iff*]: *Scons M N* ≠ *Leaf*(*a*)
⟨*proof*⟩

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*, *standard*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*, *standard*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*, *standard*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr 0*, *x*)) = *0*
⟨*proof*⟩

lemma *ndepth-Push-Node-aux*:
nat-case (*Inr* (*Suc i*)) *f k* = *Inr 0* −−> *Suc*(*LEAST x*. *f x* = *Inr 0*) <= *k*
⟨*proof*⟩

lemma *ndepth-Push-Node*:
ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
⟨*proof*⟩

lemma *ntrunc-0* [*simp*]: *ntrunc 0 M* = {}
⟨*proof*⟩

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
⟨*proof*⟩

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
⟨*proof*⟩

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
⟨*proof*⟩

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)

⟨*proof*⟩

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

16.3 Set Constructions

lemma *uprodI* [*intro*!]: [| *M*:*A*; *N*:*B* |] ==> *Scons M N* : *uprod A B*
⟨*proof*⟩

lemma *uprodE* [*elim*!]:
[| *c* : *uprod A B*;
!!*x y*. [| *x*:*A*; *y*:*B*; *c* = *Scons x y* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *uprodE2*: [| *Scons M N* : *uprod A B*; [| *M*:*A*; *N*:*B* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *usum-In0I* [*intro*]: *M*:*A* ==> *In0*(*M*) : *usum A B*
⟨*proof*⟩

lemma *usum-In1I* [*intro*]: *N*:*B* ==> *In1*(*N*) : *usum A B*
⟨*proof*⟩

lemma *usumE* [*elim*!]:
[| *u* : *usum A B*;
!!*x*. [| *x*:*A*; *u*=*In0*(*x*) |] ==> *P*;

$$!!y.\ [|\ y{:}B;\ \ u{=}In1(y)\ |]\ ==>\ P$$
$$|]\ ==>\ P$$

⟨*proof*⟩

lemma *In0-not-In1* [*iff*]: $In0(M) \neq In1(N)$
⟨*proof*⟩

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*, *standard*]

lemma *In0-inject*: $In0(M) = In0(N) ==> M{=}N$
⟨*proof*⟩

lemma *In1-inject*: $In1(M) = In1(N) ==> M{=}N$
⟨*proof*⟩

lemma *In0-eq* [*iff*]: $(In0\ M = In0\ N) = (M{=}N)$
⟨*proof*⟩

lemma *In1-eq* [*iff*]: $(In1\ M = In1\ N) = (M{=}N)$
⟨*proof*⟩

lemma *inj-In0*: *inj In0*
⟨*proof*⟩

lemma *inj-In1*: *inj In1*
⟨*proof*⟩

lemma *Lim-inject*: $Lim\ f = Lim\ g ==> f = g$
⟨*proof*⟩

lemma *ntrunc-subsetI*: $ntrunc\ k\ M <= M$
⟨*proof*⟩

lemma *ntrunc-subsetD*: $(!!k.\ ntrunc\ k\ M <= N) ==> M{<=}N$
⟨*proof*⟩

lemma *ntrunc-equality*: $(!!k.\ ntrunc\ k\ M = ntrunc\ k\ N) ==> M{=}N$
⟨*proof*⟩

lemma *ntrunc-o-equality*:
[| !!k. (*ntrunc*(k) *o h1*) = (*ntrunc*(k) *o h2*) |] ==> *h1=h2*
⟨*proof*⟩

lemma *uprod-mono*: [| A<=A'; B<=B' |] ==> *uprod A B* <= *uprod A' B'*
⟨*proof*⟩

lemma *usum-mono*: [| A<=A'; B<=B' |] ==> *usum A B* <= *usum A' B'*
⟨*proof*⟩

lemma *Scons-mono*: [| M<=M'; N<=N' |] ==> *Scons M N* <= *Scons M' N'*
⟨*proof*⟩

lemma *In0-mono*: M<=N ==> *In0*(M) <= *In0*(N)
⟨*proof*⟩

lemma *In1-mono*: M<=N ==> *In1*(M) <= *In1*(N)
⟨*proof*⟩

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
⟨*proof*⟩

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = c(M)
⟨*proof*⟩

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = d(N)
⟨*proof*⟩

lemma *ntrunc-UN1*: *ntrunc k* (*UN x*. f(x)) = (*UN x*. *ntrunc k* (*f x*))
⟨*proof*⟩

lemma *Scons-UN1-x*: *Scons* (*UN x*. *f x*) *M* = (*UN x*. *Scons* (*f x*) *M*)
⟨*proof*⟩

lemma *Scons-UN1-y*: *Scons M* (*UN x*. *f x*) = (*UN x*. *Scons M* (*f x*))
⟨*proof*⟩

lemma *In0-UN1*: *In0*(*UN x*. f(x)) = (*UN x*. *In0*(f(x)))
⟨*proof*⟩

lemma *In1-UN1*: $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
⟨*proof*⟩

lemma *dprodI* [*intro*!]:
$[|\ (M,M'):r;\ \ (N,N'):s\ |] ==> (Scons\ M\ N,\ Scons\ M'\ N') : dprod\ r\ s$
⟨*proof*⟩

lemma *dprodE* [*elim*!]:
$[|\ c : dprod\ r\ s;$
$!!x\ y\ x'\ y'.\ [|\ (x,x') : r;\ \ (y,y') : s;$
$c = (Scons\ x\ y,\ Scons\ x'\ y')\ |] ==> P$
$|] ==> P$
⟨*proof*⟩

lemma *dsum-In0I* [*intro*]: $(M,M'):r ==> (In0(M),\ In0(M')) : dsum\ r\ s$
⟨*proof*⟩

lemma *dsum-In1I* [*intro*]: $(N,N'):s ==> (In1(N),\ In1(N')) : dsum\ r\ s$
⟨*proof*⟩

lemma *dsumE* [*elim*!]:
$[|\ w : dsum\ r\ s;$
$!!x\ x'.\ [|\ (x,x') : r;\ \ w = (In0(x),\ In0(x'))\ |] ==> P;$
$!!y\ y'.\ [|\ (y,y') : s;\ \ w = (In1(y),\ In1(y'))\ |] ==> P$
$|] ==> P$
⟨*proof*⟩

lemma *dprod-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dprod\ r\ s <= dprod\ r'\ s'$
⟨*proof*⟩

lemma *dsum-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dsum\ r\ s <= dsum\ r'\ s'$
⟨*proof*⟩

lemma *dprod-Sigma*: $(dprod\ (A <*> B)\ (C <*> D)) <= (uprod\ A\ C) <*> (uprod\ B\ D)$

⟨*proof*⟩

lemmas *dprod-subset-Sigma = subset-trans [OF dprod-mono dprod-Sigma, standard]*

lemma *dprod-subset-Sigma2*:
(dprod (Sigma A B) (Sigma C D)) <=
Sigma (uprod A C) (Split (%x y. uprod (B x) (D y)))
⟨*proof*⟩

lemma *dsum-Sigma*: *(dsum (A <*> B) (C <*> D)) <= (usum A C) <*> (usum B D)*
⟨*proof*⟩

lemmas *dsum-subset-Sigma = subset-trans [OF dsum-mono dsum-Sigma, standard]*

hides popular names

hide-type (**open**) *node item*
hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

⟨*ML*⟩

end

17 Record: Extensible records with structural subtyping

theory *Record*
imports *Datatype*
uses (*Tools/record.ML*)
begin

17.1 Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification *alpha (beta-update f rec) = alpha rec* for distinct fields alpha and beta of some record rec with n fields. There are *n ^ 2* such theorems, which prohibits storage of all of them for large n. The rules can be proved on the fly by case decomposition and simplification in O(n) time. By creating O(n) isomorphic-tuple types while defining the record, however, we can prove the access/update simplification in *O(log(n) ^2)* time.

The O(n) cost of case decomposition is not because O(n) steps are taken, but rather because the resulting rule must contain O(n) new variables and

an O(n) size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields $'a$, $'b$, $'c$ and $'d$ might be introduced as isomorphic to $'a \times ('b \times ('c \times 'd))$. If we balance the tuple tree to $('a \times 'b) \times ('c \times 'd)$ then accessors can be defined by converting to the underlying type then using O(log(n)) fst or snd operations. Updators can be defined similarly, if we introduce a *fst-update* and *snd-update* function. Furthermore, we can prove the access/update theorem in O(log(n)) steps by using simple rewrites on fst, snd, *fst-update* and *snd-update*.

The catch is that, although O(log(n)) steps were taken, the underlying type we converted to is a tuple tree of size O(n). Processing this term type wastes performance. We avoid this for large n by taking each subtree of size K and defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these O(n/K) new types, or, if $n > K*K$, we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant K.

If we prove the access/update theorem on this type with the analagous steps to the tuple tree, we consume $O(log(n)\ \hat{}2)$ time as the intermediate terms are $O(log(n))$ in size and the types needed have size bounded by K. To enable this analagous traversal, we define the functions seen below: *iso-tuple-fst*, *iso-tuple-snd*, *iso-tuple-fst-update* and *iso-tuple-snd-update*. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.

These rewrites are typically used in a structured way. They are here presented as the introduction rule *isomorphic-tuple.intros* rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

17.2 Operators and lemmas for types isomorphic to tuples

datatype $('a, 'b, 'c)$ *tuple-isomorphism* =
  Tuple-Isomorphism $'a \Rightarrow 'b \times 'c\ 'b \times 'c \Rightarrow 'a$

primrec
  repr :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'a \Rightarrow 'b \times 'c$ **where**
  repr (*Tuple-Isomorphism r a*) = *r*

primrec

abst :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'b \times 'c \Rightarrow 'a$ **where**
abst (*Tuple-Isomorphism r a*) $= a$

definition
iso-tuple-fst :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'a \Rightarrow 'b$ **where**
iso-tuple-fst isom $=$ *fst* $\circ$ *repr isom*

definition
iso-tuple-snd :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'a \Rightarrow 'c$ **where**
iso-tuple-snd isom $=$ *snd* $\circ$ *repr isom*

definition
iso-tuple-fst-update ::
$('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)$ **where**
iso-tuple-fst-update isom f $=$ *abst isom* $\circ$ *apfst f* $\circ$ *repr isom*

definition
iso-tuple-snd-update ::
$('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow ('c \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'a)$ **where**
iso-tuple-snd-update isom f $=$ *abst isom* $\circ$ *apsnd f* $\circ$ *repr isom*

definition
iso-tuple-cons ::
$('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'b \Rightarrow 'c \Rightarrow 'a$ **where**
iso-tuple-cons isom $=$ *curry* (*abst isom*)

17.3 Logical infrastructure for records

definition
iso-tuple-surjective-proof-assist :: $'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b) \Rightarrow$ *bool* **where**
iso-tuple-surjective-proof-assist x y f $\longleftrightarrow$ *f x* $=$ *y*

definition
iso-tuple-update-accessor-cong-assist ::
$(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow$ *bool* **where**
iso-tuple-update-accessor-cong-assist upd acc $\longleftrightarrow$
$(\forall f\ v.\ upd\ (\lambda x.\ f\ (acc\ v))\ v = upd\ f\ v) \wedge (\forall v.\ upd\ id\ v = v)$

definition
iso-tuple-update-accessor-eq-assist ::
$(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow$ *bool*
where
iso-tuple-update-accessor-eq-assist upd acc v f v' x $\longleftrightarrow$
upd f v $= v' \wedge$ *acc v* $= x \wedge$ *iso-tuple-update-accessor-cong-assist upd acc*

lemma *update-accessor-congruence-foldE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd acc*
and *r*: $r = r'$ **and** *v*: *acc r'* $= v'$
and *f*: $\bigwedge v.\ v' = v \Longrightarrow f\ v = f'\ v$

shows *upd f r = upd f' r'*
⟨*proof*⟩

lemma *update-accessor-congruence-unfoldE*:
iso-tuple-update-accessor-cong-assist upd acc ⟹
r = r' ⟹ acc r' = v' ⟹ (⋀v. v = v' ⟹ f v = f' v) ⟹
upd f r = upd f' r'
⟨*proof*⟩

lemma *iso-tuple-update-accessor-cong-assist-id*:
iso-tuple-update-accessor-cong-assist upd acc ⟹ upd id = id
⟨*proof*⟩

lemma *update-accessor-noopE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd acc*
and *acc*: *f (acc x) = acc x*
shows *upd f x = x*
⟨*proof*⟩

lemma *update-accessor-noop-compE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd acc*
and *acc*: *f (acc x) = acc x*
shows *upd (g ∘ f) x = upd g x*
⟨*proof*⟩

lemma *update-accessor-cong-assist-idI*:
iso-tuple-update-accessor-cong-assist id id
⟨*proof*⟩

lemma *update-accessor-cong-assist-triv*:
iso-tuple-update-accessor-cong-assist upd acc ⟹
iso-tuple-update-accessor-cong-assist upd acc
⟨*proof*⟩

lemma *update-accessor-accessor-eqE*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹ acc v = x
⟨*proof*⟩

lemma *update-accessor-updator-eqE*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹ upd f v = v'
⟨*proof*⟩

lemma *iso-tuple-update-accessor-eq-assist-idI*:
v' = f v ⟹ iso-tuple-update-accessor-eq-assist id id v f v' v
⟨*proof*⟩

lemma *iso-tuple-update-accessor-eq-assist-triv*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹
iso-tuple-update-accessor-eq-assist upd acc v f v' x

⟨*proof*⟩

lemma *iso-tuple-update-accessor-cong-from-eq*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟹
iso-tuple-update-accessor-cong-assist upd acc
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assistI*:
f x = *y* ⟹ *iso-tuple-surjective-proof-assist x y f*
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assist-idE*:
iso-tuple-surjective-proof-assist x y id ⟹ *x* = *y*
⟨*proof*⟩

locale *isomorphic-tuple* =
fixes *isom* :: ('*a*, '*b*, '*c*) *tuple-isomorphism*
assumes *repr-inv*: ⋀*x*. *abst isom* (*repr isom x*) = *x*
and *abst-inv*: ⋀*y*. *repr isom* (*abst isom y*) = *y*
begin

lemma *repr-inj*: *repr isom x* = *repr isom y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *abst-inj*: *abst isom x* = *abst isom y* ⟷ *x* = *y*
⟨*proof*⟩

lemmas *simps* = *Let-def repr-inv abst-inv repr-inj abst-inj*

lemma *iso-tuple-access-update-fst-fst*:
f o h g = *j o f* ⟹
(*f o iso-tuple-fst isom*) *o* (*iso-tuple-fst-update isom o h*) *g* =
j o (*f o iso-tuple-fst isom*)
⟨*proof*⟩

lemma *iso-tuple-access-update-snd-snd*:
f o h g = *j o f* ⟹
(*f o iso-tuple-snd isom*) *o* (*iso-tuple-snd-update isom o h*) *g* =
j o (*f o iso-tuple-snd isom*)
⟨*proof*⟩

lemma *iso-tuple-access-update-fst-snd*:
(*f o iso-tuple-fst isom*) *o* (*iso-tuple-snd-update isom o h*) *g* =
id o (*f o iso-tuple-fst isom*)
⟨*proof*⟩

lemma *iso-tuple-access-update-snd-fst*:
(*f o iso-tuple-snd isom*) *o* (*iso-tuple-fst-update isom o h*) *g* =
id o (*f o iso-tuple-snd isom*)

⟨*proof*⟩

lemma *iso-tuple-update-swap-fst-fst*:
h f o j g = j g o h f ⟹
(iso-tuple-fst-update isom o h) f o (iso-tuple-fst-update isom o j) g =
(iso-tuple-fst-update isom o j) g o (iso-tuple-fst-update isom o h) f
⟨*proof*⟩

lemma *iso-tuple-update-swap-snd-snd*:
h f o j g = j g o h f ⟹
(iso-tuple-snd-update isom o h) f o (iso-tuple-snd-update isom o j) g =
(iso-tuple-snd-update isom o j) g o (iso-tuple-snd-update isom o h) f
⟨*proof*⟩

lemma *iso-tuple-update-swap-fst-snd*:
(iso-tuple-snd-update isom o h) f o (iso-tuple-fst-update isom o j) g =
(iso-tuple-fst-update isom o j) g o (iso-tuple-snd-update isom o h) f
⟨*proof*⟩

lemma *iso-tuple-update-swap-snd-fst*:
(iso-tuple-fst-update isom o h) f o (iso-tuple-snd-update isom o j) g =
(iso-tuple-snd-update isom o j) g o (iso-tuple-fst-update isom o h) f
⟨*proof*⟩

lemma *iso-tuple-update-compose-fst-fst*:
h f o j g = k (f o g) ⟹
(iso-tuple-fst-update isom o h) f o (iso-tuple-fst-update isom o j) g =
(iso-tuple-fst-update isom o k) (f o g)
⟨*proof*⟩

lemma *iso-tuple-update-compose-snd-snd*:
h f o j g = k (f o g) ⟹
(iso-tuple-snd-update isom o h) f o (iso-tuple-snd-update isom o j) g =
(iso-tuple-snd-update isom o k) (f o g)
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assist-step*:
iso-tuple-surjective-proof-assist v a (iso-tuple-fst isom o f) ⟹
iso-tuple-surjective-proof-assist v b (iso-tuple-snd isom o f) ⟹
iso-tuple-surjective-proof-assist v (iso-tuple-cons isom a b) f
⟨*proof*⟩

lemma *iso-tuple-fst-update-accessor-cong-assist*:
assumes *iso-tuple-update-accessor-cong-assist f g*
shows *iso-tuple-update-accessor-cong-assist*
(iso-tuple-fst-update isom o f) (g o iso-tuple-fst isom)
⟨*proof*⟩

lemma *iso-tuple-snd-update-accessor-cong-assist*:

assumes *iso-tuple-update-accessor-cong-assist f g*
shows *iso-tuple-update-accessor-cong-assist*
(*iso-tuple-snd-update isom o f*) (*g o iso-tuple-snd isom*)
⟨*proof*⟩

lemma *iso-tuple-fst-update-accessor-eq-assist*:
assumes *iso-tuple-update-accessor-eq-assist f g a u a′ v*
shows *iso-tuple-update-accessor-eq-assist*
(*iso-tuple-fst-update isom o f*) (*g o iso-tuple-fst isom*)
(*iso-tuple-cons isom a b*) *u* (*iso-tuple-cons isom a′ b*) *v*
⟨*proof*⟩

lemma *iso-tuple-snd-update-accessor-eq-assist*:
assumes *iso-tuple-update-accessor-eq-assist f g b u b′ v*
shows *iso-tuple-update-accessor-eq-assist*
(*iso-tuple-snd-update isom o f*) (*g o iso-tuple-snd isom*)
(*iso-tuple-cons isom a b*) *u* (*iso-tuple-cons isom a b′*) *v*
⟨*proof*⟩

lemma *iso-tuple-cons-conj-eqI*:
a = *c* ∧ *b* = *d* ∧ *P* ⟷ *Q* ⟹
iso-tuple-cons isom a b = *iso-tuple-cons isom c d* ∧ *P* ⟷ *Q*
⟨*proof*⟩

lemmas *intros* =
iso-tuple-access-update-fst-fst
iso-tuple-access-update-snd-snd
iso-tuple-access-update-fst-snd
iso-tuple-access-update-snd-fst
iso-tuple-update-swap-fst-fst
iso-tuple-update-swap-snd-snd
iso-tuple-update-swap-fst-snd
iso-tuple-update-swap-snd-fst
iso-tuple-update-compose-fst-fst
iso-tuple-update-compose-snd-snd
iso-tuple-surjective-proof-assist-step
iso-tuple-fst-update-accessor-eq-assist
iso-tuple-snd-update-accessor-eq-assist
iso-tuple-fst-update-accessor-cong-assist
iso-tuple-snd-update-accessor-cong-assist
iso-tuple-cons-conj-eqI

end

lemma *isomorphic-tuple-intro*:
fixes *repr abst*
assumes *repr-inj*: ⋀*x y*. *repr x* = *repr y* ⟷ *x* = *y*
and *abst-inv*: ⋀*z*. *repr* (*abst z*) = *z*
and *v*: *v* ≡ *Tuple-Isomorphism repr abst*

shows *isomorphic-tuple v*
⟨*proof*⟩

definition
tuple-iso-tuple ≡ *Tuple-Isomorphism id id*

lemma *tuple-iso-tuple*:
isomorphic-tuple tuple-iso-tuple
⟨*proof*⟩

lemma *refl-conj-eq*: $Q = R \Longrightarrow P \wedge Q \longleftrightarrow P \wedge R$
⟨*proof*⟩

lemma *iso-tuple-UNIV-I*: $x \in UNIV \equiv True$
⟨*proof*⟩

lemma *iso-tuple-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
⟨*proof*⟩

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
⟨*proof*⟩

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
⟨*proof*⟩

lemma *o-eq-dest-lhs*: $a\ o\ b = c \Longrightarrow a\ (b\ v) = c\ v$
⟨*proof*⟩

lemma *o-eq-id-dest*: $a\ o\ b = id\ o\ c \Longrightarrow a\ (b\ v) = c\ v$
⟨*proof*⟩

17.4 Concrete record syntax

nonterminals
ident field-type field-types field fields field-update field-updates
syntax
-constify :: *id => ident* (-)
-constify :: *longid => ident* (-)

-field-type :: *ident => type => field-type* ((*2*- ::/ -))
:: *field-type => field-types* (-)
-field-types :: *field-type => field-types => field-types* (-,/ -)
-record-type :: *field-types => type* ((*3*'(| - |')))
-record-type-scheme :: *field-types => type => type* ((*3*'(| -,/ (*2*... ::/ -) |')))

-field :: *ident => 'a => field* ((*2*- =/ -))
:: *field => fields* (-)
-fields :: *field => fields => fields* (-,/ -)
-record :: *fields => 'a* ((*3*'(| - |')))

-record-scheme :: *fields* => *'a* => *'a* ((*3'*(| -,/ (*2*... =/ -) |')))

-field-update :: *ident* => *'a* => *field-update* ((*2*- :=/ -))
:: *field-update* => *field-updates* (-)
-field-updates :: *field-update* => *field-updates* => *field-updates* (-,/ -)
-record-update :: *'a* => *field-updates* => *'b* (-/(*3'*(| - |')) [*900*, *0*] *900*)

syntax (*xsymbols*)
-record-type :: *field-types* => *type* ((*3*(⦇-⦈)))
-record-type-scheme :: *field-types* => *type* => *type* ((*3*(⦇-,/ (*2*... ::/ -)⦈)))
-record :: *fields* => *'a* ((*3*(⦇-⦈)))
-record-scheme :: *fields* => *'a* => *'a* ((*3*(⦇-,/ (*2*... =/ -)⦈)))
-record-update :: *'a* => *field-updates* => *'b* (-/(*3*(⦇-⦈)) [*900*, *0*] *900*)

17.5 Record package

⟨*ML*⟩

hide-const (**open**) *Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist iso-tuple-update-accessor-eq-assist tuple-iso-tuple*

end

18 Power: Exponentiation

theory *Power*
imports *Nat*
begin

18.1 Powers for Arbitrary Monoids

class *power* = *one* + *times*
begin

primrec *power* :: *'a* ⇒ *nat* ⇒ *'a* (**infixr** ^ *80*) **where**
power-0: *a* ^ *0* = *1*
| *power-Suc*: *a* ^ *Suc n* = *a* ∗ *a* ^ *n*

notation (*latex* **output**)
power ((-[^]) [*1000*] *1000*)

notation (*HTML* **output**)
power ((-[^]) [*1000*] *1000*)

end

context *monoid-mult*
begin

subclass *power* ⟨*proof*⟩

lemma *power-one* [*simp*]:
1 ^ n = 1
⟨*proof*⟩

lemma *power-one-right* [*simp*]:
a ^ 1 = a
⟨*proof*⟩

lemma *power-commutes*:
*a ^ n * a = a * a ^ n*
⟨*proof*⟩

lemma *power-Suc2*:
*a ^ Suc n = a ^ n * a*
⟨*proof*⟩

lemma *power-add*:
*a ^ (m + n) = a ^ m * a ^ n*
⟨*proof*⟩

lemma *power-mult*:
*a ^ (m * n) = (a ^ m) ^ n*
⟨*proof*⟩

end

context *comm-monoid-mult*
begin

lemma *power-mult-distrib*:
*(a * b) ^ n = (a ^ n) * (b ^ n)*
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *of-nat-power*:
of-nat (m ^ n) = of-nat m ^ n
⟨*proof*⟩

end

context *comm-semiring-1*
begin

The divides relation

lemma *le-imp-power-dvd*:
  assumes $m \leq n$ **shows** $a \mathbin{\hat{}} m \; dvd \; a \mathbin{\hat{}} n$
⟨*proof*⟩

lemma *power-le-dvd*:
  $a \mathbin{\hat{}} n \; dvd \; b \Longrightarrow m \leq n \Longrightarrow a \mathbin{\hat{}} m \; dvd \; b$
  ⟨*proof*⟩

lemma *dvd-power-same*:
  $x \; dvd \; y \Longrightarrow x \mathbin{\hat{}} n \; dvd \; y \mathbin{\hat{}} n$
  ⟨*proof*⟩

lemma *dvd-power-le*:
  $x \; dvd \; y \Longrightarrow m \geq n \Longrightarrow x \mathbin{\hat{}} n \; dvd \; y \mathbin{\hat{}} m$
  ⟨*proof*⟩

lemma *dvd-power* [*simp*]:
  assumes $n > (0::nat) \vee x = 1$
  shows $x \; dvd \; (x \mathbin{\hat{}} n)$
⟨*proof*⟩

end

context *ring-1*
begin

lemma *power-minus*:
  $(-\, a) \mathbin{\hat{}} n = (-\, 1) \mathbin{\hat{}} n * a \mathbin{\hat{}} n$
⟨*proof*⟩

end

context *linordered-semidom*
begin

lemma *zero-less-power* [*simp*]:
  $0 < a \Longrightarrow 0 < a \mathbin{\hat{}} n$
  ⟨*proof*⟩

lemma *zero-le-power* [*simp*]:
  $0 \leq a \Longrightarrow 0 \leq a \mathbin{\hat{}} n$
  ⟨*proof*⟩

lemma *one-le-power*[*simp*]:
  $1 \leq a \Longrightarrow 1 \leq a \mathbin{\hat{}} n$

⟨*proof*⟩

lemma *power-gt1-lemma*:
assumes *gt1*: $1 < a$
shows $1 < a * a \hat{} n$
⟨*proof*⟩

lemma *power-gt1*:
$1 < a \Longrightarrow 1 < a \hat{} Suc\ n$
⟨*proof*⟩

lemma *one-less-power* [*simp*]:
$1 < a \Longrightarrow 0 < n \Longrightarrow 1 < a \hat{} n$
⟨*proof*⟩

lemma *power-le-imp-le-exp*:
assumes *gt1*: $1 < a$
shows $a \hat{} m \leq a \hat{} n \Longrightarrow m \leq n$
⟨*proof*⟩

Surely we can strengthen this? It holds for $0<a<1$ too.

lemma *power-inject-exp* [*simp*]:
$1 < a \Longrightarrow a \hat{} m = a \hat{} n \longleftrightarrow m = n$
⟨*proof*⟩

Can relax the first premise to $(0::'a) < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*:
$1 < a \Longrightarrow a \hat{} m < a \hat{} n \Longrightarrow m < n$
⟨*proof*⟩

lemma *power-mono*:
$a \leq b \Longrightarrow 0 \leq a \Longrightarrow a \hat{} n \leq b \hat{} n$
⟨*proof*⟩

lemma *power-strict-mono* [*rule-format*]:
$a < b \Longrightarrow 0 \leq a \Longrightarrow 0 < n \longrightarrow a \hat{} n < b \hat{} n$
⟨*proof*⟩

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:
$0 < a \Longrightarrow a < 1 \Longrightarrow a * a \hat{} n < a \hat{} n$
⟨*proof*⟩

lemma *power-strict-decreasing* [*rule-format*]:
$n < N \Longrightarrow 0 < a \Longrightarrow a < 1 \longrightarrow a \hat{} N < a \hat{} n$
⟨*proof*⟩

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing* [*rule-format*]:

$n \leq N \Longrightarrow 0 \leq a \Longrightarrow a \leq 1 \longrightarrow a \mathbin{\hat{}} N \leq a \mathbin{\hat{}} n$
⟨*proof*⟩

lemma *power-Suc-less-one*:
$0 < a \Longrightarrow a < 1 \Longrightarrow a \mathbin{\hat{}} Suc\ n < 1$
⟨*proof*⟩

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing* [*rule-format*]:
$n \leq N \Longrightarrow 1 \leq a \Longrightarrow a \mathbin{\hat{}} n \leq a \mathbin{\hat{}} N$
⟨*proof*⟩

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
$1 < a \Longrightarrow a \mathbin{\hat{}} n < a * a \mathbin{\hat{}} n$
⟨*proof*⟩

lemma *power-strict-increasing* [*rule-format*]:
$n < N \Longrightarrow 1 < a \longrightarrow a \mathbin{\hat{}} n < a \mathbin{\hat{}} N$
⟨*proof*⟩

lemma *power-increasing-iff* [*simp*]:
$1 < b \Longrightarrow b \mathbin{\hat{}} x \leq b \mathbin{\hat{}} y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *power-strict-increasing-iff* [*simp*]:
$1 < b \Longrightarrow b \mathbin{\hat{}} x < b \mathbin{\hat{}} y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *power-le-imp-le-base*:
assumes *le*: $a \mathbin{\hat{}} Suc\ n \leq b \mathbin{\hat{}} Suc\ n$
and *ynonneg*: $0 \leq b$
shows $a \leq b$
⟨*proof*⟩

lemma *power-less-imp-less-base*:
assumes *less*: $a \mathbin{\hat{}} n < b \mathbin{\hat{}} n$
assumes *nonneg*: $0 \leq b$
shows $a < b$
⟨*proof*⟩

lemma *power-inject-base*:
$a \mathbin{\hat{}} Suc\ n = b \mathbin{\hat{}} Suc\ n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a = b$
⟨*proof*⟩

lemma *power-eq-imp-eq-base*:
$a \mathbin{\hat{}} n = b \mathbin{\hat{}} n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < n \Longrightarrow a = b$
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *power-abs*:
abs (*a* ^ *n*) = *abs a* ^ *n*
⟨*proof*⟩

lemma *abs-power-minus* [*simp*]:
abs ((−*a*) ^ *n*) = *abs* (*a* ^ *n*)
⟨*proof*⟩

lemma *zero-less-power-abs-iff* [*simp*, *no-atp*]:
0 < *abs a* ^ *n* ⟷ *a* ≠ 0 ∨ *n* = 0
⟨*proof*⟩

lemma *zero-le-power-abs* [*simp*]:
0 ≤ *abs a* ^ *n*
⟨*proof*⟩

end

context *ring-1-no-zero-divisors*
begin

lemma *field-power-not-zero*:
a ≠ 0 ⟹ *a* ^ *n* ≠ 0
⟨*proof*⟩

end

context *division-ring*
begin

FIXME reorient or rename to *nonzero-inverse-power*

lemma *nonzero-power-inverse*:
a ≠ 0 ⟹ *inverse* (*a* ^ *n*) = (*inverse a*) ^ *n*
⟨*proof*⟩

end

context *field*
begin

lemma *nonzero-power-divide*:
b ≠ 0 ⟹ (*a* / *b*) ^ *n* = *a* ^ *n* / *b* ^ *n*
⟨*proof*⟩

end

lemma *power-0-Suc* [*simp*]:
$(0::'a::\{power,\ semiring\text{-}0\})\ \hat{}\ Suc\ n = 0$
⟨*proof*⟩

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*:
$0\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ (0::'a::\{power,\ semiring\text{-}0\}))$
⟨*proof*⟩

lemma *power-eq-0-iff* [*simp*]:
$a\ \hat{}\ n = 0 \longleftrightarrow$
$a = (0::'a::\{mult\text{-}zero,zero\text{-}neq\text{-}one,no\text{-}zero\text{-}divisors,power\}) \wedge n \neq 0$
⟨*proof*⟩

lemma (**in** *field*) *power-diff*:
assumes *nz*: $a \neq 0$
shows $n \leq m \Longrightarrow a\ \hat{}\ (m - n) = a\ \hat{}\ m\ /\ a\ \hat{}\ n$
⟨*proof*⟩

Perhaps these should be simprules.

lemma *power-inverse*:
fixes $a :: 'a::division\text{-}ring\text{-}inverse\text{-}zero$
shows $inverse\ (a\ \hat{}\ n) = inverse\ a\ \hat{}\ n$
⟨*proof*⟩

lemma *power-one-over*:
$1\ /\ (a::'a::\{field\text{-}inverse\text{-}zero,\ power\})\ \hat{}\ n = (1\ /\ a)\ \hat{}\ n$
⟨*proof*⟩

lemma *power-divide*:
$(a\ /\ b)\ \hat{}\ n = (a::'a::field\text{-}inverse\text{-}zero)\ \hat{}\ n\ /\ b\ \hat{}\ n$
⟨*proof*⟩

18.2 Exponentiation for the Natural Numbers

lemma *nat-one-le-power* [*simp*]:
$Suc\ 0 \leq i \Longrightarrow Suc\ 0 \leq i\ \hat{}\ n$
⟨*proof*⟩

lemma *nat-zero-less-power-iff* [*simp*]:
$x\ \hat{}\ n > 0 \longleftrightarrow x > (0::nat) \vee n = 0$
⟨*proof*⟩

lemma *nat-power-eq-Suc-0-iff* [*simp*]:
$x\ \hat{}\ m = Suc\ 0 \longleftrightarrow m = 0 \vee x = Suc\ 0$
⟨*proof*⟩

lemma *power-Suc-0* [*simp*]:
Suc 0 ^ n = Suc 0
⟨*proof*⟩

Valid for the naturals, but what if *0<i<1*? Premises cannot be weakened: consider the case where $i = (0::'a)$, $m = (1::'a)$ and $n = (0::'a)$.

lemma *nat-power-less-imp-less*:
assumes *nonneg*: *0 < (i::nat)*
assumes *less*: *i ^ m < i ^ n*
shows *m < n*
⟨*proof*⟩

lemma *power-dvd-imp-le*:
i ^ m dvd i ^ n ⟹ (1::nat) < i ⟹ m ≤ n
⟨*proof*⟩

18.3 Code generator tweak

lemma *power-power-power* [*code*, *code-unfold*, *code-inline del*]:
*power = power.power (1::'a::{power}) (op *)*
⟨*proof*⟩

declare *power.power.simps* [*code*]

code-modulename *SML*
Power Arith

code-modulename *OCaml*
Power Arith

code-modulename *Haskell*
Power Arith

end

19 Option: Datatype option

theory *Option*
imports *Datatype*
begin

datatype *'a option = None | Some 'a*

lemma *not-None-eq* [*iff*]: *(x ~= None) = (EX y. x = Some y)*
⟨*proof*⟩

lemma *not-Some-eq* [*iff*]: *(ALL y. x ~= Some y) = (x = None)*
⟨*proof*⟩

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

lemma *inj-Some* [*simp*]: *inj-on Some A*
⟨*proof*⟩

lemma *option-caseE*:
assumes *c*: (*case x of None => P | Some y => Q y*)
obtains
(*None*) *x = None* **and** *P*
| (*Some*) *y* **where** *x = Some y* **and** *Q y*
⟨*proof*⟩

lemma *UNIV-option-conv*: *UNIV = insert None* (*range Some*)
⟨*proof*⟩

19.0.1 Operations

primrec *the* :: *'a option => 'a* **where**
the (*Some x*) = *x*

primrec *set* :: *'a option => 'a set* **where**
set None = {} |
set (*Some x*) = {*x*}

lemma *ospec* [*dest*]: (*ALL x:set A. P x*) ==> *A = Some x* ==> *P x*
⟨*proof*⟩

⟨*ML*⟩

lemma *elem-set* [*iff*]: (*x : set xo*) = (*xo = Some x*)
⟨*proof*⟩

lemma *set-empty-eq* [*simp*]: (*set xo* = {}) = (*xo = None*)
⟨*proof*⟩

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a option* ⇒ *'b option* **where**
map = (%*f y. case y of None => None | Some x => Some* (*f x*))

lemma *option-map-None* [*simp*, *code*]: *map f None = None*
⟨*proof*⟩

lemma *option-map-Some* [*simp*, *code*]: *map f* (*Some x*) = *Some* (*f x*)
⟨*proof*⟩

lemma *option-map-is-None* [*iff*]:
(*map f opt = None*) = (*opt = None*)
⟨*proof*⟩

lemma *option-map-eq-Some* [*iff*]:
(*map f xo* = *Some y*) = (*EX z*. *xo* = *Some z* & *f z* = *y*)
⟨*proof*⟩

lemma *option-map-comp*:
map f (*map g opt*) = *map* (*f o g*) *opt*
⟨*proof*⟩

lemma *option-map-o-sum-case* [*simp*]:
map f o sum-case g h = *sum-case* (*map f o g*) (*map f o h*)
⟨*proof*⟩

hide-const (**open**) *set map*

19.0.2 Code generator setup

definition *is-none* :: *'a option* ⇒ *bool* **where**
[*code-post*]: *is-none x* ⟷ *x* = *None*

lemma *is-none-code* [*code*]:
shows *is-none None* ⟷ *True*
and *is-none* (*Some x*) ⟷ *False*
⟨*proof*⟩

lemma *is-none-none*:
is-none x ⟷ *x* = *None*
⟨*proof*⟩

lemma [*code-unfold*]:
eq-class.eq x None ⟷ *is-none x*
⟨*proof*⟩

hide-const (**open**) *is-none*

code-type *option*
(*SML - option*)
(*OCaml - option*)
(*Haskell Maybe -*)
(*Scala* !*Option*[(-)])

code-const *None* **and** *Some*
(*SML NONE* **and** *SOME*)
(*OCaml None* **and** *Some -*)
(*Haskell Nothing* **and** *Just*)
(*Scala None* **and** !*Some*((-)))

code-instance *option* :: *eq*
(*Haskell* −)

code-const *eq-class.eq* :: *'a::eq option* ⇒ *'a option* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-reserved *SML*
option NONE SOME

code-reserved *OCaml*
option None Some

code-reserved *Scala*
Option None Some

end

20 Finite-Set: Finite sets

theory *Finite-Set*
imports *Power Option*
begin

20.1 Predicate for finite sets

inductive *finite* :: *'a set* => *bool*
where
emptyI [*simp*, *intro*!]: *finite* {}
| *insertI* [*simp*, *intro*!]: *finite A* ==> *finite* (*insert a A*)

lemma *ex-new-if-finite*: — does not depend on def of finite at all
assumes ¬ *finite* (*UNIV* :: *'a set*) **and** *finite A*
shows ∃ *a*::*'a*. *a* ∉ *A*
⟨*proof*⟩

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
finite F ==>
P {} ==> (!!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)) ==>
P F
— Discharging *x* ∉ *F* entails extra work.
⟨*proof*⟩

lemma *finite-ne-induct*[*case-names singleton insert*, *consumes 2*]:
assumes *fin*: *finite F* **shows** *F* ≠ {} ⟹
⟦ ⋀*x*. *P*{*x*};
⋀*x F*. ⟦ *finite F*; *F* ≠ {}; *x* ∉ *F*; *P F* ⟧ ⟹ *P* (*insert x F*) ⟧
⟹ *P F*
⟨*proof*⟩

lemma *finite-subset-induct* [*consumes 2*, *case-names empty insert*]:

assumes *finite F* **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: $!!a\ F.$ *finite* F ==> $a \in A$ ==> $a \notin F$ ==> $P\ F$ ==> P (*insert* $a\ F$)
shows $P\ F$
⟨*proof*⟩

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*:
finite $A \Longrightarrow$ *ALL* x:A. (*EX* y. $P\ x\ y$) $\Longrightarrow$ *EX* f. *ALL* x:A. $P\ x\ (f\ x)$
⟨*proof*⟩

Finite sets are the images of initial segments of natural numbers:

lemma *finite-imp-nat-seg-image-inj-on*:
assumes *fin*: *finite A*
shows ∃ (n::*nat*) f. $A = f$ ‘ {i. i<n} & *inj-on f* {i. i<n}
⟨*proof*⟩

lemma *nat-seg-image-imp-finite*:
$!!f\ A.\ A = f$ ‘ {i::*nat*. i<n} $\Longrightarrow$ *finite A*
⟨*proof*⟩

lemma *finite-conv-nat-seg-image*:
finite A = (∃ (n::*nat*) f. $A = f$ ‘ {i::*nat*. i<n})
⟨*proof*⟩

lemma *finite-imp-inj-to-nat-seg*:
assumes *finite A*
shows *EX f n*::*nat*. *f*‘*A* = {i. i<n} & *inj-on f A*
⟨*proof*⟩

lemma *finite-Collect-less-nat*[*iff*]: *finite*{n::*nat*. n<k}
⟨*proof*⟩

Finiteness and set theoretic constructions

lemma *finite-UnI*: *finite F* ==> *finite G* ==> *finite* (*F Un G*)
⟨*proof*⟩

lemma *finite-subset*: $A \subseteq B$ ==> *finite B* ==> *finite A*
— Every subset of a finite set is finite.
⟨*proof*⟩

lemma *rev-finite-subset*: *finite B* ==> $A \subseteq B$ ==> *finite A*
⟨*proof*⟩

lemma *finite-Un* [*iff*]: *finite* (*F Un G*) = (*finite F* & *finite G*)
⟨*proof*⟩

lemma *finite-Collect-disjI*[*simp*]:

finite{*x*. *P x* | *Q x*} = (*finite*{*x*. *P x*} & *finite*{*x*. *Q x*})
⟨*proof*⟩

lemma *finite-Int* [*simp*, *intro*]: *finite F* | *finite G* ==> *finite* (*F Int G*)
— The converse obviously fails.
⟨*proof*⟩

lemma *finite-Collect-conjI* [*simp*, *intro*]:
finite{*x*. *P x*} | *finite*{*x*. *Q x*} ==> *finite*{*x*. *P x* & *Q x*}
— The converse obviously fails.
⟨*proof*⟩

lemma *finite-Collect-le-nat*[*iff*]: *finite*{*n*::*nat*. *n*<=*k*}
⟨*proof*⟩

lemma *finite-insert* [*simp*]: *finite* (*insert a A*) = *finite A*
⟨*proof*⟩

lemma *finite-Union*[*simp*, *intro*]:
⟦ *finite A*; !!*M*. *M* ∈ *A* ⟹ *finite M* ⟧ ⟹ *finite*(⋃ *A*)
⟨*proof*⟩

lemma *finite-Inter*[*intro*]: *EX A*:*M*. *finite*(*A*) ⟹ *finite*(*Inter M*)
⟨*proof*⟩

lemma *finite-INT*[*intro*]: *EX x*:*I*. *finite*(*A x*) ⟹ *finite*(*INT x*:*I*. *A x*)
⟨*proof*⟩

lemma *finite-empty-induct*:
assumes *finite A*
and *P A*
and !!*a A*. *finite A* ==> *a*:*A* ==> *P A* ==> *P* (*A* − {*a*})
shows *P* {}
⟨*proof*⟩

lemma *finite-Diff* [*simp*]: *finite A* ==> *finite* (*A* − *B*)
⟨*proof*⟩

lemma *finite-Diff2* [*simp*]:
assumes *finite B* **shows** *finite* (*A* − *B*) = *finite A*
⟨*proof*⟩

lemma *finite-compl*[*simp*]:
finite(*A*::'*a set*) ⟹ *finite*(−*A*) = *finite*(*UNIV*::'*a set*)
⟨*proof*⟩

lemma *finite-Collect-not*[*simp*]:
finite{*x*::'*a*. *P x*} ⟹ *finite*{*x*. ~*P x*} = *finite*(*UNIV*::'*a set*)
⟨*proof*⟩

lemma *finite-Diff-insert* [*iff*]: *finite* (*A* − *insert a B*) = *finite* (*A* − *B*)
⟨*proof*⟩

Image and Inverse Image over Finite Sets

lemma *finite-imageI*[*simp*]: *finite F* ==> *finite* (*h* ‘ *F*)
— The image of a finite set is finite.
⟨*proof*⟩

lemma *finite-image-set* [*simp*]:
finite {*x*. *P x*} ⟹ *finite* { *f x* | *x*. *P x* }
⟨*proof*⟩

lemma *finite-surj*: *finite A* ==> *B* <= *f* ‘ *A* ==> *finite B*
⟨*proof*⟩

lemma *finite-range-imageI*:
finite (*range g*) ==> *finite* (*range* (%*x*. *f* (*g x*)))
⟨*proof*⟩

lemma *finite-imageD*: *finite* (*f*‘*A*) ==> *inj-on f A* ==> *finite A*
⟨*proof*⟩

lemma *inj-vimage-singleton*: *inj f* ==> *f*−‘{*a*} ⊆ {*THE x*. *f x* = *a*}
— The inverse image of a singleton under an injective function is included in a singleton.
⟨*proof*⟩

lemma *finite-vimageI*: [|*finite F*; *inj h*|] ==> *finite* (*h* −‘ *F*)
— The inverse image of a finite set under an injective function is finite.
⟨*proof*⟩

lemma *finite-vimageD*:
assumes *fin*: *finite* (*h* −‘ *F*) **and** *surj*: *surj h*
shows *finite F*
⟨*proof*⟩

lemma *finite-vimage-iff*: *bij h* ⟹ *finite* (*h* −‘ *F*) ⟷ *finite F*
⟨*proof*⟩

The finite UNION of finite sets

lemma *finite-UN-I*: *finite A* ==> (!!*a*. *a*:*A* ==> *finite* (*B a*)) ==> *finite* (*UN a*:*A*. *B a*)
⟨*proof*⟩

Strengthen RHS to (∀ *x*∈*A*. *finite* (*B x*)) ∧ *finite* {*x* ∈ *A*. *B x* ≠ {}}?
We'd need to prove *finite C* ⟹ ∀ *A B*. *UNION A B* ⊆ *C* ⟶ *finite* {*x* ∈ *A*. *B x* ≠ {}} by induction.

lemma *finite-UN* [*simp*]:
finite A ==> finite (UNION A B) = (ALL x:A. finite (B x))
⟨*proof*⟩

lemma *finite-Collect-bex*[*simp*]: *finite A ⟹*
finite{x. EX y:A. Q x y} = (ALL y:A. finite{x. Q x y})
⟨*proof*⟩

lemma *finite-Collect-bounded-ex*[*simp*]: *finite{y. P y} ⟹*
finite{x. EX y. P y & Q x y} = (ALL y. P y ⟶ finite{x. Q x y})
⟨*proof*⟩

lemma *finite-Plus*: [| *finite A*; *finite B* |] ==> *finite (A <+> B)*
⟨*proof*⟩

lemma *finite-PlusD*:
fixes *A* :: *'a set* **and** *B* :: *'b set*
assumes *fin*: *finite (A <+> B)*
shows *finite A finite B*
⟨*proof*⟩

lemma *finite-Plus-iff*[*simp*]: *finite (A <+> B) ⟷ finite A ∧ finite B*
⟨*proof*⟩

lemma *finite-Plus-UNIV-iff*[*simp*]:
finite (UNIV :: ('a + 'b) set) =
(finite (UNIV :: 'a set) & finite (UNIV :: 'b set))
⟨*proof*⟩

Sigma of finite sets

lemma *finite-SigmaI* [*simp*]:
finite A ==> (!!a. a:A ==> finite (B a)) ==> finite (SIGMA a:A. B a)
⟨*proof*⟩

lemma *finite-cartesian-product*: [| *finite A*; *finite B* |] ==>
finite (A <> B)*
⟨*proof*⟩

lemma *finite-Prod-UNIV*:
*finite (UNIV::'a set) ==> finite (UNIV::'b set) ==> finite (UNIV::('a * 'b) set)*
⟨*proof*⟩

lemma *finite-cartesian-productD1*:
[| *finite (A <*> B)*; *B ≠ {}* |] ==> *finite A*
⟨*proof*⟩

lemma *finite-cartesian-productD2*:

[| *finite* (*A* <*> *B*); *A* ≠ {} |] ==> *finite B*
⟨*proof*⟩

The powerset of a finite set

lemma *finite-Pow-iff* [*iff*]: *finite* (*Pow A*) = *finite A*
⟨*proof*⟩

lemma *finite-Collect-subsets*[*simp*,*intro*]: *finite A* ⟹ *finite*{*B*. *B* ⊆ *A*}
⟨*proof*⟩

lemma *finite-UnionD*: *finite*(⋃ *A*) ⟹ *finite A*
⟨*proof*⟩

lemma *finite-subset-image*:
assumes *finite B*
shows *B* ⊆ *f* ‘ *A* ⟹ ∃ *C*⊆*A*. *finite C* ∧ *B* = *f* ‘ *C*
⟨*proof*⟩

20.2 Class *finite*

class *finite* =
assumes *finite-UNIV*: *finite* (*UNIV* :: *'a set*)
begin

lemma *finite* [*simp*]: *finite* (*A* :: *'a set*)
⟨*proof*⟩

end

lemma *UNIV-unit* [*no-atp*]:
UNIV = {()} ⟨*proof*⟩

instance *unit* :: *finite* ⟨*proof*⟩

lemma *UNIV-bool* [*no-atp*]:
UNIV = {*False*, *True*} ⟨*proof*⟩

instance *bool* :: *finite* ⟨*proof*⟩

instance ∗ :: (*finite*, *finite*) *finite* ⟨*proof*⟩

lemma *finite-option-UNIV* [*simp*]:
finite (*UNIV* :: *'a option set*) = *finite* (*UNIV* :: *'a set*)
⟨*proof*⟩

instance *option* :: (*finite*) *finite* ⟨*proof*⟩

lemma *inj-graph*: *inj* (%*f*. {(*x*, *y*). *y* = *f x*})
⟨*proof*⟩

instance *fun* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

instance + :: (*finite*, *finite*) *finite* ⟨*proof*⟩

20.3 A basic fold functional for finite sets

The intended behaviour is *fold f z* {x_1, ..., x_n} = *f* x_1 (... (*f* x_n *z*)...) if *f* is "left-commutative":

locale *fun-left-comm* =
fixes *f* :: 'a ⇒ 'b ⇒ 'b
assumes *fun-left-comm*: *f x* (*f y z*) = *f y* (*f x z*)
begin

On a functional level it looks much nicer:

lemma *fun-comp-comm*: *f x* ∘ *f y* = *f y* ∘ *f x*
⟨*proof*⟩

end

inductive *fold-graph* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a *set* ⇒ 'b ⇒ *bool*
for *f* :: 'a ⇒ 'b ⇒ 'b **and** *z* :: 'b **where**
emptyI [*intro*]: *fold-graph f z* {} *z* |
insertI [*intro*]: *x* ∉ *A* ⟹ *fold-graph f z A y*
⟹ *fold-graph f z* (*insert x A*) (*f x y*)

inductive-cases *empty-fold-graphE* [*elim*!]: *fold-graph f z* {} *x*

definition *fold* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a *set* ⇒ 'b **where**
[*code del*]: *fold f z A* = (*THE y*. *fold-graph f z A y*)

A tempting alternative for the definiens is *if finite A then THE y*. *fold-graph f z A y else e*. It allows the removal of finiteness assumptions from the theorems *fold-comm*, *fold-reindex* and *fold-distrib*. The proofs become ugly. It is not worth the effort. (???)

lemma *Diff1-fold-graph*:
fold-graph f z (*A* − {*x*}) *y* ⟹ *x* ∈ *A* ⟹ *fold-graph f z A* (*f x y*)
⟨*proof*⟩

lemma *fold-graph-imp-finite*: *fold-graph f z A x* ⟹ *finite A*
⟨*proof*⟩

lemma *finite-imp-fold-graph*: *finite A* ⟹ ∃ *x*. *fold-graph f z A x*
⟨*proof*⟩

20.3.1 From *fold-graph* to *fold*

context *fun-left-comm*
begin

lemma *fold-graph-insertE-aux*:
fold-graph f z A y $\Longrightarrow$ $a \in A \Longrightarrow \exists y'.\ y = f\ a\ y' \wedge$ *fold-graph f z* $(A - \{a\})\ y'$
⟨*proof*⟩

lemma *fold-graph-insertE*:
assumes *fold-graph f z* (*insert x A*) *v* **and** $x \notin A$
obtains *y* **where** $v = f\ x\ y$ **and** *fold-graph f z A y*
⟨*proof*⟩

lemma *fold-graph-determ*:
fold-graph f z A x $\Longrightarrow$ *fold-graph f z A y* $\Longrightarrow$ $y = x$
⟨*proof*⟩

lemma *fold-equality*:
fold-graph f z A y $\Longrightarrow$ *fold f z A* $= y$
⟨*proof*⟩

lemma *fold-graph-fold*: *finite A* $\Longrightarrow$ *fold-graph f z A* (*fold f z A*)
⟨*proof*⟩

The base case for *fold*:

lemma (**in** −) *fold-empty* [*simp*]: *fold f z* $\{\} = z$
⟨*proof*⟩

The various recursion equations for *fold*:

lemma *fold-insert* [*simp*]:
finite A ==> $x \notin A$ ==> *fold f z* (*insert x A*) $=$ *f x* (*fold f z A*)
⟨*proof*⟩

lemma *fold-fun-comm*:
finite A $\Longrightarrow$ *f x* (*fold f z A*) $=$ *fold f* (*f x z*) *A*
⟨*proof*⟩

lemma *fold-insert2*:
finite A $\Longrightarrow$ $x \notin A \Longrightarrow$ *fold f z* (*insert x A*) $=$ *fold f* (*f x z*) *A*
⟨*proof*⟩

lemma *fold-rec*:
assumes *finite A* **and** $x \in A$
shows *fold f z A* $=$ *f x* (*fold f z* $(A - \{x\})$)
⟨*proof*⟩

lemma *fold-insert-remove*:
assumes *finite A*
shows *fold f z* (*insert x A*) $=$ *f x* (*fold f z* $(A - \{x\})$)

⟨*proof*⟩

end

A simplified version for idempotent functions:

locale *fun-left-comm-idem* = *fun-left-comm* +
  assumes *fun-left-idem*: *f x* (*f x z*) = *f x z*
begin

The nice version:

lemma *fun-comp-idem* : *f x o f x* = *f x*
⟨*proof*⟩

lemma *fold-insert-idem*:
  assumes *fin*: *finite A*
  shows *fold f z* (*insert x A*) = *f x* (*fold f z A*)
⟨*proof*⟩

declare *fold-insert*[*simp del*] *fold-insert-idem*[*simp*]

lemma *fold-insert-idem2*:
  finite A ⟹ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
⟨*proof*⟩

end

20.3.2 Expressing set operations via *fold*

lemma (**in** *fun-left-comm*) *fun-left-comm-apply*:
  fun-left-comm (λ*x*. *f* (*g x*))
⟨*proof*⟩

lemma (**in** *fun-left-comm-idem*) *fun-left-comm-idem-apply*:
  fun-left-comm-idem (λ*x*. *f* (*g x*))
  ⟨*proof*⟩

lemma *fun-left-comm-idem-insert*:
  fun-left-comm-idem insert
⟨*proof*⟩

lemma *fun-left-comm-idem-remove*:
  fun-left-comm-idem (λ*x A*. *A* − {*x*})
⟨*proof*⟩

lemma (**in** *semilattice-inf*) *fun-left-comm-idem-inf*:
  fun-left-comm-idem inf
⟨*proof*⟩

lemma (**in** *semilattice-sup*) *fun-left-comm-idem-sup*:

fun-left-comm-idem sup
⟨*proof*⟩

lemma *union-fold-insert*:
assumes *finite A*
shows $A \cup B = fold\ insert\ B\ A$
⟨*proof*⟩

lemma *minus-fold-remove*:
assumes *finite A*
shows $B - A = fold\ (\lambda x\ A.\ A - \{x\})\ B\ A$
⟨*proof*⟩

context *complete-lattice*
begin

lemma *inf-Inf-fold-inf*:
assumes *finite A*
shows $inf\ B\ (Inf\ A) = fold\ inf\ B\ A$
⟨*proof*⟩

lemma *sup-Sup-fold-sup*:
assumes *finite A*
shows $sup\ B\ (Sup\ A) = fold\ sup\ B\ A$
⟨*proof*⟩

lemma *Inf-fold-inf*:
assumes *finite A*
shows $Inf\ A = fold\ inf\ top\ A$
⟨*proof*⟩

lemma *Sup-fold-sup*:
assumes *finite A*
shows $Sup\ A = fold\ sup\ bot\ A$
⟨*proof*⟩

lemma *inf-INFI-fold-inf*:
assumes *finite A*
shows $inf\ B\ (INFI\ A\ f) = fold\ (\lambda A.\ inf\ (f\ A))\ B\ A$ (**is** *?inf = ?fold*)
⟨*proof*⟩

lemma *sup-SUPR-fold-sup*:
assumes *finite A*
shows $sup\ B\ (SUPR\ A\ f) = fold\ (\lambda A.\ sup\ (f\ A))\ B\ A$ (**is** *?sup = ?fold*)
⟨*proof*⟩

lemma *INFI-fold-inf*:
assumes *finite A*
shows $INFI\ A\ f = fold\ (\lambda A.\ inf\ (f\ A))\ top\ A$

⟨*proof*⟩

lemma *SUPR-fold-sup*:
assumes *finite A*
shows *SUPR A f* = *fold* (λ*A*. *sup* (*f A*)) *bot A*
⟨*proof*⟩

end

20.4 The derived combinator *fold-image*

definition *fold-image* :: ('*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*b* ⇒ '*a set* ⇒ '*b*
where *fold-image f g* = *fold* (%*x y*. *f* (*g x*) *y*)

lemma *fold-image-empty*[*simp*]: *fold-image f g z* {} = *z*
⟨*proof*⟩

context *ab-semigroup-mult*
begin

lemma *fold-image-insert*[*simp*]:
assumes *finite A* **and** *a* ∉ *A*
shows *fold-image times g z* (*insert a A*) = *g a* ∗ (*fold-image times g z A*)
⟨*proof*⟩

lemma *fold-image-reindex*:
assumes *fin*: *finite A*
shows *inj-on h A* ⟹ *fold-image times g z* (*h*‘*A*) = *fold-image times* (*g*∘*h*) *z A*
⟨*proof*⟩

lemma *fold-image-cong*:
finite A ⟹
(!!*x*. *x*:*A* ==> *g x* = *h x*) ==> *fold-image times g z A* = *fold-image times h z A*
⟨*proof*⟩

end

context *comm-monoid-mult*
begin

lemma *fold-image-1*:
finite S ⟹ (∀ *x*∈*S*. *f x* = *1*) ⟹ *fold-image op* ∗ *f 1 S* = *1*
⟨*proof*⟩

lemma *fold-image-Un-Int*:

finite A ==> finite B ==>
*fold-image times g 1 A * fold-image times g 1 B =*
*fold-image times g 1 (A Un B) * fold-image times g 1 (A Int B)*
⟨*proof*⟩

lemma *fold-image-Un-one*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: ∀ *x* ∈ *S*∩*T*. *f x = 1*
shows *fold-image (op *) f 1 (S ∪ T) = fold-image (op *) f 1 S * fold-image (op *) f 1 T*
⟨*proof*⟩

corollary *fold-Un-disjoint*:
finite A ==> finite B ==> A Int B = {} ==>
fold-image times g 1 (A Un B) =
*fold-image times g 1 A * fold-image times g 1 B*
⟨*proof*⟩

lemma *fold-image-UN-disjoint*:
⟦ *finite I*; *ALL i:I. finite (A i)*;
ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {} ⟧
⟹ *fold-image times g 1 (UNION I A) =*
fold-image times (%i. fold-image times g 1 (A i)) 1 I
⟨*proof*⟩

lemma *fold-image-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
fold-image times (%x. fold-image times (g x) 1 (B x)) 1 A =
fold-image times (split g) 1 (SIGMA x:A. B x)
⟨*proof*⟩

lemma *fold-image-distrib*: *finite A* ⟹
*fold-image times (%x. g x * h x) 1 A =*
*fold-image times g 1 A * fold-image times h 1 A*
⟨*proof*⟩

lemma *fold-image-related*:
assumes *Re*: *R e e*
and *Rop*: ∀ *x1 y1 x2 y2*. *R x1 x2* ∧ *R y1 y2* ⟶ *R (x1 * y1) (x2 * y2)*
and *fS*: *finite S* **and** *Rfg*: ∀ *x*∈*S*. *R (h x) (g x)*
shows *R (fold-image (op *) h e S) (fold-image (op *) g e S)*
⟨*proof*⟩

lemma *fold-image-eq-general*:
assumes *fS*: *finite S*
and *h*: ∀ *y*∈*S'*. ∃!*x*. *x*∈ *S* ∧ *h*(*x*) = *y*
and *f12*: ∀ *x*∈*S*. *h x* ∈ *S'* ∧ *f2*(*h x*) = *f1 x*
shows *fold-image (op *) f1 e S = fold-image (op *) f2 e S'*
⟨*proof*⟩

lemma *fold-image-eq-general-inverses*:
assumes *fS*: *finite S*
and *kh*: $\bigwedge y.\ y \in T \Longrightarrow k\ y \in S \wedge h\ (k\ y) = y$
and *hk*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x\ \wedge g\ (h\ x) = f\ x$
shows *fold-image* $(op\ *)\ f\ e\ S =$ *fold-image* $(op\ *)\ g\ e\ T$

⟨*proof*⟩

end

20.5 A fold functional for non-empty sets

Does not require start value.

inductive
fold1Set :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow bool$
for f :: $'a \Rightarrow 'a \Rightarrow 'a$
where
fold1Set-insertI [*intro*]:
$[\![$ *fold-graph f a A x*; $a \notin A\ ]\!] \Longrightarrow$ *fold1Set f* (*insert a A*) *x*

definition *fold1* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a$ **where**
fold1 f A == *THE x. fold1Set f A x*

lemma *fold1Set-nonempty*:
fold1Set f A x $\Longrightarrow A \neq \{\}$
⟨*proof*⟩

inductive-cases *empty-fold1SetE* [*elim*!]: *fold1Set f* $\{\}$ *x*

inductive-cases *insert-fold1SetE* [*elim*!]: *fold1Set f* (*insert a X*) *x*

lemma *fold1Set-sing* [*iff*]: (*fold1Set f* $\{a\}$ *b*) = $(a = b)$
⟨*proof*⟩

lemma *fold1-singleton* [*simp*]: *fold1 f* $\{a\} = a$
⟨*proof*⟩

lemma *finite-nonempty-imp-fold1Set*:
$[\![$ *finite A*; $A \neq \{\}\ ]\!] \Longrightarrow$ *EX x. fold1Set f A x*
⟨*proof*⟩

First, some lemmas about *fold-graph*.

context *ab-semigroup-mult*
begin

lemma *fun-left-comm*: *fun-left-comm*$(op\ *)$
⟨*proof*⟩

lemma *fold-graph-insert-swap*:
assumes *fold*: *fold-graph times* (*b*::'*a*) *A y* **and** $b \notin A$
shows *fold-graph times z* (*insert b A*) (*z* ∗ *y*)
⟨*proof*⟩

lemma *fold-graph-permute-diff*:
assumes *fold*: *fold-graph times b A x*
shows !!*a*. ⟦*a* ∈ *A*; *b* ∉ *A*⟧ ⟹ *fold-graph times a* (*insert b* (*A*−{*a*})) *x*
⟨*proof*⟩

lemma *fold1-eq-fold*:
assumes *finite A a* ∉ *A* **shows** *fold1 times* (*insert a A*) = *fold times a A*
⟨*proof*⟩

lemma *nonempty-iff*: (*A* ≠ {}) = (∃ *x B*. *A* = *insert x B* & *x* ∉ *B*)
⟨*proof*⟩

lemma *fold1-insert*:
assumes *nonempty*: *A* ≠ {} **and** *A*: *finite A x* ∉ *A*
shows *fold1 times* (*insert x A*) = *x* ∗ *fold1 times A*
⟨*proof*⟩

end

context *ab-semigroup-idem-mult*
begin

lemma *fun-left-comm-idem*: *fun-left-comm-idem*(*op* ∗)
⟨*proof*⟩

lemma *fold1-insert-idem* [*simp*]:
assumes *nonempty*: *A* ≠ {} **and** *A*: *finite A*
shows *fold1 times* (*insert x A*) = *x* ∗ *fold1 times A*
⟨*proof*⟩

lemma *hom-fold1-commute*:
assumes *hom*: !!*x y*. *h* (*x* ∗ *y*) = *h x* ∗ *h y*
and *N*: *finite N N* ≠ {} **shows** *h* (*fold1 times N*) = *fold1 times* (*h* ‘ *N*)
⟨*proof*⟩

lemma *fold1-eq-fold-idem*:
assumes *finite A*
shows *fold1 times* (*insert a A*) = *fold times a A*
⟨*proof*⟩

end

Now the recursion rules for definitions:

lemma *fold1-singleton-def*: *g* = *fold1 f* ⟹ *g* {*a*} = *a*

⟨*proof*⟩

lemma (**in** *ab-semigroup-mult*) *fold1-insert-def*:
⟦ *g* = *fold1 times*; *finite A*; *x* ∉ *A*; *A* ≠ {} ⟧ ⟹ *g* (*insert x A*) = *x* ∗ *g A*
⟨*proof*⟩

lemma (**in** *ab-semigroup-idem-mult*) *fold1-insert-idem-def*:
⟦ *g* = *fold1 times*; *finite A*; *A* ≠ {} ⟧ ⟹ *g* (*insert x A*) = *x* ∗ *g A*
⟨*proof*⟩

20.5.1 Determinacy for *fold1Set*

declare
empty-fold-graphE [*rule del*] *fold-graph.intros* [*rule del*]
empty-fold1SetE [*rule del*] *insert-fold1SetE* [*rule del*]
— No more proofs involve these relations.

20.5.2 Lemmas about *fold1*

context *ab-semigroup-mult*
begin

lemma *fold1-Un*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹ *A Int B* = {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
⟨*proof*⟩

lemma *fold1-in*:
assumes *A*: *finite* (*A*) *A* ≠ {} **and** *elem*: ⋀*x y*. *x* ∗ *y* ∈ {*x*,*y*}
shows *fold1 times A* ∈ *A*
⟨*proof*⟩

end

lemma (**in** *ab-semigroup-idem-mult*) *fold1-Un2*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
⟨*proof*⟩

20.6 Locales as mini-packages for fold operations

20.6.1 The natural case

locale *folding* =
fixes *f* :: *'a* ⇒ *'b* ⇒ *'b*
fixes *F* :: *'a set* ⇒ *'b* ⇒ *'b*
assumes *commute-comp*: *f y* ∘ *f x* = *f x* ∘ *f y*
assumes *eq-fold*: *finite A* ⟹ *F A s* = *fold f s A*

begin

lemma *empty* [*simp*]:
F {} = *id*
⟨*proof*⟩

lemma *insert* [*simp*]:
assumes *finite A* **and** $x \notin A$
shows *F* (*insert x A*) = *F A* ∘ *f x*
⟨*proof*⟩

lemma *remove*:
assumes *finite A* **and** $x \in A$
shows *F A* = *F* (*A* − {*x*}) ∘ *f x*
⟨*proof*⟩

lemma *insert-remove*:
assumes *finite A*
shows *F* (*insert x A*) = *F* (*A* − {*x*}) ∘ *f x*
⟨*proof*⟩

lemma *commute-left-comp*:
f y ∘ (*f x* ∘ *g*) = *f x* ∘ (*f y* ∘ *g*)
⟨*proof*⟩

lemma *commute-comp′*:
assumes *finite A*
shows *f x* ∘ *F A* = *F A* ∘ *f x*
⟨*proof*⟩

lemma *commute-left-comp′*:
assumes *finite A*
shows *f x* ∘ (*F A* ∘ *g*) = *F A* ∘ (*f x* ∘ *g*)
⟨*proof*⟩

lemma *commute-comp″*:
assumes *finite A* **and** *finite B*
shows *F B* ∘ *F A* = *F A* ∘ *F B*
⟨*proof*⟩

lemma *commute-left-comp″*:
assumes *finite A* **and** *finite B*
shows *F B* ∘ (*F A* ∘ *g*) = *F A* ∘ (*F B* ∘ *g*)
⟨*proof*⟩

lemmas *commute-comps* = *o-assoc* [*symmetric*] *commute-comp commute-left-comp commute-comp′ commute-left-comp′ commute-comp″ commute-left-comp″*

lemma *union-inter*:

assumes *finite A* **and** *finite B*
shows $F\ (A \cup B) \circ F\ (A \cap B) = F\ A \circ F\ B$
⟨*proof*⟩

lemma *union*:
assumes *finite A* **and** *finite B*
and $A \cap B = \{\}$
shows $F\ (A \cup B) = F\ A \circ F\ B$
⟨*proof*⟩

end

20.6.2 The natural case with idempotency

locale *folding-idem* = *folding* +
assumes *idem-comp*: $f\ x \circ f\ x = f\ x$
begin

lemma *idem-left-comp*:
$f\ x \circ (f\ x \circ g) = f\ x \circ g$
⟨*proof*⟩

lemma *in-comp-idem*:
assumes *finite A* **and** $x \in A$
shows $F\ A \circ f\ x = F\ A$
⟨*proof*⟩

lemma *subset-comp-idem*:
assumes *finite A* **and** $B \subseteq A$
shows $F\ A \circ F\ B = F\ A$
⟨*proof*⟩

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
assumes *finite A*
shows $F\ (insert\ x\ A) = F\ A \circ f\ x$
⟨*proof*⟩

lemma *union-idem*:
assumes *finite A* **and** *finite B*
shows $F\ (A \cup B) = F\ A \circ F\ B$
⟨*proof*⟩

end

20.6.3 The image case with fixed function

no-notation *times* (**infixl** $*$ *70*)
no-notation *Groups.one* (*1*)

locale *folding-image-simple* = *comm-monoid* +
  fixes $g :: ('b \Rightarrow 'a)$
  fixes $F :: 'b\ set \Rightarrow 'a$
  assumes *eq-fold-g*: *finite* $A \Longrightarrow F\ A = fold\text{-}image\ f\ g\ 1\ A$
begin

lemma *empty* [*simp*]:
  $F\ \{\} = 1$
  ⟨*proof*⟩

lemma *insert* [*simp*]:
  assumes *finite* A **and** $x \notin A$
  shows $F\ (insert\ x\ A) = g\ x * F\ A$
⟨*proof*⟩

lemma *remove*:
  assumes *finite* A **and** $x \in A$
  shows $F\ A = g\ x * F\ (A - \{x\})$
⟨*proof*⟩

lemma *insert-remove*:
  assumes *finite* A
  shows $F\ (insert\ x\ A) = g\ x * F\ (A - \{x\})$
  ⟨*proof*⟩

lemma *neutral*:
  assumes *finite* A **and** $\forall x{\in}A.\ g\ x = 1$
  shows $F\ A = 1$
  ⟨*proof*⟩

lemma *union-inter*:
  assumes *finite* A **and** *finite* B
  shows $F\ (A \cup B) * F\ (A \cap B) = F\ A * F\ B$
⟨*proof*⟩

corollary *union-inter-neutral*:
  assumes *finite* A **and** *finite* B
  and *I0*: $\forall x \in A{\cap}B.\ g\ x = 1$
  shows $F\ (A \cup B) = F\ A * F\ B$
  ⟨*proof*⟩

corollary *union-disjoint*:
  assumes *finite* A **and** *finite* B
  assumes $A \cap B = \{\}$
  shows $F\ (A \cup B) = F\ A * F\ B$
  ⟨*proof*⟩

end

20.6.4 The image case with flexible function

locale *folding-image = comm-monoid +*
  fixes $F :: ('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'a$
  assumes *eq-fold*: $\bigwedge g.\ finite\ A \Longrightarrow F\ g\ A = fold\text{-}image\ f\ g\ 1\ A$

sublocale *folding-image < folding-image-simple op * 1 g F g* ⟨*proof*⟩

context *folding-image*
begin

lemma *reindex*:
  assumes *finite A* **and** *inj-on h A*
  shows $F\ g\ (h\ `\ A) = F\ (g \circ h)\ A$
  ⟨*proof*⟩

lemma *cong*:
  assumes *finite A* **and** $\bigwedge x.\ x \in A \Longrightarrow g\ x = h\ x$
  shows $F\ g\ A = F\ h\ A$
⟨*proof*⟩

lemma *UNION-disjoint*:
  assumes *finite I* **and** $\forall i{\in}I.\ finite\ (A\ i)$
  and $\forall i{\in}I.\ \forall j{\in}I.\ i \neq j \longrightarrow A\ i \cap A\ j = \{\}$
  shows $F\ g\ (UNION\ I\ A) = F\ (F\ g \circ A)\ I$
⟨*proof*⟩

lemma *distrib*:
  assumes *finite A*
  shows $F\ (\lambda x.\ g\ x * h\ x)\ A = F\ g\ A * F\ h\ A$
  ⟨*proof*⟩

lemma *related*:
  assumes *Re*: $R\ 1\ 1$
  and *Rop*: $\forall\, x1\ y1\ x2\ y2.\ R\ x1\ x2 \wedge R\ y1\ y2 \longrightarrow R\ (x1 * y1)\ (x2 * y2)$
  and *fS*: *finite S* **and** *Rfg*: $\forall x{\in}S.\ R\ (h\ x)\ (g\ x)$
  shows $R\ (F\ h\ S)\ (F\ g\ S)$
  ⟨*proof*⟩

lemma *eq-general*:
  assumes *fS*: *finite S*
  and *h*: $\forall y{\in}S'.\ \exists!x.\ x \in S \wedge h\ x = y$
  and *f12*: $\forall x{\in}S.\ h\ x \in S' \wedge f2\ (h\ x) = f1\ x$
  shows $F\ f1\ S = F\ f2\ S'$
⟨*proof*⟩

lemma *eq-general-inverses*:
  assumes *fS*: *finite S*
  and *kh*: $\bigwedge y.\ y \in T \Longrightarrow k\ y \in S \wedge h\ (k\ y) = y$
  and *hk*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x \wedge g\ (h\ x) = j\ x$

shows $F\ j\ S = F\ g\ T$

⟨*proof*⟩

end

20.6.5 The image case with fixed function and idempotency

locale *folding-image-simple-idem* = *folding-image-simple* +
assumes *idem*: $x * x = x$

sublocale *folding-image-simple-idem* < *semilattice* ⟨*proof*⟩

context *folding-image-simple-idem*
begin

lemma *in-idem*:
assumes *finite* A **and** $x \in A$
shows $g\ x * F\ A = F\ A$
⟨*proof*⟩

lemma *subset-idem*:
assumes *finite* A **and** $B \subseteq A$
shows $F\ B * F\ A = F\ A$
⟨*proof*⟩

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
assumes *finite* A
shows $F\ (\mathit{insert}\ x\ A) = g\ x * F\ A$
⟨*proof*⟩

lemma *union-idem*:
assumes *finite* A **and** *finite* B
shows $F\ (A \cup B) = F\ A * F\ B$
⟨*proof*⟩

end

20.6.6 The image case with flexible function and idempotency

locale *folding-image-idem* = *folding-image* +
assumes *idem*: $x * x = x$

sublocale *folding-image-idem* < *folding-image-simple-idem op* $*$ *1 g F g* ⟨*proof*⟩

20.6.7 The neutral-less case

locale *folding-one* = *abel-semigroup* +

fixes $F :: 'a\ set \Rightarrow 'a$
assumes *eq-fold*: *finite* $A \Longrightarrow F\ A = fold1\ f\ A$
begin

lemma *singleton* [*simp*]:
$F\ \{x\} = x$
⟨*proof*⟩

lemma *eq-fold'*:
assumes *finite* A **and** $x \notin A$
shows $F\ (insert\ x\ A) = fold\ (op\ *)\ x\ A$
⟨*proof*⟩

lemma *insert* [*simp*]:
assumes *finite* A **and** $x \notin A$ **and** $A \neq \{\}$
shows $F\ (insert\ x\ A) = x * F\ A$
⟨*proof*⟩
thm *fold.commute-comp'* [*of* B b, *simplified expand-fun-eq*, *simplified*]
⟨*proof*⟩

lemma *remove*:
assumes *finite* A **and** $x \in A$
shows $F\ A = (if\ A - \{x\} = \{\}\ then\ x\ else\ x * F\ (A - \{x\}))$
⟨*proof*⟩

lemma *insert-remove*:
assumes *finite* A
shows $F\ (insert\ x\ A) = (if\ A - \{x\} = \{\}\ then\ x\ else\ x * F\ (A - \{x\}))$
⟨*proof*⟩

lemma *union-disjoint*:
assumes *finite* A $A \neq \{\}$ **and** *finite* B $B \neq \{\}$ **and** $A \cap B = \{\}$
shows $F\ (A \cup B) = F\ A * F\ B$
⟨*proof*⟩

lemma *union-inter*:
assumes *finite* A **and** *finite* B **and** $A \cap B \neq \{\}$
shows $F\ (A \cup B) * F\ (A \cap B) = F\ A * F\ B$
⟨*proof*⟩

lemma *closed*:
assumes *finite* A $A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x * y \in \{x, y\}$
shows $F\ A \in A$
⟨*proof*⟩

end

20.6.8 The neutral-less case with idempotency

locale *folding-one-idem* = *folding-one* +
  assumes *idem*: $x * x = x$

sublocale *folding-one-idem* < *semilattice* ⟨*proof*⟩

context *folding-one-idem*
begin

lemma *in-idem*:
  assumes *finite* A **and** $x \in A$
  shows $x * F\ A = F\ A$
⟨*proof*⟩

lemma *subset-idem*:
  assumes *finite* A $B \neq \{\}$ **and** $B \subseteq A$
  shows $F\ B * F\ A = F\ A$
⟨*proof*⟩

lemma *eq-fold-idem'*:
  assumes *finite* A
  shows $F\ (insert\ a\ A) = fold\ (op\ *)\ a\ A$
⟨*proof*⟩

lemma *insert-idem* [*simp*]:
  assumes *finite* A **and** $A \neq \{\}$
  shows $F\ (insert\ x\ A) = x * F\ A$
⟨*proof*⟩

lemma *union-idem*:
  assumes *finite* A $A \neq \{\}$ **and** *finite* B $B \neq \{\}$
  shows $F\ (A \cup B) = F\ A * F\ B$
⟨*proof*⟩

lemma *hom-commute*:
  assumes *hom*: $\bigwedge x\ y.\ h\ (x * y) = h\ x * h\ y$
  and N: *finite* N $N \neq \{\}$ **shows** $h\ (F\ N) = F\ (h\ `\ N)$
⟨*proof*⟩

end

notation *times* (**infixl** $*$ *70*)
notation *Groups.one* (*1*)

20.7 Finite cardinality

This definition, although traditional, is ugly to work with: *card* A == *LEAST* n. *EX* f. $A = \{f\ i \mid i.\ i < n\}$. But now that we have *fold-image*

things are easy:

definition *card* :: *'a set* ⇒ *nat* **where**
card A = (*if finite A then fold-image* (*op* +) (λ*x*. *1*) *0 A else 0*)

interpretation *card*!: *folding-image-simple op* + *0* λ*x*. *1 card* ⟨*proof*⟩

lemma *card-infinite* [*simp*]:
¬ *finite A* ⟹ *card A* = *0*
⟨*proof*⟩

lemma *card-empty*:
card {} = *0*
⟨*proof*⟩

lemma *card-insert-disjoint*:
finite A ==> *x* ∉ *A* ==> *card* (*insert x A*) = *Suc* (*card A*)
⟨*proof*⟩

lemma *card-insert-if*:
finite A ==> *card* (*insert x A*) = (*if x* ∈ *A then card A else Suc* (*card A*))
⟨*proof*⟩

lemma *card-ge-0-finite*:
card A > *0* ⟹ *finite A*
⟨*proof*⟩

lemma *card-0-eq* [*simp*, *no-atp*]:
finite A ⟹ *card A* = *0* ⟷ *A* = {}
⟨*proof*⟩

lemma *finite-UNIV-card-ge-0*:
finite (*UNIV* :: *'a set*) ⟹ *card* (*UNIV* :: *'a set*) > *0*
⟨*proof*⟩

lemma *card-eq-0-iff*:
card A = *0* ⟷ *A* = {} ∨ ¬ *finite A*
⟨*proof*⟩

lemma *card-gt-0-iff*:
0 < *card A* ⟷ *A* ≠ {} ∧ *finite A*
⟨*proof*⟩

lemma *card-Suc-Diff1*: *finite A* ==> *x*: *A* ==> *Suc* (*card* (*A* − {*x*})) = *card A*
⟨*proof*⟩

lemma *card-Diff-singleton*:
finite A ==> *x*: *A* ==> *card* (*A* − {*x*}) = *card A* − *1*
⟨*proof*⟩

lemma *card-Diff-singleton-if*:
finite A ==> card (A−{x}) = (if x : A then card A − 1 else card A)
⟨*proof*⟩

lemma *card-Diff-insert*[*simp*]:
assumes *finite A* **and** *a*:*A* **and** *a* ~: *B*
shows *card*(*A − insert a B*) = *card*(*A − B*) *− 1*
⟨*proof*⟩

lemma *card-insert*: *finite A ==> card (insert x A) = Suc (card (A − {x}))*
⟨*proof*⟩

lemma *card-insert-le*: *finite A ==> card A <= card (insert x A)*
⟨*proof*⟩

lemma *card-mono*:
assumes *finite B* **and** *A* ⊆ *B*
shows *card A* ≤ *card B*
⟨*proof*⟩

lemma *card-seteq*: *finite B ==>* (!!*A*. *A <= B ==> card B <= card A ==> A = B*)
⟨*proof*⟩

lemma *psubset-card-mono*: *finite B ==> A < B ==> card A < card B*
⟨*proof*⟩

lemma *card-Un-Int*: *finite A ==> finite B*
==> card A + card B = card (A Un B) + card (A Int B)
⟨*proof*⟩

lemma *card-Un-disjoint*: *finite A ==> finite B*
==> A Int B = {} ==> card (A Un B) = card A + card B
⟨*proof*⟩

lemma *card-Diff-subset*:
assumes *finite B* **and** *B* ⊆ *A*
shows *card (A − B) = card A − card B*
⟨*proof*⟩

lemma *card-Diff-subset-Int*:
assumes *AB*: *finite* (*A* ∩ *B*) **shows** *card (A − B) = card A − card* (*A* ∩ *B*)
⟨*proof*⟩

lemma *card-Diff1-less*: *finite A ==> x*: *A ==> card (A − {x}) < card A*
⟨*proof*⟩

lemma *card-Diff2-less*:
finite A ==> x: *A ==> y*: *A ==> card (A − {x} − {y}) < card A*

⟨*proof*⟩

lemma *card-Diff1-le*: *finite A* ==> *card* $(A - \{x\})$ <= *card A*
⟨*proof*⟩

lemma *card-psubset*: *finite B* ==> $A \subseteq B$ ==> *card A* < *card B* ==> $A < B$
⟨*proof*⟩

lemma *insert-partition*:
$[\![x \notin F; \forall c1 \in insert\ x\ F.\ \forall c2 \in insert\ x\ F.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}]\!]$
$\Longrightarrow x \cap \bigcup F = \{\}$
⟨*proof*⟩

lemma *finite-psubset-induct*[*consumes 1*, *case-names psubset*]:
assumes *fin*: *finite A*
and *major*: $\bigwedge A.$ *finite A* $\Longrightarrow (\bigwedge B.\ B \subset A \Longrightarrow P\ B) \Longrightarrow P\ A$
shows *P A*
⟨*proof*⟩

main cardinality theorem

lemma *card-partition* [*rule-format*]:
finite C ==>
finite $(\bigcup C)$ -->
$(\forall c \in C.$ *card c* = *k*$)$ -->
$(\forall c1 \in C.\ \forall c2 \in C.\ c1 \neq c2$ --> $c1 \cap c2 = \{\}) -->$
$k * card(C) =$ *card* $(\bigcup C)$
⟨*proof*⟩

lemma *card-eq-UNIV-imp-eq-UNIV*:
assumes *fin*: *finite* (*UNIV* :: *'a set*)
and *card*: *card A* = *card* (*UNIV* :: *'a set*)
shows *A* = (*UNIV* :: *'a set*)
⟨*proof*⟩

The form of a finite set of given cardinality

lemma *card-eq-SucD*:
assumes *card A* = *Suc k*
shows $\exists b\ B.$ *A* = *insert b B* & $b \notin B$ & *card B* = *k* & (*k*=*0* $\longrightarrow$ *B*={})
⟨*proof*⟩

lemma *card-Suc-eq*:
(*card A* = *Suc k*) =
($\exists b\ B.$ *A* = *insert b B* & $b \notin B$ & *card B* = *k* & (*k*=*0* $\longrightarrow$ *B*={}))
⟨*proof*⟩

lemma *finite-fun-UNIVD2*:
assumes *fin*: *finite* (*UNIV* :: (*'a* ⇒ *'b*) *set*)
shows *finite* (*UNIV* :: *'b set*)
⟨*proof*⟩

lemma *card-UNIV-unit*: *card* (*UNIV* :: *unit set*) = *1*
⟨*proof*⟩

20.7.1 Cardinality of image

lemma *card-image-le*: *finite A* ==> *card* (*f* ‘ *A*) <= *card A*
⟨*proof*⟩

lemma *card-image*:
assumes *inj-on f A*
shows *card* (*f* ‘ *A*) = *card A*
⟨*proof*⟩

lemma *bij-betw-same-card*: *bij-betw f A B* ⟹ *card A* = *card B*
⟨*proof*⟩

lemma *endo-inj-surj*: *finite A* ==> *f* ‘ *A* ⊆ *A* ==> *inj-on f A* ==> *f* ‘ *A* = *A*
⟨*proof*⟩

lemma *eq-card-imp-inj-on*:
[| *finite A*; *card*(*f* ‘ *A*) = *card A* |] ==> *inj-on f A*
⟨*proof*⟩

lemma *inj-on-iff-eq-card*:
finite A ==> *inj-on f A* = (*card*(*f* ‘ *A*) = *card A*)
⟨*proof*⟩

lemma *card-inj-on-le*:
[|*inj-on f A*; *f* ‘ *A* ⊆ *B*; *finite B* |] ==> *card A* ≤ *card B*
⟨*proof*⟩

lemma *card-bij-eq*:
[|*inj-on f A*; *f* ‘ *A* ⊆ *B*; *inj-on g B*; *g* ‘ *B* ⊆ *A*;
finite A; *finite B* |] ==> *card A* = *card B*
⟨*proof*⟩

20.7.2 Cardinality of sums

lemma *card-Plus*:
assumes *finite A* **and** *finite B*
shows *card* (*A* <+> *B*) = *card A* + *card B*
⟨*proof*⟩

lemma *card-Plus-conv-if*:
card (*A* <+> *B*) = (*if finite A* ∧ *finite B then card A* + *card B else 0*)
⟨*proof*⟩

20.7.3 Cardinality of the Powerset

lemma *card-Pow*: *finite A ==> card (Pow A) = Suc (Suc 0) ^ card A*
⟨*proof*⟩

Relates to equivalence classes. Based on a theorem of F. KammÃ¼ller.

lemma *dvd-partition*:
finite (Union C) ==>
ALL c : C. k dvd card c ==>
(ALL c1: C. ALL c2: C. c1 ≠ c2 −−> c1 Int c2 = {}) ==>
k dvd card (Union C)
⟨*proof*⟩

20.7.4 Relating injectivity and surjectivity

lemma *finite-surj-inj*: *finite(A) ⟹ A <= f'A ⟹ inj-on f A*
⟨*proof*⟩

lemma *finite-UNIV-surj-inj*: **fixes** *f* :: *'a ⇒ 'a*
shows *finite(UNIV:: 'a set) ⟹ surj f ⟹ inj f*
⟨*proof*⟩

lemma *finite-UNIV-inj-surj*: **fixes** *f* :: *'a ⇒ 'a*
shows *finite(UNIV:: 'a set) ⟹ inj f ⟹ surj f*
⟨*proof*⟩

corollary *infinite-UNIV-nat*[*iff*]: *~finite(UNIV::nat set)*
⟨*proof*⟩

lemma *infinite-UNIV-char-0*[*no-atp*]:
¬ finite (UNIV::'a::semiring-char-0 set)
⟨*proof*⟩

end

21 Relation: Relations

theory *Relation*
imports *Datatype Finite-Set*
begin

21.1 Definitions

definition
*converse :: ('a * 'b) set => ('b * 'a) set*
((-^-1) [1000] 999) **where**
r^-1 == {(y, x). (x, y) : r}

notation (*xsymbols*)
converse ((-$^{-1}$) [*1000*] *999*)

definition
rel-comp :: [('*a* * '*b*) *set*, ('*b* * '*c*) *set*] => ('*a* * '*c*) *set*
(**infixr** *O 75*) **where**
r O s == {(*x*,*z*). *EX y*. (*x*, *y*) : *r* & (*y*, *z*) : *s*}

definition
Image :: [('*a* * '*b*) *set*, '*a set*] => '*b set*
(**infixl** `` *90*) **where**
r `` s == {*y*. *EX x*:*s*. (*x*,*y*):*r*}

definition
Id :: ('*a* * '*a*) *set* **where** — the identity relation
Id == {*p*. *EX x*. *p* = (*x*,*x*)}

definition
Id-on :: '*a set* => ('*a* * '*a*) *set* **where** — diagonal: identity over a set
Id-on A == $\bigcup x{\in}A$. {(*x*,*x*)}

definition
Domain :: ('*a* * '*b*) *set* => '*a set* **where**
Domain r == {*x*. *EX y*. (*x*,*y*):*r*}

definition
Range :: ('*a* * '*b*) *set* => '*b set* **where**
Range r == *Domain*(*r*^−*1*)

definition
Field :: ('*a* * '*a*) *set* => '*a set* **where**
Field r == *Domain r* ∪ *Range r*

definition
refl-on :: ['*a set*, ('*a* * '*a*) *set*] => *bool* **where** — reflexivity over a set
refl-on A r == *r* ⊆ *A* × *A* & (*ALL x*: *A*. (*x*,*x*) : *r*)

abbreviation
refl :: ('*a* * '*a*) *set* => *bool* **where** — reflexivity over a type
refl == *refl-on UNIV*

definition
sym :: ('*a* * '*a*) *set* => *bool* **where** — symmetry predicate
sym r == *ALL x y*. (*x*,*y*): *r* −−> (*y*,*x*): *r*

definition
antisym :: ('*a* * '*a*) *set* => *bool* **where** — antisymmetry predicate
antisym r == *ALL x y*. (*x*,*y*):*r* −−> (*y*,*x*):*r* −−> *x*=*y*

definition
trans :: $('a * 'a)$ *set* => *bool* **where** — transitivity predicate
trans r == *(ALL x y z. (x,y):r --> (y,z):r --> (x,z):r)*

definition
irrefl :: $('a * 'a)$ *set* => *bool* **where**
irrefl r $\equiv \forall x.\ (x,x) \notin r$

definition
total-on :: *'a set* => $('a * 'a)$ *set* => *bool* **where**
total-on A r $\equiv \forall x{\in}A. \forall y{\in}A.\ x{\neq}y \longrightarrow (x,y){\in}r \vee (y,x){\in}r$

abbreviation *total* $\equiv$ *total-on UNIV*

definition
single-valued :: $('a * 'b)$ *set* => *bool* **where**
single-valued r == *ALL x y. (x,y):r --> (ALL z. (x,z):r --> y=z)*

definition
inv-image :: $('b * 'b)$ *set* => $('a$ => $'b)$ => $('a * 'a)$ *set* **where**
inv-image r f == $\{(x,\ y).\ (f\ x,\ f\ y) : r\}$

21.2 The identity relation

lemma *IdI* [*intro*]: $(a,\ a) : Id$
⟨*proof*⟩

lemma *IdE* [*elim*!]: $p : Id$ ==> $(!!x.\ p = (x,\ x)$ ==> $P)$ ==> P
⟨*proof*⟩

lemma *pair-in-Id-conv* [*iff*]: $((a,\ b) : Id) = (a = b)$
⟨*proof*⟩

lemma *refl-Id*: *refl Id*
⟨*proof*⟩

lemma *antisym-Id*: *antisym Id*
— A strange result, since *Id* is also symmetric.
⟨*proof*⟩

lemma *sym-Id*: *sym Id*
⟨*proof*⟩

lemma *trans-Id*: *trans Id*
⟨*proof*⟩

21.3 Diagonal: identity over a set

lemma *Id-on-empty* [*simp*]: *Id-on* $\{\} = \{\}$

⟨*proof*⟩

lemma *Id-on-eqI*: $a = b \Longrightarrow a : A \Longrightarrow (a, b) : \mathit{Id}\text{-}\mathit{on}\ A$
⟨*proof*⟩

lemma *Id-onI* [*intro*!,*no-atp*]: $a : A \Longrightarrow (a, a) : \mathit{Id}\text{-}\mathit{on}\ A$
⟨*proof*⟩

lemma *Id-onE* [*elim*!]:
$c : \mathit{Id}\text{-}\mathit{on}\ A \Longrightarrow (!!x.\ x : A \Longrightarrow c = (x, x) \Longrightarrow P) \Longrightarrow P$
— The general elimination rule.
⟨*proof*⟩

lemma *Id-on-iff*: $((x, y) : \mathit{Id}\text{-}\mathit{on}\ A) = (x = y \ \&\ x : A)$
⟨*proof*⟩

lemma *Id-on-subset-Times*: $\mathit{Id}\text{-}\mathit{on}\ A \subseteq A \times A$
⟨*proof*⟩

21.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a, b) : r \Longrightarrow (b, c) : s \Longrightarrow (a, c) : r\ O\ s$
⟨*proof*⟩

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s \Longrightarrow$
$(!!x\ y\ z.\ xz = (x, z) \Longrightarrow (x, y) : r \Longrightarrow (y, z) : s \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-compEpair*:
$(a, c) : r\ O\ s \Longrightarrow (!!y.\ (a, y) : r \Longrightarrow (y, c) : s \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *R-O-Id* [*simp*]: $R\ O\ \mathit{Id} = R$
⟨*proof*⟩

lemma *Id-O-R* [*simp*]: $\mathit{Id}\ O\ R = R$
⟨*proof*⟩

lemma *rel-comp-empty1*[*simp*]: $\{\}\ O\ R = \{\}$
⟨*proof*⟩

lemma *rel-comp-empty2*[*simp*]: $R\ O\ \{\} = \{\}$
⟨*proof*⟩

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
⟨*proof*⟩

lemma *trans-O-subset*: $\mathit{trans}\ r \Longrightarrow r\ O\ r \subseteq r$

⟨*proof*⟩

lemma *rel-comp-mono*: $r' \subseteq r \Longrightarrow s' \subseteq s \Longrightarrow (r'\ O\ s') \subseteq (r\ O\ s)$
⟨*proof*⟩

lemma *rel-comp-subset-Sigma*:
$r \subseteq A \times B \Longrightarrow s \subseteq B \times C \Longrightarrow (r\ O\ s) \subseteq A \times C$
⟨*proof*⟩

lemma *rel-comp-distrib*[*simp*]: $R\ O\ (S \cup T) = (R\ O\ S) \cup (R\ O\ T)$
⟨*proof*⟩

lemma *rel-comp-distrib2*[*simp*]: $(S \cup T)\ O\ R = (S\ O\ R) \cup (T\ O\ R)$
⟨*proof*⟩

lemma *rel-comp-UNION-distrib*: $s\ O\ UNION\ I\ r = UNION\ I\ (\%i.\ s\ O\ r\ i)$
⟨*proof*⟩

lemma *rel-comp-UNION-distrib2*: $UNION\ I\ r\ O\ s = UNION\ I\ (\%i.\ r\ i\ O\ s)$
⟨*proof*⟩

21.5 Reflexivity

lemma *refl-onI*: $r \subseteq A \times A \Longrightarrow (!!x.\ x : A \Longrightarrow (x, x) : r) \Longrightarrow refl\text{-}on\ A\ r$
⟨*proof*⟩

lemma *refl-onD*: $refl\text{-}on\ A\ r \Longrightarrow a : A \Longrightarrow (a, a) : r$
⟨*proof*⟩

lemma *refl-onD1*: $refl\text{-}on\ A\ r \Longrightarrow (x, y) : r \Longrightarrow x : A$
⟨*proof*⟩

lemma *refl-onD2*: $refl\text{-}on\ A\ r \Longrightarrow (x, y) : r \Longrightarrow y : A$
⟨*proof*⟩

lemma *refl-on-Int*: $refl\text{-}on\ A\ r \Longrightarrow refl\text{-}on\ B\ s \Longrightarrow refl\text{-}on\ (A \cap B)\ (r \cap s)$
⟨*proof*⟩

lemma *refl-on-Un*: $refl\text{-}on\ A\ r \Longrightarrow refl\text{-}on\ B\ s \Longrightarrow refl\text{-}on\ (A \cup B)\ (r \cup s)$
⟨*proof*⟩

lemma *refl-on-INTER*:
$ALL\ x{:}S.\ refl\text{-}on\ (A\ x)\ (r\ x) \Longrightarrow refl\text{-}on\ (INTER\ S\ A)\ (INTER\ S\ r)$
⟨*proof*⟩

lemma *refl-on-UNION*:
$ALL\ x{:}S.\ refl\text{-}on\ (A\ x)\ (r\ x) \Longrightarrow refl\text{-}on\ (UNION\ S\ A)\ (UNION\ S\ r)$
⟨*proof*⟩

lemma *refl-on-empty*[*simp*]: *refl-on* {} {}
⟨*proof*⟩

lemma *refl-on-Id-on*: *refl-on A* (*Id-on A*)
⟨*proof*⟩

21.6 Antisymmetry

lemma *antisymI*:
(!!*x y*. (*x*, *y*) : *r* ==> (*y*, *x*) : *r* ==> *x*=*y*) ==> *antisym r*
⟨*proof*⟩

lemma *antisymD*: *antisym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r* ==> *a* = *b*
⟨*proof*⟩

lemma *antisym-subset*: *r* ⊆ *s* ==> *antisym s* ==> *antisym r*
⟨*proof*⟩

lemma *antisym-empty* [*simp*]: *antisym* {}
⟨*proof*⟩

lemma *antisym-Id-on* [*simp*]: *antisym* (*Id-on A*)
⟨*proof*⟩

21.7 Symmetry

lemma *symI*: (!!*a b*. (*a*, *b*) : *r* ==> (*b*, *a*) : *r*) ==> *sym r*
⟨*proof*⟩

lemma *symD*: *sym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r*
⟨*proof*⟩

lemma *sym-Int*: *sym r* ==> *sym s* ==> *sym* (*r* ∩ *s*)
⟨*proof*⟩

lemma *sym-Un*: *sym r* ==> *sym s* ==> *sym* (*r* ∪ *s*)
⟨*proof*⟩

lemma *sym-INTER*: *ALL x*:*S*. *sym* (*r x*) ==> *sym* (*INTER S r*)
⟨*proof*⟩

lemma *sym-UNION*: *ALL x*:*S*. *sym* (*r x*) ==> *sym* (*UNION S r*)
⟨*proof*⟩

lemma *sym-Id-on* [*simp*]: *sym* (*Id-on A*)
⟨*proof*⟩

21.8 Transitivity

lemma *transI*:

(!!$x\ y\ z.\ (x,\ y) : r \Longrightarrow (y,\ z) : r \Longrightarrow (x,\ z) : r) \Longrightarrow trans\ r$
⟨*proof*⟩

lemma *transD*: $trans\ r \Longrightarrow (a,\ b) : r \Longrightarrow (b,\ c) : r \Longrightarrow (a,\ c) : r$
⟨*proof*⟩

lemma *trans-Int*: $trans\ r \Longrightarrow trans\ s \Longrightarrow trans\ (r \cap s)$
⟨*proof*⟩

lemma *trans-INTER*: $ALL\ x{:}S.\ trans\ (r\ x) \Longrightarrow trans\ (INTER\ S\ r)$
⟨*proof*⟩

lemma *trans-Id-on* [*simp*]: $trans\ (Id\text{-}on\ A)$
⟨*proof*⟩

lemma *trans-diff-Id*: $trans\ r \Longrightarrow antisym\ r \Longrightarrow trans\ (r - Id)$
⟨*proof*⟩

21.9 Irreflexivity

lemma *irrefl-diff-Id*[*simp*]: $irrefl(r - Id)$
⟨*proof*⟩

21.10 Totality

lemma *total-on-empty*[*simp*]: $total\text{-}on\ \{\}\ r$
⟨*proof*⟩

lemma *total-on-diff-Id*[*simp*]: $total\text{-}on\ A\ (r - Id) = total\text{-}on\ A\ r$
⟨*proof*⟩

21.11 Converse

lemma *converse-iff* [*iff*]: $((a,b){:}\ r\hat{}-1) = ((b,a) : r)$
⟨*proof*⟩

lemma *converseI*[*sym*]: $(a,\ b) : r \Longrightarrow (b,\ a) : r\hat{}-1$
⟨*proof*⟩

lemma *converseD*[*sym*]: $(a,b) : r\hat{}-1 \Longrightarrow (b,\ a) : r$
⟨*proof*⟩

lemma *converseE* [*elim*!]:
$yx : r\hat{}-1 \Longrightarrow (!!x\ y.\ yx = (y,\ x) \Longrightarrow (x,\ y) : r \Longrightarrow P) \Longrightarrow P$
— More general than *converseD*, as it "splits" the member of the relation.
⟨*proof*⟩

lemma *converse-converse* [*simp*]: $(r\hat{}-1)\hat{}-1 = r$
⟨*proof*⟩

lemma *converse-rel-comp*: $(r\ O\ s)\hat{}-1 = s\hat{}-1\ O\ r\hat{}-1$
⟨*proof*⟩

lemma *converse-Int*: $(r \cap s)\hat{}-1 = r\hat{}-1 \cap s\hat{}-1$
⟨*proof*⟩

lemma *converse-Un*: $(r \cup s)\hat{}-1 = r\hat{}-1 \cup s\hat{}-1$
⟨*proof*⟩

lemma *converse-INTER*: $(INTER\ S\ r)\hat{}-1 = (INT\ x{:}S.\ (r\ x)\hat{}-1)$
⟨*proof*⟩

lemma *converse-UNION*: $(UNION\ S\ r)\hat{}-1 = (UN\ x{:}S.\ (r\ x)\hat{}-1)$
⟨*proof*⟩

lemma *converse-Id* [*simp*]: $Id\hat{}-1 = Id$
⟨*proof*⟩

lemma *converse-Id-on* [*simp*]: $(Id\text{-}on\ A)\hat{}-1 = Id\text{-}on\ A$
⟨*proof*⟩

lemma *refl-on-converse* [*simp*]: $refl\text{-}on\ A\ (converse\ r) = refl\text{-}on\ A\ r$
⟨*proof*⟩

lemma *sym-converse* [*simp*]: $sym\ (converse\ r) = sym\ r$
⟨*proof*⟩

lemma *antisym-converse* [*simp*]: $antisym\ (converse\ r) = antisym\ r$
⟨*proof*⟩

lemma *trans-converse* [*simp*]: $trans\ (converse\ r) = trans\ r$
⟨*proof*⟩

lemma *sym-conv-converse-eq*: $sym\ r = (r\hat{}-1 = r)$
⟨*proof*⟩

lemma *sym-Un-converse*: $sym\ (r \cup r\hat{}-1)$
⟨*proof*⟩

lemma *sym-Int-converse*: $sym\ (r \cap r\hat{}-1)$
⟨*proof*⟩

lemma *total-on-converse*[*simp*]: $total\text{-}on\ A\ (r\hat{}-1) = total\text{-}on\ A\ r$
⟨*proof*⟩

21.12 Domain

declare *Domain-def* [*no-atp*]

lemma *Domain-iff*: $(a : Domain\ r) = (EX\ y.\ (a,\ y) : r)$
⟨*proof*⟩

lemma *DomainI* [*intro*]: $(a,\ b) : r \Longrightarrow a : Domain\ r$
⟨*proof*⟩

lemma *DomainE* [*elim*!]:
$a : Domain\ r \Longrightarrow (!!y.\ (a,\ y) : r \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Domain-empty* [*simp*]: $Domain\ \{\} = \{\}$
⟨*proof*⟩

lemma *Domain-empty-iff*: $Domain\ r = \{\} \longleftrightarrow r = \{\}$
⟨*proof*⟩

lemma *Domain-insert*: $Domain\ (insert\ (a,\ b)\ r) = insert\ a\ (Domain\ r)$
⟨*proof*⟩

lemma *Domain-Id* [*simp*]: $Domain\ Id = UNIV$
⟨*proof*⟩

lemma *Domain-Id-on* [*simp*]: $Domain\ (Id\text{-}on\ A) = A$
⟨*proof*⟩

lemma *Domain-Un-eq*: $Domain(A \cup B) = Domain(A) \cup Domain(B)$
⟨*proof*⟩

lemma *Domain-Int-subset*: $Domain(A \cap B) \subseteq Domain(A) \cap Domain(B)$
⟨*proof*⟩

lemma *Domain-Diff-subset*: $Domain(A) - Domain(B) \subseteq Domain(A - B)$
⟨*proof*⟩

lemma *Domain-Union*: $Domain\ (Union\ S) = (\bigcup A{\in}S.\ Domain\ A)$
⟨*proof*⟩

lemma *Domain-converse*[*simp*]: $Domain(r\hat{}-1) = Range\ r$
⟨*proof*⟩

lemma *Domain-mono*: $r \subseteq s \Longrightarrow Domain\ r \subseteq Domain\ s$
⟨*proof*⟩

lemma *fst-eq-Domain*: $fst\ `\ R = Domain\ R$
⟨*proof*⟩

lemma *Domain-dprod* [*simp*]: $Domain\ (dprod\ r\ s) = uprod\ (Domain\ r)\ (Domain\ s)$
⟨*proof*⟩

lemma *Domain-dsum* [*simp*]: *Domain* (*dsum r s*) = *usum* (*Domain r*) (*Domain s*)
⟨*proof*⟩

21.13 Range

lemma *Range-iff*: (*a* : *Range r*) = (*EX y*. (*y*, *a*) : *r*)
⟨*proof*⟩

lemma *RangeI* [*intro*]: (*a*, *b*) : *r* ==> *b* : *Range r*
⟨*proof*⟩

lemma *RangeE* [*elim*!]: *b* : *Range r* ==> (!!*x*. (*x*, *b*) : *r* ==> *P*) ==> *P*
⟨*proof*⟩

lemma *Range-empty* [*simp*]: *Range* {} = {}
⟨*proof*⟩

lemma *Range-empty-iff*: *Range r* = {} ⟷ *r* = {}
⟨*proof*⟩

lemma *Range-insert*: *Range* (*insert* (*a*, *b*) *r*) = *insert b* (*Range r*)
⟨*proof*⟩

lemma *Range-Id* [*simp*]: *Range Id* = *UNIV*
⟨*proof*⟩

lemma *Range-Id-on* [*simp*]: *Range* (*Id-on A*) = *A*
⟨*proof*⟩

lemma *Range-Un-eq*: *Range*(*A* ∪ *B*) = *Range*(*A*) ∪ *Range*(*B*)
⟨*proof*⟩

lemma *Range-Int-subset*: *Range*(*A* ∩ *B*) ⊆ *Range*(*A*) ∩ *Range*(*B*)
⟨*proof*⟩

lemma *Range-Diff-subset*: *Range*(*A*) − *Range*(*B*) ⊆ *Range*(*A* − *B*)
⟨*proof*⟩

lemma *Range-Union*: *Range* (*Union S*) = (⋃ *A*∈*S*. *Range A*)
⟨*proof*⟩

lemma *Range-converse*[*simp*]: *Range*(*r*^−*1*) = *Domain r*
⟨*proof*⟩

lemma *snd-eq-Range*: *snd* ‘ *R* = *Range R*
⟨*proof*⟩

21.14 Field

lemma *mono-Field*: $r \subseteq s \Longrightarrow Field\ r \subseteq Field\ s$
⟨*proof*⟩

lemma *Field-empty*[*simp*]: $Field\ \{\} = \{\}$
⟨*proof*⟩

lemma *Field-insert*[*simp*]: $Field\ (insert\ (a,b)\ r) = \{a,b\} \cup Field\ r$
⟨*proof*⟩

lemma *Field-Un*[*simp*]: $Field\ (r \cup s) = Field\ r \cup Field\ s$
⟨*proof*⟩

lemma *Field-Union*[*simp*]: $Field\ (\bigcup R) = \bigcup (Field\ `\ R)$
⟨*proof*⟩

lemma *Field-converse*[*simp*]: $Field(r\hat{}-1) = Field\ r$
⟨*proof*⟩

21.15 Image of a set under a relation

declare *Image-def* [*no-atp*]

lemma *Image-iff*: $(b : r``A) = (EX\ x{:}A.\ (x,\ b) : r)$
⟨*proof*⟩

lemma *Image-singleton*: $r``\{a\} = \{b.\ (a,\ b) : r\}$
⟨*proof*⟩

lemma *Image-singleton-iff* [*iff*]: $(b : r``\{a\}) = ((a,\ b) : r)$
⟨*proof*⟩

lemma *ImageI* [*intro*,*no-atp*]: $(a,\ b) : r ==> a : A ==> b : r``A$
⟨*proof*⟩

lemma *ImageE* [*elim*!]:
$b : r\ ``\ A ==> (!!x.\ (x,\ b) : r ==> x : A ==> P) ==> P$
⟨*proof*⟩

lemma *rev-ImageI*: $a : A ==> (a,\ b) : r ==> b : r\ ``\ A$
— This version's more effective when we already have the required a
⟨*proof*⟩

lemma *Image-empty* [*simp*]: $R``\{\} = \{\}$
⟨*proof*⟩

lemma *Image-Id* [*simp*]: $Id\ ``\ A = A$
⟨*proof*⟩

lemma *Image-Id-on* [*simp*]: *Id-on A `` B = A ∩ B*
⟨*proof*⟩

lemma *Image-Int-subset*: *R `` (A ∩ B) ⊆ R `` A ∩ R `` B*
⟨*proof*⟩

lemma *Image-Int-eq*:
single-valued (converse R) ==> R `` (A ∩ B) = R `` A ∩ R `` B
⟨*proof*⟩

lemma *Image-Un*: *R `` (A ∪ B) = R `` A ∪ R `` B*
⟨*proof*⟩

lemma *Un-Image*: *(R ∪ S) `` A = R `` A ∪ S `` A*
⟨*proof*⟩

lemma *Image-subset*: *r ⊆ A × B ==> r``C ⊆ B*
⟨*proof*⟩

lemma *Image-eq-UN*: *r``B = (⋃y∈ B. r``{y})*
— NOT suitable for rewriting
⟨*proof*⟩

lemma *Image-mono*: *r' ⊆ r ==> A' ⊆ A ==> (r' `` A') ⊆ (r `` A)*
⟨*proof*⟩

lemma *Image-UN*: *(r `` (UNION A B)) = (⋃x∈A. r `` (B x))*
⟨*proof*⟩

lemma *Image-INT-subset*: *(r `` INTER A B) ⊆ (⋂x∈A. r `` (B x))*
⟨*proof*⟩

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
[|single-valued (r⁻¹); A≠{}|] ==> r `` INTER A B = (⋂x∈A. r `` B x)
⟨*proof*⟩

lemma *Image-subset-eq*: *(r``A ⊆ B) = (A ⊆ - ((r^-1) `` (-B)))*
⟨*proof*⟩

21.16 Single valued relations

lemma *single-valuedI*:
ALL x y. (x,y):r --> (ALL z. (x,z):r --> y=z) ==> single-valued r
⟨*proof*⟩

lemma *single-valuedD*:
single-valued r ==> (x, y) : r ==> (x, z) : r ==> y = z
⟨*proof*⟩

lemma *single-valued-rel-comp*:
single-valued r ==> single-valued s ==> single-valued (r O s)
⟨*proof*⟩

lemma *single-valued-subset*:
r ⊆ s ==> single-valued s ==> single-valued r
⟨*proof*⟩

lemma *single-valued-Id* [*simp*]: *single-valued Id*
⟨*proof*⟩

lemma *single-valued-Id-on* [*simp*]: *single-valued (Id-on A)*
⟨*proof*⟩

21.17 Graphs given by *Collect*

lemma *Domain-Collect-split* [*simp*]: *Domain{(x,y). P x y} = {x. EX y. P x y}*
⟨*proof*⟩

lemma *Range-Collect-split* [*simp*]: *Range{(x,y). P x y} = {y. EX x. P x y}*
⟨*proof*⟩

lemma *Image-Collect-split* [*simp*]: *{(x,y). P x y} `` A = {y. EX x:A. P x y}*
⟨*proof*⟩

21.18 Inverse image

lemma *sym-inv-image*: *sym r ==> sym (inv-image r f)*
⟨*proof*⟩

lemma *trans-inv-image*: *trans r ==> trans (inv-image r f)*
⟨*proof*⟩

lemma *in-inv-image*[*simp*]: *((x,y) : inv-image r f) = ((f x, f y) : r)*
⟨*proof*⟩

lemma *converse-inv-image*[*simp*]: *(inv-image R f)^-1 = inv-image (R^-1) f*
⟨*proof*⟩

21.19 Finiteness

lemma *finite-converse* [*iff*]: *finite (r^-1) = finite r*
⟨*proof*⟩

lemma *finite-Domain*: *finite r ==> finite (Domain r)*
⟨*proof*⟩

lemma *finite-Range*: *finite r ==> finite (Range r)*
⟨*proof*⟩

lemma *finite-Field*: *finite r* ==> *finite* (*Field r*)
— A finite relation has a finite field (= *domain* ∪ *range*.
⟨*proof*⟩

21.20 Miscellaneous

Version of *lfp-induct* for binary relations

lemmas *lfp-induct2* =
lfp-induct-set [*of* (*a*, *b*), *split-format* (*complete*)]

Version of *subsetI* for binary relations

lemma *subrelI*: (⋀*x y*. (*x*, *y*) ∈ *r* ⟹ (*x*, *y*) ∈ *s*) ⟹ *r* ⊆ *s*
⟨*proof*⟩

end

22 Predicate: Predicates as relations and enumerations

theory *Predicate*
imports *Inductive Relation*
begin

notation
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*) **and**
top (⊤) **and**
bot (⊥)

22.1 Predicates as (complete) lattices

Handy introduction and elimination rules for ≤ on unary and binary predicates

lemma *predicate1I*:
assumes *PQ*: ⋀*x*. *P x* ⟹ *Q x*
shows *P* ≤ *Q*
⟨*proof*⟩

lemma *predicate1D* [*Pure.dest?*, *dest?*]:
P ≤ *Q* ⟹ *P x* ⟹ *Q x*
⟨*proof*⟩

lemma *rev-predicate1D*:

$P\ x$ ==> P <= Q ==> $Q\ x$
⟨*proof*⟩

lemma *predicate2I* [*Pure.intro*!, *intro*!]:
assumes *PQ*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y$
shows $P \leq Q$
⟨*proof*⟩

lemma *predicate2D* [*Pure.dest*, *dest*]:
$P \leq Q \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
⟨*proof*⟩

lemma *rev-predicate2D*:
$P\ x\ y$ ==> P <= Q ==> $Q\ x\ y$
⟨*proof*⟩

22.1.1 Equality

lemma *pred-equals-eq*: $((\lambda x.\ x \in R) = (\lambda x.\ x \in S)) = (R = S)$
⟨*proof*⟩

lemma *pred-equals-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) = (\lambda x\ y.\ (x,\ y) \in S)) = (R = S)$
⟨*proof*⟩

22.1.2 Order relation

lemma *pred-subset-eq*: $((\lambda x.\ x \in R) <= (\lambda x.\ x \in S)) = (R <= S)$
⟨*proof*⟩

lemma *pred-subset-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) <= (\lambda x\ y.\ (x,\ y) \in S)) = (R <= S)$
⟨*proof*⟩

22.1.3 Top and bottom elements

lemma *top1I* [*intro*!]: *top x*
⟨*proof*⟩

lemma *top2I* [*intro*!]: *top x y*
⟨*proof*⟩

lemma *bot1E* [*elim*!]: *bot x* $\Longrightarrow P$
⟨*proof*⟩

lemma *bot2E* [*elim*!]: *bot x y* $\Longrightarrow P$
⟨*proof*⟩

lemma *bot-empty-eq*: *bot* $= (\lambda x.\ x \in \{\})$
⟨*proof*⟩

lemma *bot-empty-eq2*: $bot = (\lambda x\ y.\ (x,\ y) \in \{\})$
⟨*proof*⟩

22.1.4 Binary union

lemma *sup1I1* [*elim?*]: $A\ x \Longrightarrow sup\ A\ B\ x$
⟨*proof*⟩

lemma *sup2I1* [*elim?*]: $A\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
⟨*proof*⟩

lemma *sup1I2* [*elim?*]: $B\ x \Longrightarrow sup\ A\ B\ x$
⟨*proof*⟩

lemma *sup2I2* [*elim?*]: $B\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
⟨*proof*⟩

lemma *sup1E* [*elim!*]: $sup\ A\ B\ x \Longrightarrow (A\ x \Longrightarrow P) \Longrightarrow (B\ x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *sup2E* [*elim!*]: $sup\ A\ B\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow P) \Longrightarrow (B\ x\ y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI* [*intro!*]: $(\sim B\ x \Longrightarrow A\ x) \Longrightarrow sup\ A\ B\ x$
⟨*proof*⟩

lemma *sup2CI* [*intro!*]: $(\sim B\ x\ y \Longrightarrow A\ x\ y) \Longrightarrow sup\ A\ B\ x\ y$
⟨*proof*⟩

lemma *sup-Un-eq*: $sup\ (\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
⟨*proof*⟩

lemma *sup-Un-eq2* [*pred-set-conv*]: $sup\ (\lambda x\ y.\ (x,\ y) \in R)\ (\lambda x\ y.\ (x,\ y) \in S) = (\lambda x\ y.\ (x,\ y) \in R \cup S)$
⟨*proof*⟩

22.1.5 Binary intersection

lemma *inf1I* [*intro!*]: $A\ x \Longrightarrow B\ x \Longrightarrow inf\ A\ B\ x$
⟨*proof*⟩

lemma *inf2I* [*intro!*]: $A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow inf\ A\ B\ x\ y$
⟨*proof*⟩

lemma *inf1E* [*elim!*]: $inf\ A\ B\ x \Longrightarrow (A\ x \Longrightarrow B\ x \Longrightarrow P) \Longrightarrow P$

⟨*proof*⟩

lemma *inf2E* [*elim*!]: *inf A B x y* ==> (*A x y* ==> *B x y* ==> *P*) ==> *P*
⟨*proof*⟩

lemma *inf1D1*: *inf A B x* ==> *A x*
⟨*proof*⟩

lemma *inf2D1*: *inf A B x y* ==> *A x y*
⟨*proof*⟩

lemma *inf1D2*: *inf A B x* ==> *B x*
⟨*proof*⟩

lemma *inf2D2*: *inf A B x y* ==> *B x y*
⟨*proof*⟩

lemma *inf-Int-eq*: *inf* (λ*x*. *x* ∈ *R*) (λ*x*. *x* ∈ *S*) = (λ*x*. *x* ∈ *R* ∩ *S*)
⟨*proof*⟩

lemma *inf-Int-eq2* [*pred-set-conv*]: *inf* (λ*x y*. (*x*, *y*) ∈ *R*) (λ*x y*. (*x*, *y*) ∈ *S*) = (λ*x y*. (*x*, *y*) ∈ *R* ∩ *S*)
⟨*proof*⟩

22.1.6 Unions of families

lemma *SUP1-iff*: (*SUP x*:*A*. *B x*) *b* = (*EX x*:*A*. *B x b*)
⟨*proof*⟩

lemma *SUP2-iff*: (*SUP x*:*A*. *B x*) *b c* = (*EX x*:*A*. *B x b c*)
⟨*proof*⟩

lemma *SUP1-I* [*intro*]: *a* : *A* ==> *B a b* ==> (*SUP x*:*A*. *B x*) *b*
⟨*proof*⟩

lemma *SUP2-I* [*intro*]: *a* : *A* ==> *B a b c* ==> (*SUP x*:*A*. *B x*) *b c*
⟨*proof*⟩

lemma *SUP1-E* [*elim*!]: (*SUP x*:*A*. *B x*) *b* ==> (!!*x*. *x* : *A* ==> *B x b* ==> *R*) ==> *R*
⟨*proof*⟩

lemma *SUP2-E* [*elim*!]: (*SUP x*:*A*. *B x*) *b c* ==> (!!*x*. *x* : *A* ==> *B x b c* ==> *R*) ==> *R*
⟨*proof*⟩

lemma *SUP-UN-eq*: (*SUP i*. (λ*x*. *x* ∈ *r i*)) = (λ*x*. *x* ∈ (*UN i*. *r i*))
⟨*proof*⟩

lemma *SUP-UN-eq2*: $(SUP\ i.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (UN\ i.\ r\ i))$
⟨*proof*⟩

22.1.7 Intersections of families

lemma *INF1-iff*: $(INF\ x{:}A.\ B\ x)\ b = (ALL\ x{:}A.\ B\ x\ b)$
⟨*proof*⟩

lemma *INF2-iff*: $(INF\ x{:}A.\ B\ x)\ b\ c = (ALL\ x{:}A.\ B\ x\ b\ c)$
⟨*proof*⟩

lemma *INF1-I* [*intro*!]: $(!!x.\ x : A \Longrightarrow B\ x\ b) \Longrightarrow (INF\ x{:}A.\ B\ x)\ b$
⟨*proof*⟩

lemma *INF2-I* [*intro*!]: $(!!x.\ x : A \Longrightarrow B\ x\ b\ c) \Longrightarrow (INF\ x{:}A.\ B\ x)\ b\ c$
⟨*proof*⟩

lemma *INF1-D* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b \Longrightarrow a : A \Longrightarrow B\ a\ b$
⟨*proof*⟩

lemma *INF2-D* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b\ c \Longrightarrow a : A \Longrightarrow B\ a\ b\ c$
⟨*proof*⟩

lemma *INF1-E* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b \Longrightarrow (B\ a\ b \Longrightarrow R) \Longrightarrow (a\ {}^{\sim}{:}\ A \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *INF2-E* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b\ c \Longrightarrow (B\ a\ b\ c \Longrightarrow R) \Longrightarrow (a\ {}^{\sim}{:}\ A \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *INF-INT-eq*: $(INF\ i.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (INT\ i.\ r\ i))$
⟨*proof*⟩

lemma *INF-INT-eq2*: $(INF\ i.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (INT\ i.\ r\ i))$
⟨*proof*⟩

22.2 Predicates as relations

22.2.1 Composition

inductive
pred-comp :: $['a \Rightarrow 'b \Rightarrow bool,\ 'b \Rightarrow 'c \Rightarrow bool] \Rightarrow 'a \Rightarrow 'c \Rightarrow bool$
(**infixr** *OO* 75)
for $r :: 'a \Rightarrow 'b \Rightarrow bool$ **and** $s :: 'b \Rightarrow 'c \Rightarrow bool$
where
pred-compI [*intro*]: $r\ a\ b \Longrightarrow s\ b\ c \Longrightarrow (r\ OO\ s)\ a\ c$

inductive-cases *pred-compE* [*elim*!]: $(r\ OO\ s)\ a\ c$

lemma *pred-comp-rel-comp-eq* [*pred-set-conv*]:
$((\lambda x\ y.\ (x,\ y) \in r)\ OO\ (\lambda x\ y.\ (x,\ y) \in s)) = (\lambda x\ y.\ (x,\ y) \in r\ O\ s)$
⟨*proof*⟩

22.2.2 Converse

inductive
conversep :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b \Rightarrow 'a \Rightarrow bool$
((-ˆ−−1) [1000] 1000)
for $r :: 'a \Rightarrow 'b \Rightarrow bool$
where
conversepI: $r\ a\ b \Longrightarrow r\hat{}{-}{-}1\ b\ a$

notation (*xsymbols*)
conversep ((-$^{-1-1}$) [1000] 1000)

lemma *conversepD*:
assumes *ab*: $r\hat{}{-}{-}1\ a\ b$
shows $r\ b\ a$ ⟨*proof*⟩

lemma *conversep-iff* [*iff*]: $r\hat{}{-}{-}1\ a\ b = r\ b\ a$
⟨*proof*⟩

lemma *conversep-converse-eq* [*pred-set-conv*]:
$(\lambda x\ y.\ (x,\ y) \in r)\hat{}{-}{-}1 = (\lambda x\ y.\ (x,\ y) \in r\hat{}{-}1)$
⟨*proof*⟩

lemma *conversep-conversep* [*simp*]: $(r\hat{}{-}{-}1)\hat{}{-}{-}1 = r$
⟨*proof*⟩

lemma *converse-pred-comp*: $(r\ OO\ s)\hat{}{-}{-}1 = s\hat{}{-}{-}1\ OO\ r\hat{}{-}{-}1$
⟨*proof*⟩

lemma *converse-meet*: $(inf\ r\ s)\hat{}{-}{-}1 = inf\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
⟨*proof*⟩

lemma *converse-join*: $(sup\ r\ s)\hat{}{-}{-}1 = sup\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
⟨*proof*⟩

lemma *conversep-noteq* [*simp*]: $(op\ \tilde{}{=})\hat{}{-}{-}1 = op\ \tilde{}{=}$
⟨*proof*⟩

lemma *conversep-eq* [*simp*]: $(op\ =)\hat{}{-}{-}1 = op\ =$
⟨*proof*⟩

22.2.3 Domain

inductive
DomainP :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$
for $r :: 'a \Rightarrow 'b \Rightarrow bool$

where
DomainPI [*intro*]: *r a b* ==> *DomainP r a*

inductive-cases *DomainPE* [*elim*!]: *DomainP r a*

lemma *DomainP-Domain-eq* [*pred-set-conv*]: *DomainP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Domain r*)
⟨*proof*⟩

22.2.4 Range

inductive
RangeP :: ('*a* => '*b* => *bool*) => '*b* => *bool*
for *r* :: '*a* => '*b* => *bool*
where
RangePI [*intro*]: *r a b* ==> *RangeP r b*

inductive-cases *RangePE* [*elim*!]: *RangeP r b*

lemma *RangeP-Range-eq* [*pred-set-conv*]: *RangeP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Range r*)
⟨*proof*⟩

22.2.5 Inverse image

definition
inv-imagep :: ('*b* => '*b* => *bool*) => ('*a* => '*b*) => '*a* => '*a* => *bool* **where**
inv-imagep r f == %*x y*. *r* (*f x*) (*f y*)

lemma [*pred-set-conv*]: *inv-imagep* (λ*x y*. (*x*, *y*) ∈ *r*) *f* = (λ*x y*. (*x*, *y*) ∈ *inv-image r f*)
⟨*proof*⟩

lemma *in-inv-imagep* [*simp*]: *inv-imagep r f x y* = *r* (*f x*) (*f y*)
⟨*proof*⟩

22.2.6 Powerset

definition *Powp* :: ('*a* ⇒ *bool*) ⇒ '*a set* ⇒ *bool* **where**
Powp A == λ*B*. ∀ *x* ∈ *B*. *A x*

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp* (λ*x*. *x* ∈ *A*) = (λ*x*. *x* ∈ *Pow A*)
⟨*proof*⟩

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred pred-subset-eq*]

22.2.7 Properties of relations

abbreviation *antisymP* :: ('*a* => '*a* => *bool*) => *bool* **where**
antisymP r == *antisym* {(*x*, *y*). *r x y*}

abbreviation *transP* :: (*'a => 'a => bool*) => *bool* **where**
transP r == trans {(x, y). r x y}

abbreviation *single-valuedP* :: (*'a => 'b => bool*) => *bool* **where**
single-valuedP r == single-valued {(x, y). r x y}

22.3 Predicates as enumerations

22.3.1 The type of predicate enumerations (a monad)

datatype *'a pred = Pred 'a ⇒ bool*

primrec *eval* :: *'a pred ⇒ 'a ⇒ bool* **where**
eval-pred: *eval (Pred f) = f*

lemma *Pred-eval* [*simp*]:
Pred (eval x) = x
⟨*proof*⟩

lemma *eval-inject*: *eval x = eval y ⟷ x = y*
⟨*proof*⟩

definition *single* :: *'a ⇒ 'a pred* **where**
single x = Pred ((op =) x)

definition *bind* :: *'a pred ⇒ ('a ⇒ 'b pred) ⇒ 'b pred* (**infixl** *⤜ 70*) **where**
P ⤜ f = Pred (λx. (∃ y. eval P y ∧ eval (f y) x))

instantiation *pred* :: (*type*) {*complete-lattice*, *boolean-algebra*}
begin

definition
P ≤ Q ⟷ eval P ≤ eval Q

definition
P < Q ⟷ eval P < eval Q

definition
⊥ = Pred ⊥

definition
⊤ = Pred ⊤

definition
P ⊓ Q = Pred (eval P ⊓ eval Q)

definition
P ⊔ Q = Pred (eval P ⊔ eval Q)

definition
[*code del*]: $\bigsqcap A = Pred\ (INFI\ A\ eval)$

definition
[*code del*]: $\bigsqcup A = Pred\ (SUPR\ A\ eval)$

definition
$- P = Pred\ (-\ eval\ P)$

definition
$P - Q = Pred\ (eval\ P - eval\ Q)$

instance ⟨*proof*⟩

end

lemma *bind-bind*:
$(P \gg= Q) \gg= R = P \gg= (\lambda x.\ Q\ x \gg= R)$
⟨*proof*⟩

lemma *bind-single*:
$P \gg= single = P$
⟨*proof*⟩

lemma *single-bind*:
$single\ x \gg= P = P\ x$
⟨*proof*⟩

lemma *bottom-bind*:
$\bot \gg= P = \bot$
⟨*proof*⟩

lemma *sup-bind*:
$(P \sqcup Q) \gg= R = P \gg= R \sqcup Q \gg= R$
⟨*proof*⟩

lemma *Sup-bind*: $(\bigsqcup A \gg= f) = \bigsqcup ((\lambda x.\ x \gg= f)\ `\ A)$
⟨*proof*⟩

lemma *pred-iffI*:
assumes $\bigwedge x.\ eval\ A\ x \Longrightarrow eval\ B\ x$
and $\bigwedge x.\ eval\ B\ x \Longrightarrow eval\ A\ x$
shows $A = B$
⟨*proof*⟩

lemma *singleI*: $eval\ (single\ x)\ x$
⟨*proof*⟩

lemma *singleI-unit*: $eval\ (single\ ())\ x$

⟨*proof*⟩

lemma *singleE*: *eval* (*single x*) *y* ⟹ (*y* = *x* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *singleE'*: *eval* (*single x*) *y* ⟹ (*x* = *y* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *bindI*: *eval P x* ⟹ *eval* (*Q x*) *y* ⟹ *eval* (*P* ⤜ *Q*) *y*
⟨*proof*⟩

lemma *bindE*: *eval* (*R* ⤜ *Q*) *y* ⟹ (⋀*x*. *eval R x* ⟹ *eval* (*Q x*) *y* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *botE*: *eval* ⊥ *x* ⟹ *P*
⟨*proof*⟩

lemma *supI1*: *eval A x* ⟹ *eval* (*A* ⊔ *B*) *x*
⟨*proof*⟩

lemma *supI2*: *eval B x* ⟹ *eval* (*A* ⊔ *B*) *x*
⟨*proof*⟩

lemma *supE*: *eval* (*A* ⊔ *B*) *x* ⟹ (*eval A x* ⟹ *P*) ⟹ (*eval B x* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *single-not-bot* [*simp*]:
single x ≠ ⊥
⟨*proof*⟩

lemma *not-bot*:
assumes *A* ≠ ⊥
obtains *x* **where** *eval A x*
⟨*proof*⟩

22.3.2 Emptiness check and definite choice

definition *is-empty* :: *'a pred* ⇒ *bool* **where**
is-empty A ⟷ *A* = ⊥

lemma *is-empty-bot*:
is-empty ⊥
⟨*proof*⟩

lemma *not-is-empty-single*:
¬ *is-empty* (*single x*)
⟨*proof*⟩

lemma *is-empty-sup*:
is-empty $(A \sqcup B) \longleftrightarrow$ *is-empty* $A \wedge$ *is-empty* B
⟨*proof*⟩

definition *singleton* :: $(unit \Rightarrow 'a) \Rightarrow 'a\ pred \Rightarrow 'a$ **where**
singleton dfault A = (*if* $\exists!x.$ *eval* $A\ x$ *then THE* $x.$ *eval* $A\ x$ *else dfault* ())

lemma *singleton-eqI*:
$\exists!x.$ *eval* $A\ x \Longrightarrow$ *eval* $A\ x \Longrightarrow$ *singleton dfault* $A = x$
⟨*proof*⟩

lemma *eval-singletonI*:
$\exists!x.$ *eval* $A\ x \Longrightarrow$ *eval* A (*singleton dfault* A)
⟨*proof*⟩

lemma *single-singleton*:
$\exists!x.$ *eval* $A\ x \Longrightarrow$ *single* (*singleton dfault* A) $= A$
⟨*proof*⟩

lemma *singleton-undefinedI*:
$\neg$ ($\exists!x.$ *eval* $A\ x$) $\Longrightarrow$ *singleton dfault* A = *dfault* ()
⟨*proof*⟩

lemma *singleton-bot*:
singleton dfault $\bot$ = *dfault* ()
⟨*proof*⟩

lemma *singleton-single*:
singleton dfault (*single* x) $= x$
⟨*proof*⟩

lemma *singleton-sup-single-single*:
singleton dfault (*single* $x \sqcup$ *single* y) = (*if* $x = y$ *then* x *else dfault* ())
⟨*proof*⟩

lemma *singleton-sup-aux*:
singleton dfault $(A \sqcup B)$ = (*if* $A = \bot$ *then singleton dfault* B
else if $B = \bot$ *then singleton dfault* A
else singleton dfault
(*single* (*singleton dfault* A) $\sqcup$ *single* (*singleton dfault* B)))
⟨*proof*⟩

lemma *singleton-sup*:
singleton dfault $(A \sqcup B)$ = (*if* $A = \bot$ *then singleton dfault* B
else if $B = \bot$ *then singleton dfault* A
else if singleton dfault A = *singleton dfault* B *then singleton dfault* A *else dfault*
())
⟨*proof*⟩

22.3.3 Derived operations

definition *if-pred* :: *bool* ⇒ *unit pred* **where**
if-pred-eq: *if-pred b* = (*if b then single* () *else* ⊥)

definition *holds* :: *unit pred* ⇒ *bool* **where**
holds-eq: *holds P* = *eval P* ()

definition *not-pred* :: *unit pred* ⇒ *unit pred* **where**
not-pred-eq: *not-pred P* = (*if eval P* () *then* ⊥ *else single* ())

lemma *if-predI*: *P* ⟹ *eval* (*if-pred P*) ()
⟨*proof*⟩

lemma *if-predE*: *eval* (*if-pred b*) *x* ⟹ (*b* ⟹ *x* = () ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *not-predI*: ¬ *P* ⟹ *eval* (*not-pred* (*Pred* (λ*u*. *P*))) ()
⟨*proof*⟩

lemma *not-predI′*: ¬ *eval P* () ⟹ *eval* (*not-pred P*) ()
⟨*proof*⟩

lemma *not-predE*: *eval* (*not-pred* (*Pred* (λ*u*. *P*))) *x* ⟹ (¬ *P* ⟹ *thesis*) ⟹ *thesis*
⟨*proof*⟩

lemma *not-predE′*: *eval* (*not-pred P*) *x* ⟹ (¬ *eval P x* ⟹ *thesis*) ⟹ *thesis*
⟨*proof*⟩
lemma *f* () = *False* ∨ *f* () = *True*
⟨*proof*⟩

lemma *closure-of-bool-cases*:
assumes (*f* :: *unit* ⇒ *bool*) = (%*u*. *False*) ⟹ *P f*
assumes *f* = (%*u*. *True*) ⟹ *P f*
shows *P f*
⟨*proof*⟩

lemma *unit-pred-cases*:
assumes *P* ⊥
assumes *P* (*single* ())
shows *P Q*
⟨*proof*⟩

lemma *holds-if-pred*:
holds (*if-pred b*) = *b*
⟨*proof*⟩

lemma *if-pred-holds*:
if-pred (*holds P*) = *P*

⟨*proof*⟩

lemma *is-empty-holds*:
is-empty P ⟷ ¬ *holds P*
⟨*proof*⟩

22.3.4 Implementation

datatype *'a seq* = *Empty* | *Insert 'a 'a pred* | *Join 'a pred 'a seq*

primrec *pred-of-seq* :: *'a seq* ⇒ *'a pred* **where**
pred-of-seq Empty = ⊥
| *pred-of-seq* (*Insert x P*) = *single x* ⊔ *P*
| *pred-of-seq* (*Join P xq*) = *P* ⊔ *pred-of-seq xq*

definition *Seq* :: (*unit* ⇒ *'a seq*) ⇒ *'a pred* **where**
Seq f = *pred-of-seq* (*f* ())

code-datatype *Seq*

primrec *member* :: *'a seq* ⇒ *'a* ⇒ *bool* **where**
member Empty x ⟷ *False*
| *member* (*Insert y P*) *x* ⟷ *x* = *y* ∨ *eval P x*
| *member* (*Join P xq*) *x* ⟷ *eval P x* ∨ *member xq x*

lemma *eval-member*:
member xq = *eval* (*pred-of-seq xq*)
⟨*proof*⟩

lemma *eval-code* [*code*]: *eval* (*Seq f*) = *member* (*f* ())
⟨*proof*⟩

lemma *single-code* [*code*]:
single x = *Seq* (λ*u*. *Insert x* ⊥)
⟨*proof*⟩

primrec *apply* :: (*'a* ⇒ *'b Predicate.pred*) ⇒ *'a seq* ⇒ *'b seq* **where**
apply f Empty = *Empty*
| *apply f* (*Insert x P*) = *Join* (*f x*) (*Join* (*P* ⤜ *f*) *Empty*)
| *apply f* (*Join P xq*) = *Join* (*P* ⤜ *f*) (*apply f xq*)

lemma *apply-bind*:
pred-of-seq (*apply f xq*) = *pred-of-seq xq* ⤜ *f*
⟨*proof*⟩

lemma *bind-code* [*code*]:
Seq g ⤜ *f* = *Seq* (λ*u*. *apply f* (*g* ()))
⟨*proof*⟩

lemma *bot-set-code* [*code*]:
$\bot = Seq\ (\lambda u.\ Empty)$
⟨*proof*⟩

primrec *adjunct* :: *'a pred* ⇒ *'a seq* ⇒ *'a seq* **where**
adjunct P Empty = *Join P Empty*
| *adjunct P* (*Insert x Q*) = *Insert x* (*Q* ⊔ *P*)
| *adjunct P* (*Join Q xq*) = *Join Q* (*adjunct P xq*)

lemma *adjunct-sup*:
pred-of-seq (*adjunct P xq*) = *P* ⊔ *pred-of-seq xq*
⟨*proof*⟩

lemma *sup-code* [*code*]:
Seq f ⊔ *Seq g* = *Seq* (λ*u*. *case f* ()
of Empty ⇒ *g* ()
| *Insert x P* ⇒ *Insert x* (*P* ⊔ *Seq g*)
| *Join P xq* ⇒ *adjunct* (*Seq g*) (*Join P xq*))
⟨*proof*⟩

primrec *contained* :: *'a seq* ⇒ *'a pred* ⇒ *bool* **where**
contained Empty Q ⟷ *True*
| *contained* (*Insert x P*) *Q* ⟷ *eval Q x* ∧ *P* ≤ *Q*
| *contained* (*Join P xq*) *Q* ⟷ *P* ≤ *Q* ∧ *contained xq Q*

lemma *single-less-eq-eval*:
single x ≤ *P* ⟷ *eval P x*
⟨*proof*⟩

lemma *contained-less-eq*:
contained xq Q ⟷ *pred-of-seq xq* ≤ *Q*
⟨*proof*⟩

lemma *less-eq-pred-code* [*code*]:
Seq f ≤ *Q* = (*case f* ()
of Empty ⇒ *True*
| *Insert x P* ⇒ *eval Q x* ∧ *P* ≤ *Q*
| *Join P xq* ⇒ *P* ≤ *Q* ∧ *contained xq Q*)
⟨*proof*⟩

lemma *eq-pred-code* [*code*]:
fixes *P Q* :: *'a pred*
shows *eq-class.eq P Q* ⟷ *P* ≤ *Q* ∧ *Q* ≤ *P*
⟨*proof*⟩

lemma [*code*]:
pred-case f P = *f* (*eval P*)
⟨*proof*⟩

lemma [*code*]:
pred-rec f P = *f* (*eval P*)
⟨*proof*⟩

inductive *eq* :: *'a* ⇒ *'a* ⇒ *bool* **where** *eq x x*

lemma *eq-is-eq*: *eq x y* ≡ (*x* = *y*)
⟨*proof*⟩

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a pred* ⇒ *'b pred* **where**
map f P = *P* ⤜ (*single* ∘ *f*)

primrec *null* :: *'a seq* ⇒ *bool* **where**
null Empty ⟷ *True*
| *null* (*Insert x P*) ⟷ *False*
| *null* (*Join P xq*) ⟷ *is-empty P* ∧ *null xq*

lemma *null-is-empty*:
null xq ⟷ *is-empty* (*pred-of-seq xq*)
⟨*proof*⟩

lemma *is-empty-code* [*code*]:
is-empty (*Seq f*) ⟷ *null* (*f* ())
⟨*proof*⟩

primrec *the-only* :: (*unit* ⇒ *'a*) ⇒ *'a seq* ⇒ *'a* **where**
[*code del*]: *the-only dfault Empty* = *dfault* ()
| *the-only dfault* (*Insert x P*) = (*if is-empty P then x else let y* = *singleton dfault P in if x* = *y then x else dfault* ())
| *the-only dfault* (*Join P xq*) = (*if is-empty P then the-only dfault xq else if null xq then singleton dfault P*
else let x = *singleton dfault P*; *y* = *the-only dfault xq in*
if x = *y then x else dfault* ())

lemma *the-only-singleton*:
the-only dfault xq = *singleton dfault* (*pred-of-seq xq*)
⟨*proof*⟩

lemma *singleton-code* [*code*]:
singleton dfault (*Seq f*) = (*case f* ()
of Empty ⇒ *dfault* ()
| *Insert x P* ⇒ *if is-empty P then x*
else let y = *singleton dfault P in*
if x = *y then x else dfault* ()
| *Join P xq* ⇒ *if is-empty P then the-only dfault xq*
else if null xq then singleton dfault P
else let x = *singleton dfault P*; *y* = *the-only dfault xq in*
if x = *y then x else dfault* ())
⟨*proof*⟩

definition *not-unique* :: *'a pred* => *'a*
where
  [*code del*]: *not-unique A* = (*THE x*. *eval A x*)

definition *the* :: *'a pred* => *'a*
where
  [*code del*]: *the A* = (*THE x*. *eval A x*)

lemma *the-eq*[*code*]: *the A* = *singleton* (λ*x*. *not-unique A*) *A*
⟨*proof*⟩

code-abort *not-unique*

code-reflect *Predicate*
  datatypes *pred* = *Seq* **and** *seq* = *Empty* | *Insert* | *Join*
  functions *map*

⟨*ML*⟩

no-notation
  inf (**infixl** ⊓ *70*) **and**
  sup (**infixl** ⊔ *65*) **and**
  Inf (⨅ - [*900*] *900*) **and**
  Sup (⨆ - [*900*] *900*) **and**
  top (⊤) **and**
  bot (⊥) **and**
  bind (**infixl** ⤜ *70*)

hide-type (**open**) *pred seq*
hide-const (**open**) *Pred eval single bind is-empty singleton if-pred not-pred holds Empty Insert Join Seq member pred-of-seq apply adjunct null the-only eq map not-unique the*

end

23 Transitive-Closure: Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Predicate*
uses ~~/*src*/*Provers*/*trancl.ML*
begin

rtrancl is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to

be atomic.

inductive-set
rtrancl :: $('a \times 'a)$ *set* $\Rightarrow$ $('a \times 'a)$ *set* ((-^*) [*1000*] *999*)
for r :: $('a \times 'a)$ *set*
where
rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: $(a, a) : r\hat{}*$
| *rtrancl-into-rtrancl* [*Pure.intro*]: $(a, b) : r\hat{}* \Longrightarrow (b, c) : r \Longrightarrow (a, c) : r\hat{}*$

inductive-set
trancl :: $('a \times 'a)$ *set* $\Rightarrow$ $('a \times 'a)$ *set* ((-^+) [*1000*] *999*)
for r :: $('a \times 'a)$ *set*
where
r-into-trancl [*intro*, *Pure.intro*]: $(a, b) : r \Longrightarrow (a, b) : r\hat{}+$
| *trancl-into-trancl* [*Pure.intro*]: $(a, b) : r\hat{}+ \Longrightarrow (b, c) : r \Longrightarrow (a, c) : r\hat{}+$

declare *rtrancl-def* [*nitpick-def del*]
rtranclp-def [*nitpick-def del*]
trancl-def [*nitpick-def del*]
tranclp-def [*nitpick-def del*]

notation
rtranclp ((-^**) [*1000*] *1000*) **and**
tranclp ((-^++) [*1000*] *1000*)

abbreviation
reflclp :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ ((-^==) [*1000*] *1000*)
where
$r\hat{}== == sup\ r\ op =$

abbreviation
reflcl :: $('a \times 'a)$ *set* $\Rightarrow$ $('a \times 'a)$ *set* ((-^=) [*1000*] *999*) **where**
$r\hat{}= == r \cup Id$

notation (*xsymbols*)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-$^{++}$) [*1000*] *1000*) **and**
reflclp ((-$^{==}$) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-$^{+}$) [*1000*] *999*) **and**
reflcl ((-$^{=}$) [*1000*] *999*)

notation (*HTML* **output**)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-$^{++}$) [*1000*] *1000*) **and**
reflclp ((-$^{==}$) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-$^{+}$) [*1000*] *999*) **and**
reflcl ((-$^{=}$) [*1000*] *999*)

23.1 Reflexive closure

lemma *refl-reflcl*[*simp*]: *refl*($r\hat{}=$)
⟨*proof*⟩

lemma *antisym-reflcl*[*simp*]: *antisym*($r\hat{}=$) = *antisym r*
⟨*proof*⟩

lemma *trans-reflclI*[*simp*]: *trans r* ⟹ *trans*($r\hat{}=$)
⟨*proof*⟩

23.2 Reflexive-transitive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: (*sup* (λ*x y*. (*x*, *y*) ∈ *r*) *op* =) = (λ*x y*. (*x*, *y*) ∈ *r* ∪ *Id*)
⟨*proof*⟩

lemma *r-into-rtrancl* [*intro*]: !!*p*. *p* ∈ *r* ==> *p* ∈ $r\hat{}*$
— *rtrancl* of *r* contains *r*
⟨*proof*⟩

lemma *r-into-rtranclp* [*intro*]: *r x y* ==> $r\hat{}**$ *x y*
— *rtrancl* of *r* contains *r*
⟨*proof*⟩

lemma *rtranclp-mono*: *r* ≤ *s* ==> $r\hat{}** \leq s\hat{}**$
— monotonicity of *rtrancl*
⟨*proof*⟩

lemmas *rtrancl-mono* = *rtranclp-mono* [*to-set*]

theorem *rtranclp-induct* [*consumes 1*, *case-names base step*, *induct set*: *rtranclp*]:
assumes *a*: $r\hat{}**$ *a b*
and *cases*: *P a* !!*y z*. [| $r\hat{}**$ *a y*; *r y z*; *P y* |] ==> *P z*
shows *P b* ⟨*proof*⟩

lemmas *rtrancl-induct* [*induct set*: *rtrancl*] = *rtranclp-induct* [*to-set*]

lemmas *rtranclp-induct2* =
rtranclp-induct[*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*,
consumes 1, *case-names refl step*]

lemmas *rtrancl-induct2* =
rtrancl-induct[*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *refl-rtrancl*: *refl* ($r\hat{}*$)
⟨*proof*⟩

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* $(r\hat{}*)$
⟨*proof*⟩

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD*, *standard*]

lemma *rtranclp-trans*:
assumes *xy*: $r\hat{}** x\ y$
and *yz*: $r\hat{}** y\ z$
shows $r\hat{}** x\ z$ ⟨*proof*⟩

lemma *rtranclE* [*cases set*: *rtrancl*]:
assumes *major*: $(a::'a,\ b) : r\hat{}*$
obtains
(*base*) $a = b$
| (*step*) *y* **where** $(a,\ y) : r\hat{}*$ **and** $(y,\ b) : r$
— elimination of *rtrancl* – by induction on a special formula
⟨*proof*⟩

lemma *rtrancl-Int-subset*: $[|\ Id \subseteq s;\ (r\hat{}* \cap s)\ O\ r \subseteq s|] ==> r\hat{}* \subseteq s$
⟨*proof*⟩

lemma *converse-rtranclp-into-rtranclp*:
$r\ a\ b \Longrightarrow r^{**}\ b\ c \Longrightarrow r^{**}\ a\ c$
⟨*proof*⟩

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [*to-set*]

More r^* equations and inclusions.

lemma *rtranclp-idemp* [*simp*]: $(r\hat{}**)\hat{}** = r\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [*to-set*]

lemma *rtrancl-idemp-self-comp* [*simp*]: $R\hat{}*\ O\ R\hat{}* = R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-subset-rtrancl*: $r \subseteq s\hat{}* ==> r\hat{}* \subseteq s\hat{}*$
⟨*proof*⟩

lemma *rtranclp-subset*: $R \leq S ==> S \leq R\hat{}** ==> S\hat{}** = R\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(sup\ (R\hat{}**)\ (S\hat{}**))\hat{}** = (sup\ R\ S)\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflcl* [*simp*]: $(R\hat{}==)\hat{}** = R\hat{}**$
⟨*proof*⟩

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)\hat{}* = r\hat{}*$
⟨*proof*⟩

lemma *rtranclp-r-diff-Id*: $(inf\ r\ op\ \tilde{}=)\hat{}** = r\hat{}**$
⟨*proof*⟩

theorem *rtranclp-converseD*:
assumes *r*: $(r\hat{}--1)\hat{}**\ x\ y$
shows $r\hat{}**\ y\ x$
⟨*proof*⟩

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

theorem *rtranclp-converseI*:
assumes $r\hat{}**\ y\ x$
shows $(r\hat{}--1)\hat{}**\ x\ y$
⟨*proof*⟩

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

lemma *rtrancl-converse*: $(r\hat{}-1)\hat{}* = (r\hat{}*)\hat{}-1$
⟨*proof*⟩

lemma *sym-rtrancl*: *sym r* ==> *sym* $(r\hat{}*)$
⟨*proof*⟩

theorem *converse-rtranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: $r\hat{}**\ a\ b$
and *cases*: $P\ b$!!$y\ z$. [| $r\ y\ z$; $r\hat{}**\ z\ b$; $P\ z$ |] ==> $P\ y$
shows $P\ a$
⟨*proof*⟩

lemmas *converse-rtrancl-induct* = *converse-rtranclp-induct* [*to-set*]

lemmas *converse-rtranclp-induct2* =
converse-rtranclp-induct [*of* - (ax,ay) (bx,by), *split-rule*,
consumes 1, *case-names refl step*]

lemmas *converse-rtrancl-induct2* =
converse-rtrancl-induct [*of* (ax,ay) (bx,by), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *converse-rtranclpE* [*consumes 1*, *case-names base step*]:
assumes *major*: $r\hat{}**\ x\ z$

and *cases*: $x{=}z \Longrightarrow P$
$!!y.\ [|\ r\ x\ y;\ r\hat{}{*}{*}\ y\ z\ |] \Longrightarrow P$
shows P
⟨*proof*⟩

lemmas *converse-rtranclE* = *converse-rtranclpE* [*to-set*]

lemmas *converse-rtranclpE2* = *converse-rtranclpE* [*of* - (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemmas *converse-rtranclE2* = *converse-rtranclE* [*of* (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemma *r-comp-rtrancl-eq*: $r\ O\ r\hat{}{*} = r\hat{}{*}\ O\ r$
⟨*proof*⟩

lemma *rtrancl-unfold*: $r\hat{}{*} = Id\ Un\ r\hat{}{*}\ O\ r$
⟨*proof*⟩

lemma *rtrancl-Un-separatorE*:
$(a,b) : (P \cup Q)\hat{}{*} \Longrightarrow \forall x\ y.\ (a,x) : P\hat{}{*} \longrightarrow (x,y) : Q \longrightarrow x{=}y \Longrightarrow (a,b) : P\hat{}{*}$
⟨*proof*⟩

lemma *rtrancl-Un-separator-converseE*:
$(a,b) : (P \cup Q)\hat{}{*} \Longrightarrow \forall x\ y.\ (x,b) : P\hat{}{*} \longrightarrow (y,x) : Q \longrightarrow y{=}x \Longrightarrow (a,b) : P\hat{}{*}$
⟨*proof*⟩

lemma *Image-closed-trancl*:
assumes $r\ ``\ X \subseteq X$ **shows** $r^*\ ``\ X = X$
⟨*proof*⟩

23.3 Transitive closure

lemma *trancl-mono*: $!!p.\ p \in r\hat{}{+} \Longrightarrow r \subseteq s \Longrightarrow p \in s\hat{}{+}$
⟨*proof*⟩

lemma *r-into-trancl'*: $!!p.\ p : r \Longrightarrow p : r\hat{}{+}$
⟨*proof*⟩

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: $r\hat{}{+}{+}\ a\ b \Longrightarrow r\hat{}{*}{*}\ a\ b$
⟨*proof*⟩

lemmas *trancl-into-rtrancl* = *tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*: **assumes** *r*: $r\hat{}{*}{*}\ a\ b$
shows $!!c.\ r\ b\ c \Longrightarrow r\hat{}{+}{+}\ a\ c$ ⟨*proof*⟩

lemmas *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*: $[|\ r\ a\ b;\ r\hat{}{*}{*}\ b\ c\ |] \Longrightarrow r\hat{}{+}{+}\ a\ c$

— intro rule from r and *rtrancl*
⟨*proof*⟩

lemmas *rtrancl-into-trancl2* = *rtranclp-into-tranclp2* [*to-set*]

Nice induction rule for *trancl*

lemma *tranclp-induct* [*consumes 1*, *case-names base step*, *induct pred*: *tranclp*]:
assumes *a*: $r\hat{}{+}{+}\ a\ b$
and *cases*: $!!y.\ r\ a\ y \Longrightarrow P\ y$
$!!y\ z.\ r\hat{}{+}{+}\ a\ y \Longrightarrow r\ y\ z \Longrightarrow P\ y \Longrightarrow P\ z$
shows $P\ b$ ⟨*proof*⟩

lemmas *trancl-induct* [*induct set*: *trancl*] = *tranclp-induct* [*to-set*]

lemmas *tranclp-induct2* =
tranclp-induct [*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*,
consumes 1, *case-names base step*]

lemmas *trancl-induct2* =
trancl-induct [*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names base step*]

lemma *tranclp-trans-induct*:
assumes *major*: $r\hat{}{+}{+}\ x\ y$
and *cases*: $!!x\ y.\ r\ x\ y \Longrightarrow P\ x\ y$
$!!x\ y\ z.\ [|\ r\hat{}{+}{+}\ x\ y;\ P\ x\ y;\ r\hat{}{+}{+}\ y\ z;\ P\ y\ z\ |] \Longrightarrow P\ x\ z$
shows $P\ x\ y$
— Another induction rule for trancl, incorporating transitivity
⟨*proof*⟩

lemmas *trancl-trans-induct* = *tranclp-trans-induct* [*to-set*]

lemma *tranclE* [*cases set*: *trancl*]:
assumes $(a,\ b) : r\hat{}{+}$
obtains
(*base*) $(a,\ b) : r$
| (*step*) c **where** $(a,\ c) : r\hat{}{+}$ **and** $(c,\ b) : r$
⟨*proof*⟩

lemma *trancl-Int-subset*: $[|\ r \subseteq s;\ (r\hat{}{+} \cap s)\ O\ r \subseteq s|] \Longrightarrow r\hat{}{+} \subseteq s$
⟨*proof*⟩

lemma *trancl-unfold*: $r\hat{}{+} = r\ Un\ r\hat{}{+}\ O\ r$
⟨*proof*⟩

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* $(r\hat{}{+})$
⟨*proof*⟩

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*, *standard*]

lemma *tranclp-trans*:
assumes *xy*: $r\hat{}{+}{+}\ x\ y$
and *yz*: $r\hat{}{+}{+}\ y\ z$
shows $r\hat{}{+}{+}\ x\ z$ ⟨*proof*⟩

lemma *trancl-id* [*simp*]: *trans* $r \Longrightarrow r\hat{}{+} = r$
⟨*proof*⟩

lemma *rtranclp-tranclp-tranclp*:
assumes $r\hat{}{*}{*}\ x\ y$
shows $!!z.\ r\hat{}{+}{+}\ y\ z \Longrightarrow r\hat{}{+}{+}\ x\ z$ ⟨*proof*⟩

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: $r\ a\ b \Longrightarrow r\hat{}{+}{+}\ b\ c \Longrightarrow r\hat{}{+}{+}\ a\ c$
⟨*proof*⟩

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-insert*:
$(insert\ (y,\ x)\ r)\hat{}{+} = r\hat{}{+} \cup \{(a,\ b).\ (a,\ y) \in r\hat{}{*} \wedge (x,\ b) \in r\hat{}{*}\}$
— primitive recursion for *trancl* over finite relations
⟨*proof*⟩

lemma *tranclp-converseI*: $(r\hat{}{+}{+})\hat{}{-}{-}1\ x\ y \Longrightarrow (r\hat{}{-}{-}1)\hat{}{+}{+}\ x\ y$
⟨*proof*⟩

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*: $(r\hat{}{-}{-}1)\hat{}{+}{+}\ x\ y \Longrightarrow (r\hat{}{+}{+})\hat{}{-}{-}1\ x\ y$
⟨*proof*⟩

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

lemma *tranclp-converse*: $(r\hat{}{-}{-}1)\hat{}{+}{+} = (r\hat{}{+}{+})\hat{}{-}{-}1$
⟨*proof*⟩

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: *sym* $r \Longrightarrow$ *sym* $(r\hat{}{+})$
⟨*proof*⟩

lemma *converse-tranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: $r\hat{}{+}{+}\ a\ b$
and *cases*: $!!y.\ r\ y\ b \Longrightarrow P(y)$
$!!y\ z.[|\ r\ y\ z;\ \ r\hat{}{+}{+}\ z\ b;\ \ P(z)\ |] \Longrightarrow P(y)$
shows $P\ a$

⟨*proof*⟩

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

lemma *tranclpD*: $R\hat{}{+}{+}\ x\ y \Longrightarrow EX\ z.\ R\ x\ z \wedge R\hat{}{*}{*}\ z\ y$
⟨*proof*⟩

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *converse-tranclpE*:
assumes *major*: *tranclp r x z*
assumes *base*: $r\ x\ z \Longrightarrow P$
assumes *step*: $\bigwedge y.\ [\!|\ r\ x\ y;\ tranclp\ r\ y\ z\ |\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemmas *converse-tranclE* = *converse-tranclpE* [*to-set*]

lemma *tranclD2*:
$(x,\ y) \in R^+ \Longrightarrow \exists z.\ (x,\ z) \in R^* \wedge (z,\ y) \in R$
⟨*proof*⟩

lemma *irrefl-tranclI*: $r\hat{}{-}1 \cap r\hat{}{*} = \{\} \Longrightarrow (x,\ x) \notin r\hat{}{+}$
⟨*proof*⟩

lemma *irrefl-trancl-rD*: $!!X.\ ALL\ x.\ (x,\ x) \notin r\hat{}{+} \Longrightarrow (x,\ y) \in r \Longrightarrow x \neq y$
⟨*proof*⟩

lemma *trancl-subset-Sigma-aux*:
$(a,\ b) \in r\hat{}{*} \Longrightarrow r \subseteq A \times A \Longrightarrow a = b \vee a \in A$
⟨*proof*⟩

lemma *trancl-subset-Sigma*: $r \subseteq A \times A \Longrightarrow r\hat{}{+} \subseteq A \times A$
⟨*proof*⟩

lemma *reflcl-tranclp* [*simp*]: $(r\hat{}{+}{+})\hat{}{=}{=} = r\hat{}{*}{*}$
⟨*proof*⟩

lemmas *reflcl-trancl* [*simp*] = *reflcl-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r\hat{}{=})\hat{}{+} = r\hat{}{*}$
⟨*proof*⟩

lemma *trancl-empty* [*simp*]: $\{\}\hat{}{+} = \{\}$
⟨*proof*⟩

lemma *rtrancl-empty* [*simp*]: $\{\}\hat{}{*} = Id$
⟨*proof*⟩

lemma *rtranclpD*: $R\hat{}** a\ b ==> a = b \vee a \neq b \wedge R\hat{}++ a\ b$
⟨*proof*⟩

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*:
$(x,y) \in R^* = (x{=}y \vee x{\neq}y \wedge (x,y) \in R^+)$
⟨*proof*⟩

lemma *trancl-unfold-right*: $r\hat{}+ = r\hat{}* \ O\ r$
⟨*proof*⟩

lemma *trancl-unfold-left*: $r\hat{}+ = r\ O\ r\hat{}*$
⟨*proof*⟩

Simplifying nested closures

lemma *rtrancl-trancl-absorb*[*simp*]: $(R\hat{}*)\hat{}+ = R\hat{}*$
⟨*proof*⟩

lemma *trancl-rtrancl-absorb*[*simp*]: $(R\hat{}+)\hat{}* = R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-reflcl-absorb*[*simp*]: $(R\hat{}*)\hat{}= = R\hat{}*$
⟨*proof*⟩

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* $(R\hat{}*) =$ *UNIV*
⟨*proof*⟩

lemma *Range-rtrancl* [*simp*]: *Range* $(R\hat{}*) =$ *UNIV*
⟨*proof*⟩

lemma *rtrancl-Un-subset*: $(R\hat{}* \cup S\hat{}*) \subseteq (R\ Un\ S)\hat{}*$
⟨*proof*⟩

lemma *in-rtrancl-UnI*: $x \in R\hat{}* \vee x \in S\hat{}* ==> x \in (R \cup S)\hat{}*$
⟨*proof*⟩

lemma *trancl-domain* [*simp*]: *Domain* $(r\hat{}+) =$ *Domain r*
⟨*proof*⟩

lemma *trancl-range* [*simp*]: *Range* $(r\hat{}+) =$ *Range r*
⟨*proof*⟩

lemma *Not-Domain-rtrancl*:
$x \sim$: *Domain R* $==> ((x,\ y) : R\hat{}*) = (x = y)$
⟨*proof*⟩

lemma *trancl-subset-Field2*: $r\hat{}+ <= Field\ r \times Field\ r$

⟨*proof*⟩

lemma *finite-trancl*: *finite* (*r*^+) = *finite r*
⟨*proof*⟩

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
⟦ *single-valued r*; (*x*,*y*) ∈ *r*^*; (*x*,*z*) ∈ *r*^* ⟧
⟹ (*y*,*z*) ∈ *r*^* ∨ (*z*,*y*) ∈ *r*^*
⟨*proof*⟩

lemma *r-r-into-trancl*: (*a*, *b*) ∈ *R* ==> (*b*, *c*) ∈ *R* ==> (*a*, *c*) ∈ *R*^+
⟨*proof*⟩

lemma *trancl-into-trancl* [*rule-format*]:
(*a*, *b*) ∈ *r*^+ ==> (*b*, *c*) ∈ *r* −−> (*a*,*c*) ∈ *r*^+
⟨*proof*⟩

lemma *tranclp-rtranclp-tranclp*:
r^++ *a b* ==> *r*^** *b c* ==> *r*^++ *a c*
⟨*proof*⟩

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

23.4 The power operation on relations

R ^^ *n* = *R O ... O R*, the n-fold composition of *R*

overloading
relpow == *compow* :: *nat* ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set*
begin

primrec *relpow* :: *nat* ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* **where**
relpow 0 R = *Id*
| *relpow* (*Suc n*) *R* = (*R* ^^ *n*) *O R*

end

lemma *rel-pow-1* [*simp*]:
fixes $R :: ('a \times 'a)\ set$
shows $R\ \hat{}\hat{}\ 1 = R$
⟨*proof*⟩

lemma *rel-pow-0-I*:
$(x,\ x) \in R\ \hat{}\hat{}\ 0$
⟨*proof*⟩

lemma *rel-pow-Suc-I*:
$(x,\ y) \in R\ \hat{}\hat{}\ n \Longrightarrow (y,\ z) \in R \Longrightarrow (x,\ z) \in R\ \hat{}\hat{}\ Suc\ n$
⟨*proof*⟩

lemma *rel-pow-Suc-I2*:
$(x,\ y) \in R \Longrightarrow (y,\ z) \in R\ \hat{}\hat{}\ n \Longrightarrow (x,\ z) \in R\ \hat{}\hat{}\ Suc\ n$
⟨*proof*⟩

lemma *rel-pow-0-E*:
$(x,\ y) \in R\ \hat{}\hat{}\ 0 \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-Suc-E*:
$(x,\ z) \in R\ \hat{}\hat{}\ Suc\ n \Longrightarrow (\bigwedge y.\ (x,\ y) \in R\ \hat{}\hat{}\ n \Longrightarrow (y,\ z) \in R \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-E*:
$(x,\ z) \in R\ \hat{}\hat{}\ n \Longrightarrow\ (n = 0 \Longrightarrow x = z \Longrightarrow P)$
$\Longrightarrow (\bigwedge y\ m.\ n = Suc\ m \Longrightarrow (x,\ y) \in R\ \hat{}\hat{}\ m \Longrightarrow (y,\ z) \in R \Longrightarrow P)$
$\Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-Suc-D2*:
$(x,\ z) \in R\ \hat{}\hat{}\ Suc\ n \Longrightarrow (\exists\, y.\ (x,\ y) \in R \wedge (y,\ z) \in R\ \hat{}\hat{}\ n)$
⟨*proof*⟩

lemma *rel-pow-Suc-E2*:
$(x,\ z) \in R\ \hat{}\hat{}\ Suc\ n \Longrightarrow (\bigwedge y.\ (x,\ y) \in R \Longrightarrow (y,\ z) \in R\ \hat{}\hat{}\ n \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rel-pow-Suc-D2'*:
$\forall x\ y\ z.\ (x,\ y) \in R\ \hat{}\hat{}\ n \wedge (y,\ z) \in R \longrightarrow (\exists\, w.\ (x,\ w) \in R \wedge (w,\ z) \in R\ \hat{}\hat{}\ n)$
⟨*proof*⟩

lemma *rel-pow-E2*:
$(x,\ z) \in R\ \hat{}\hat{}\ n \Longrightarrow\ (n = 0 \Longrightarrow x = z \Longrightarrow P)$
$\Longrightarrow (\bigwedge y\ m.\ n = Suc\ m \Longrightarrow (x,\ y) \in R \Longrightarrow (y,\ z) \in R\ \hat{}\hat{}\ m \Longrightarrow P)$
$\Longrightarrow P$

⟨*proof*⟩

lemma *rel-pow-add*: $R \mathbin{\hat{}\hat{}} (m+n) = R \mathbin{\hat{}\hat{}} m \ O \ R \mathbin{\hat{}\hat{}} n$
⟨*proof*⟩

lemma *rel-pow-commute*: $R \ O \ R \mathbin{\hat{}\hat{}} n = R \mathbin{\hat{}\hat{}} n \ O \ R$
⟨*proof*⟩

lemma *rtrancl-imp-UN-rel-pow*:
assumes $p \in R\hat{}*$
shows $p \in (\bigcup n.\ R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *rel-pow-imp-rtrancl*:
assumes $p \in R \mathbin{\hat{}\hat{}} n$
shows $p \in R\hat{}*$
⟨*proof*⟩

lemma *rtrancl-is-UN-rel-pow*:
$R\hat{}* = (\bigcup n.\ R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *rtrancl-power*:
$p \in R\hat{}* \longleftrightarrow (\exists n.\ p \in R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *trancl-power*:
$p \in R\hat{}+ \longleftrightarrow (\exists n > 0.\ p \in R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

lemma *rtrancl-imp-rel-pow*:
$p \in R\hat{}* \Longrightarrow \exists n.\ p \in R \mathbin{\hat{}\hat{}} n$
⟨*proof*⟩

lemma *single-valued-rel-pow*:
fixes $R :: ('a * 'a)\ set$
shows *single-valued* $R \Longrightarrow$ *single-valued* $(R \mathbin{\hat{}\hat{}} n)$
⟨*proof*⟩

23.5 Setup of transitivity reasoner

⟨*ML*⟩

Optional methods.

⟨*ML*⟩

end

24 Wellfounded: Well-founded Recursion

theory *Wellfounded*
imports *Transitive-Closure*
uses (*Tools/Function/size.ML*)
begin

24.1 Basic Definitions

definition *wf* :: ('*a* ∗ '*a*) *set* => *bool* **where**
wf(*r*) == (!*P*. (!*x*. (!*y*. (*y*,*x*):*r* --> *P*(*y*)) --> *P*(*x*)) --> (!*x*. *P*(*x*)))

definition *wfP* :: ('*a* => '*a* => *bool*) => *bool* **where**
wfP r == *wf* {(*x*, *y*). *r x y*}

lemma *wfP-wf-eq* [*pred-set-conv*]: *wfP* (λ*x y*. (*x*, *y*) ∈ *r*) = *wf r*
⟨*proof*⟩

lemma *wfUNIVI*:
(!!*P x*. (*ALL x*. (*ALL y*. (*y*,*x*) : *r* --> *P*(*y*)) --> *P*(*x*)) ==> *P*(*x*)) ==> *wf*(*r*)
⟨*proof*⟩

lemmas *wfPUNIVI* = *wfUNIVI* [*to-pred*]

Restriction to domain *A* and range *B*. If *r* is well-founded over their intersection, then *wf r*

lemma *wfI*:
[| *r* ⊆ *A* <∗> *B*;
!!*x P*. [|∀ *x*. (∀ *y*. (*y*,*x*) : *r* --> *P y*) --> *P x*; *x* : *A*; *x* : *B* |] ==> *P x* |]
==> *wf r*
⟨*proof*⟩

lemma *wf-induct*:
[| *wf*(*r*);
!!*x*.[| *ALL y*. (*y*,*x*): *r* --> *P*(*y*) |] ==> *P*(*x*)
|] ==> *P*(*a*)
⟨*proof*⟩

lemmas *wfP-induct* = *wf-induct* [*to-pred*]

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *consumes 1*, *case-names less*, *induct set*: *wf*]

lemmas *wfP-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set*: *wfP*]

lemma *wf-not-sym*: *wf r* ==> (*a*, *x*) : *r* ==> (*x*, *a*) ~: *r*
⟨*proof*⟩

lemma *wf-asym*:
assumes *wf r* $(a, x) \in r$
obtains $(x, a) \notin r$
⟨*proof*⟩

lemma *wf-not-refl* [*simp*]: *wf r* ==> $(a, a) \sim: r$
⟨*proof*⟩

lemma *wf-irrefl*: **assumes** *wf r* **obtains** $(a, a) \notin r$
⟨*proof*⟩

lemma *wf-wellorderI*:
assumes *wf*: *wf* $\{(x::'a::ord, y).\ x < y\}$
assumes *lin*: *OFCLASS*('a::*ord*, *linorder-class*)
shows *OFCLASS*('a::*ord*, *wellorder-class*)
⟨*proof*⟩

lemma (**in** *wellorder*) *wf*:
wf $\{(x, y).\ x < y\}$
⟨*proof*⟩

24.2 Basic Results

Point-free characterization of well-foundedness

lemma *wfE-pf*:
assumes *wf*: *wf R*
assumes *a*: $A \subseteq R$ `` A
shows $A = \{\}$
⟨*proof*⟩

lemma *wfI-pf*:
assumes *a*: $\bigwedge A.\ A \subseteq R$ `` $A \Longrightarrow A = \{\}$
shows *wf R*
⟨*proof*⟩

Minimal-element characterization of well-foundedness

lemma *wfE-min*:
assumes *wf*: *wf R* **and** *Q*: $x \in Q$
obtains *z* **where** $z \in Q$ $\bigwedge y.\ (y, z) \in R \Longrightarrow y \notin Q$
⟨*proof*⟩

lemma *wfI-min*:
assumes *a*: $\bigwedge x\ Q.\ x \in Q \Longrightarrow \exists\, z{\in}Q.\ \forall\, y.\ (y, z) \in R \longrightarrow y \notin Q$
shows *wf R*
⟨*proof*⟩

lemma *wf-eq-minimal*: *wf r* = $(\forall\, Q\ x.\ x{\in}Q$ −−> $(\exists\, z{\in}Q.\ \forall\, y.\ (y,z){\in}r$ −−> $y{\notin}Q))$
⟨*proof*⟩

lemmas *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

Well-foundedness of transitive closure

lemma *wf-trancl*:
assumes *wf r*
shows *wf* (*r*^+)
⟨*proof*⟩

lemmas *wfP-trancl* = *wf-trancl* [*to-pred*]

lemma *wf-converse-trancl*: *wf* (*r*^−1) ==> *wf* ((*r*^+)^−1)
⟨*proof*⟩

Well-foundedness of subsets

lemma *wf-subset*: [| *wf*(*r*); *p*<=*r* |] ==> *wf*(*p*)
⟨*proof*⟩

lemmas *wfP-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

lemma *wf-empty* [*iff*]: *wf* {}
⟨*proof*⟩

lemma *wfP-empty* [*iff*]:
wfP (λ*x y*. *False*)
⟨*proof*⟩

lemma *wf-Int1*: *wf r* ==> *wf* (*r Int r'*)
⟨*proof*⟩

lemma *wf-Int2*: *wf r* ==> *wf* (*r' Int r*)
⟨*proof*⟩

Exponentiation

lemma *wf-exp*:
assumes *wf* (*R* ^^ *n*)
shows *wf R*
⟨*proof*⟩

Well-foundedness of insert

lemma *wf-insert* [*iff*]: *wf*(*insert* (*y*,*x*) *r*) = (*wf*(*r*) & (*x*,*y*) ~: *r*^*)
⟨*proof*⟩

Well-foundedness of image

lemma *wf-prod-fun-image*: [| *wf r*; *inj f* |] ==> *wf*(*prod-fun f f* ‘ *r*)
⟨*proof*⟩

24.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
assumes *wf R wf S*
assumes $R\ O\ S \subseteq R$
shows *wf* $(R \cup S)$
⟨*proof*⟩

Well-foundedness of indexed union with disjoint domains and ranges

lemma *wf-UN*: [| *ALL i:I. wf*(*r i*);
ALL i:I. ALL j:I. r i ~= *r j* −−> *Domain*(*r i*) *Int Range*(*r j*) = {}
|] ==> *wf*(*UN i:I. r i*)
⟨*proof*⟩

lemma *wfP-SUP*:
$\forall i.\ wfP\ (r\ i) \Longrightarrow \forall i\ j.\ r\ i \neq r\ j \longrightarrow inf\ (DomainP\ (r\ i))\ (RangeP\ (r\ j)) = bot$
$\Longrightarrow wfP\ (SUPR\ UNIV\ r)$
⟨*proof*⟩

lemma *wf-Union*:
[| *ALL r:R. wf r*;
ALL r:R. ALL s:R. r ~= *s* −−> *Domain r Int Range s* = {}
|] ==> *wf*(*Union R*)
⟨*proof*⟩

lemma *wf-Un*:
[| *wf r*; *wf s*; *Domain r Int Range s* = {} |] ==> *wf*(*r Un s*)
⟨*proof*⟩

lemma *wf-union-merge*:
$wf\ (R \cup S) = wf\ (R\ O\ R \cup S\ O\ R \cup S)$ (**is** *wf ?A* = *wf ?B*)
⟨*proof*⟩

lemma *wf-comp-self*: $wf\ R = wf\ (R\ O\ R)$ — special case
⟨*proof*⟩

24.4 Acyclic relations

definition *acyclic* :: $('a * 'a)\ set \Rightarrow bool$ **where**
acyclic r == !*x*. (*x*,*x*) ~: *r*^+

abbreviation *acyclicP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
acyclicP r == *acyclic* {(*x*, *y*). *r x y*}

lemma *acyclicI*: *ALL x*. (*x*, *x*) ~: *r*^+ ==> *acyclic r*
⟨*proof*⟩

lemma *wf-acyclic*: *wf r* ==> *acyclic r*
⟨*proof*⟩

lemmas *wfP-acyclicP* = *wf-acyclic* [*to-pred*]

lemma *acyclic-insert* [*iff*]:
  acyclic(*insert* (*y*,*x*) *r*) = (*acyclic r* & (*x*,*y*) ~: *r*^*)
⟨*proof*⟩

lemma *acyclic-converse* [*iff*]: *acyclic*(*r*^−1) = *acyclic r*
⟨*proof*⟩

lemmas *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

lemma *acyclic-impl-antisym-rtrancl*: *acyclic r* ==> *antisym*(*r*^*)
⟨*proof*⟩

lemma *acyclic-subset*: [| *acyclic s*; *r* <= *s* |] ==> *acyclic r*
⟨*proof*⟩

Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf* [*rule-format*]: *finite r* ==> *acyclic r* −−> *wf r*
⟨*proof*⟩

lemma *finite-acyclic-wf-converse*: [|*finite r*; *acyclic r*|] ==> *wf* (*r*^−1)
⟨*proof*⟩

lemma *wf-iff-acyclic-if-finite*: *finite r* ==> *wf r* = *acyclic r*
⟨*proof*⟩

24.5 *nat* is well-founded

lemma *less-nat-rel*: *op* < = (λ*m n*. *n* = *Suc m*) ^++
⟨*proof*⟩

definition
  pred-nat :: (*nat* ∗ *nat*) *set* **where**
  pred-nat = {(*m*, *n*). *n* = *Suc m*}

definition
  less-than :: (*nat* ∗ *nat*) *set* **where**
  less-than = *pred-nat*^+

lemma *less-eq*: (*m*, *n*) ∈ *pred-nat*^+ ⟷ *m* < *n*
  ⟨*proof*⟩

lemma *pred-nat-trancl-eq-le*:
  (*m*, *n*) ∈ *pred-nat*^* ⟷ *m* ≤ *n*
  ⟨*proof*⟩

lemma *wf-pred-nat*: *wf pred-nat*
⟨*proof*⟩

lemma *wf-less-than* [*iff*]: *wf less-than*
⟨*proof*⟩

lemma *trans-less-than* [*iff*]: *trans less-than*
⟨*proof*⟩

lemma *less-than-iff* [*iff*]: ((*x*,*y*): *less-than*) = (*x*<*y*)
⟨*proof*⟩

lemma *wf-less*: *wf* {(*x*, *y*::*nat*). *x* < *y*}
⟨*proof*⟩

24.6 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [**?**].

inductive-set
acc :: ('*a* * '*a*) *set* => '*a set*
for *r* :: ('*a* * '*a*) *set*
where
accI: (!!*y*. (*y*, *x*) : *r* ==> *y* : *acc r*) ==> *x* : *acc r*

abbreviation
termip :: ('*a* => '*a* => *bool*) => '*a* => *bool* **where**
termip r == *accp* (r^{-1-1})

abbreviation
termi :: ('*a* * '*a*) *set* => '*a set* **where**
termi r == *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

Induction rules

theorem *accp-induct*:
assumes *major*: *accp r a*
assumes *hyp*: !!*x*. *accp r x* ==> ∀ *y*. *r y x* --> *P y* ==> *P x*
shows *P a*
⟨*proof*⟩

theorems *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp r b* ==> *r a b* ==> *accp r a*
⟨*proof*⟩

lemma *not-accp-down*:
assumes *na*: ¬ *accp R x*

obtains z **where** $R\ z\ x$ **and** $\neg\ accp\ R\ z$
⟨*proof*⟩

lemma *accp-downwards-aux*: $r^{**}\ b\ a \Longrightarrow accp\ r\ a \longrightarrow accp\ r\ b$
⟨*proof*⟩

theorem *accp-downwards*: $accp\ r\ a \Longrightarrow r^{**}\ b\ a \Longrightarrow accp\ r\ b$
⟨*proof*⟩

theorem *accp-wfPI*: $\forall x.\ accp\ r\ x \Longrightarrow wfP\ r$
⟨*proof*⟩

theorem *accp-wfPD*: $wfP\ r \Longrightarrow accp\ r\ x$
⟨*proof*⟩

theorem *wfP-accp-iff*: $wfP\ r = (\forall x.\ accp\ r\ x)$
⟨*proof*⟩

Smaller relations have bigger accessible parts:

lemma *accp-subset*:
assumes *sub*: $R1 \leq R2$
shows $accp\ R2 \leq accp\ R1$
⟨*proof*⟩

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
assumes *subset*: $D \leq accp\ R$
and *dcl*: $\bigwedge x\ z.\ [\![D\ x;\ R\ z\ x]\!] \Longrightarrow D\ z$
and $D\ x$
and *istep*: $\bigwedge x.\ [\![D\ x;\ (\bigwedge z.\ R\ z\ x \Longrightarrow P\ z)]\!] \Longrightarrow P\ x$
shows $P\ x$
⟨*proof*⟩

Set versions of the above theorems

lemmas *acc-induct* = *accp-induct* [*to-set*]

lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

lemmas *acc-downward* = *accp-downward* [*to-set*]

lemmas *not-acc-down* = *not-accp-down* [*to-set*]

lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]

lemmas *acc-downwards* = *accp-downwards* [*to-set*]

lemmas *acc-wfI* = *accp-wfPI* [*to-set*]

lemmas *acc-wfD* = *accp-wfPD* [*to-set*]

lemmas *wf-acc-iff* = *wfP-accp-iff* [*to-set*]

lemmas *acc-subset* = *accp-subset* [*to-set pred-subset-eq*]

lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set pred-subset-eq*]

24.7 Tools for building wellfounded relations

Inverse Image

lemma *wf-inv-image* [*simp*,*intro*!]: *wf*(*r*) ==> *wf*(*inv-image r* (*f*::'*a*=>'*b*))
⟨*proof*⟩

Measure functions into *nat*

definition *measure* :: ('*a* => *nat*) => ('*a* * '*a*)*set*
where *measure* == *inv-image less-than*

lemma *in-measure*[*simp*]: ((*x*,*y*) : *measure f*) = (*f x* < *f y*)
⟨*proof*⟩

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
⟨*proof*⟩

Lexicographic combinations

definition
lex-prod :: [('*a**'*a*)*set*, ('*b**'*b*)*set*] => (('*a**'*b*)*('*a**'*b*))*set*
(**infixr** <**lex**> *80*)
where
ra <**lex**> *rb* == {((*a*,*b*),(*a*',*b*')). (*a*,*a*') : *ra* | *a*=*a*' & (*b*,*b*') : *rb*}

lemma *wf-lex-prod* [*intro*!]: [| *wf*(*ra*); *wf*(*rb*) |] ==> *wf*(*ra* <**lex**> *rb*)
⟨*proof*⟩

lemma *in-lex-prod*[*simp*]:
(((*a*,*b*),(*a*',*b*')): *r* <**lex**> *s*) = ((*a*,*a*'): *r* ∨ (*a* = *a*' ∧ (*b*, *b*') : *s*))
⟨*proof*⟩

op <**lex**> preserves transitivity

lemma *trans-lex-prod* [*intro*!]:
[| *trans R1*; *trans R2* |] ==> *trans* (*R1* <**lex**> *R2*)
⟨*proof*⟩

lexicographic combinations with measure functions

definition
mlex-prod :: ('*a* ⇒ *nat*) ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* (**infixr** <**mlex**> *80*)
where
f <**mlex**> *R* = *inv-image* (*less-than* <**lex**> *R*) (%*x*. (*f x*, *x*))

lemma *wf-mlex*: *wf R* $\Longrightarrow$ *wf* (*f* $<*mlex*>$ *R*)
⟨*proof*⟩

lemma *mlex-less*: *f x* $<$ *f y* $\Longrightarrow$ $(x, y) \in f$ $<*mlex*>$ *R*
⟨*proof*⟩

lemma *mlex-leq*: *f x* $\leq$ *f y* $\Longrightarrow$ $(x, y) \in R \Longrightarrow (x, y) \in f$ $<*mlex*>$ *R*
⟨*proof*⟩

proper subset relation on finite sets

definition *finite-psubset* :: (*'a set* $*$ *'a set*) *set*
where *finite-psubset* == {(*A*,*B*). *A* $<$ *B* & *finite B*}

lemma *wf-finite-psubset*[*simp*]: *wf*(*finite-psubset*)
⟨*proof*⟩

lemma *trans-finite-psubset*: *trans finite-psubset*
⟨*proof*⟩

lemma *in-finite-psubset*[*simp*]: $(A, B) \in$ *finite-psubset* = (*A* $<$ *B* & *finite B*)
⟨*proof*⟩

max- and min-extension of order to finite sets

inductive-set *max-ext* :: (*'a* $\times$ *'a*) *set* $\Rightarrow$ (*'a set* $\times$ *'a set*) *set*
for *R* :: (*'a* $\times$ *'a*) *set*
where
max-extI[*intro*]: *finite X* $\Longrightarrow$ *finite Y* $\Longrightarrow$ *Y* $\neq$ {} $\Longrightarrow$ ($\bigwedge x.\ x \in X \Longrightarrow \exists\, y{\in}Y.$ $(x, y) \in R) \Longrightarrow (X, Y) \in$ *max-ext R*

lemma *max-ext-wf*:
assumes *wf*: *wf r*
shows *wf* (*max-ext r*)
⟨*proof*⟩

lemma *max-ext-additive*:
$(A, B) \in$ *max-ext R* $\Longrightarrow$ $(C, D) \in$ *max-ext R* $\Longrightarrow$
$(A \cup C, B \cup D) \in$ *max-ext R*
⟨*proof*⟩

definition
min-ext :: (*'a* $\times$ *'a*) *set* $\Rightarrow$ (*'a set* $\times$ *'a set*) *set*
where
[*code del*]: *min-ext r* = {(X, Y) | *X Y*. *X* $\neq$ {} $\wedge$ ($\forall\, y \in Y.$ ($\exists\, x \in X.$ $(x, y) \in$ *r*))}

lemma *min-ext-wf*:
assumes *wf r*

shows *wf* (*min-ext r*)
⟨*proof*⟩

24.8 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

lemma *sequence-trans*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* |] ==> (*f* (*i*+*k*), *f i*) : *r*^*
⟨*proof*⟩

lemma *wf-weak-decr-stable*:
assumes *as*: *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* *wf* (*r*^+)
shows *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
⟨*proof*⟩

lemma *weak-decr-stable*:
ALL i. *f* (*Suc i*) <= ((*f i*)::*nat*) ==> *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
⟨*proof*⟩

24.9 size of a datatype value

⟨*ML*⟩

lemma *size-bool* [*code*]:
size (*b*::*bool*) = *0* ⟨*proof*⟩

lemma *nat-size* [*simp*, *code*]: *size* (*n*::*nat*) = *n*
⟨*proof*⟩

declare *prod.size* [*no-atp*]

lemma [*code*]:
size (*P* :: *'a Predicate.pred*) = *0* ⟨*proof*⟩

lemma [*code*]:
pred-size f P = *0* ⟨*proof*⟩

end

25 FunDef: Function Definitions and Termination Proofs

theory *FunDef*
imports *Wellfounded*
uses
Tools/prop-logic.ML

Tools/sat-solver.ML
(*Tools/Function/function-lib.ML*)
(*Tools/Function/function-common.ML*)
(*Tools/Function/context-tree.ML*)
(*Tools/Function/function-core.ML*)
(*Tools/Function/sum-tree.ML*)
(*Tools/Function/mutual.ML*)
(*Tools/Function/pattern-split.ML*)
(*Tools/Function/function.ML*)
(*Tools/Function/relation.ML*)
(*Tools/Function/measure-functions.ML*)
(*Tools/Function/lexicographic-order.ML*)
(*Tools/Function/pat-completeness.ML*)
(*Tools/Function/fun.ML*)
(*Tools/Function/induction-schema.ML*)
(*Tools/Function/termination.ML*)
(*Tools/Function/scnp-solve.ML*)
(*Tools/Function/scnp-reconstruct.ML*)
begin

25.1 Definitions with default value.

definition
THE-default :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
THE-default d P = (*if* $(\exists!x.\ P\ x)$ *then* $(THE\ x.\ P\ x)$ *else d*)

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
⟨*proof*⟩

lemma *THE-default1-equality*:
$[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow THE\text{-}default\ d\ P = a$
⟨*proof*⟩

lemma *THE-default-none*:
$\neg(\exists!x.\ P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$
⟨*proof*⟩

lemma *fundef-ex1-existence*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $G\ x\ (f\ x)$
⟨*proof*⟩

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
assumes *elm*: $G\ x\ (h\ x)$
shows $h\ x = f\ x$

⟨*proof*⟩

lemma *fundef-ex1-iff*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $(G\ x\ y) = (f\ x = y)$
⟨*proof*⟩

lemma *fundef-default-value*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *graph*: $\bigwedge x\ y.\ G\ x\ y \Longrightarrow D\ x$
assumes $\neg\ D\ x$
shows $f\ x = d\ x$
⟨*proof*⟩

definition *in-rel-def*[*simp*]:
$in\text{-}rel\ R\ x\ y == (x,\ y) \in R$

lemma *wf-in-rel*:
$wf\ R \Longrightarrow wfP\ (in\text{-}rel\ R)$
⟨*proof*⟩

⟨*ML*⟩

25.2 Measure Functions

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$
where *is-measure-trivial*: *is-measure f*

⟨*ML*⟩

lemma *measure-size*[*measure-function*]: *is-measure size*
⟨*proof*⟩

lemma *measure-fst*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* $(\lambda p.\ f\ (fst\ p))$
⟨*proof*⟩
lemma *measure-snd*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* $(\lambda p.\ f\ (snd\ p))$
⟨*proof*⟩

⟨*ML*⟩

25.3 Congruence Rules

lemma *let-cong* [*fundef-cong*]:
$M = N \Longrightarrow (\bigwedge x.\ x = N \Longrightarrow f\ x = g\ x) \Longrightarrow Let\ M\ f = Let\ N\ g$
⟨*proof*⟩

lemmas [*fundef-cong*] =
if-cong image-cong INT-cong UN-cong

bex-cong ball-cong imp-cong

lemma *split-cong* [*fundef-cong*]:
$(\bigwedge x\ y.\ (x,\ y) = q \Longrightarrow f\ x\ y = g\ x\ y) \Longrightarrow p = q$
$\Longrightarrow$ *split f p* = *split g q*
⟨*proof*⟩

lemma *comp-cong* [*fundef-cong*]:
$f\ (g\ x) = f'\ (g'\ x') \Longrightarrow (f\ o\ g)\ x = (f'\ o\ g')\ x'$
⟨*proof*⟩

25.4 Simp rules for termination proofs

lemma *termination-basic-simps*[*termination-simp*]:
$x < (y::nat) \Longrightarrow x < y + z$
$x < z \Longrightarrow x < y + z$
$x \leq y \Longrightarrow x \leq y + (z::nat)$
$x \leq z \Longrightarrow x \leq y + (z::nat)$
$x < y \Longrightarrow x \leq (y::nat)$
⟨*proof*⟩

declare *le-imp-less-Suc*[*termination-simp*]

lemma *prod-size-simp*[*termination-simp*]:
prod-size f g p = *f* (*fst p*) + *g* (*snd p*) + *Suc 0*
⟨*proof*⟩

25.5 Decomposition

lemma *less-by-empty*:
$A = \{\} \Longrightarrow A \subseteq B$
and *union-comp-emptyL*:
$[\![\ A\ O\ C = \{};\ B\ O\ C = \{\}\ ]\!] \Longrightarrow (A \cup B)\ O\ C = \{\}$
and *union-comp-emptyR*:
$[\![\ A\ O\ B = \{};\ A\ O\ C = \{\}\ ]\!] \Longrightarrow A\ O\ (B \cup C) = \{\}$
and *wf-no-loop*:
$R\ O\ R = \{\} \Longrightarrow wf\ R$
⟨*proof*⟩

25.6 Reduction Pairs

definition
reduction-pair P = (*wf* (*fst P*) ∧ *fst P O snd P* ⊆ *fst P*)

lemma *reduction-pairI*[*intro*]: *wf R* $\Longrightarrow$ *R O S* ⊆ *R* $\Longrightarrow$ *reduction-pair* (*R*, *S*)
⟨*proof*⟩

lemma *reduction-pair-lemma*:
assumes *rp*: *reduction-pair P*
assumes *R* ⊆ *fst P*

assumes $S \subseteq snd\ P$
assumes $wf\ S$
shows $wf\ (R \cup S)$
⟨*proof*⟩

definition
$rp\text{-}inv\text{-}image = (\lambda(R,S)\ f.\ (inv\text{-}image\ R\ f,\ inv\text{-}image\ S\ f))$

lemma *rp-inv-image-rp*:
$reduction\text{-}pair\ P \Longrightarrow reduction\text{-}pair\ (rp\text{-}inv\text{-}image\ P\ f)$
⟨*proof*⟩

25.7 Concrete orders for SCNP termination proofs

definition $pair\text{-}less = less\text{-}than <*lex*> less\text{-}than$
definition [*code del*]: $pair\text{-}leq = pair\text{-}less\hat{}=$
definition $max\text{-}strict = max\text{-}ext\ pair\text{-}less$
definition [*code del*]: $max\text{-}weak = max\text{-}ext\ pair\text{-}leq \cup \{(\{\}, \{\})\}$
definition [*code del*]: $min\text{-}strict = min\text{-}ext\ pair\text{-}less$
definition [*code del*]: $min\text{-}weak = min\text{-}ext\ pair\text{-}leq \cup \{(\{\}, \{\})\}$

lemma *wf-pair-less*[*simp*]: $wf\ pair\text{-}less$
⟨*proof*⟩

Introduction rules for *pair-less*/*pair-leq*

lemma *pair-leqI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in pair\text{-}leq$
and *pair-leqI2*: $a \leq b \Longrightarrow s \leq t \Longrightarrow ((a, s), (b, t)) \in pair\text{-}leq$
and *pair-lessI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in pair\text{-}less$
and *pair-lessI2*: $a \leq b \Longrightarrow s < t \Longrightarrow ((a, s), (b, t)) \in pair\text{-}less$
⟨*proof*⟩

Introduction rules for max

lemma *smax-emptyI*:
$finite\ Y \Longrightarrow Y \neq \{\} \Longrightarrow (\{\}, Y) \in max\text{-}strict$
and *smax-insertI*:
$[\![y \in Y; (x, y) \in pair\text{-}less; (X, Y) \in max\text{-}strict]\!] \Longrightarrow (insert\ x\ X, Y) \in max\text{-}strict$
and *wmax-emptyI*:
$finite\ X \Longrightarrow (\{\}, X) \in max\text{-}weak$
and *wmax-insertI*:
$[\![y \in YS; (x, y) \in pair\text{-}leq; (XS, YS) \in max\text{-}weak]\!] \Longrightarrow (insert\ x\ XS, YS) \in max\text{-}weak$
⟨*proof*⟩

Introduction rules for min

lemma *smin-emptyI*:
$X \neq \{\} \Longrightarrow (X, \{\}) \in min\text{-}strict$
and *smin-insertI*:
$[\![x \in XS; (x, y) \in pair\text{-}less; (XS, YS) \in min\text{-}strict]\!] \Longrightarrow (XS, insert\ y\ YS) \in min\text{-}strict$

and *wmin-emptyI*:
$(X, \{\}) \in$ *min-weak*
and *wmin-insertI*:
$[\![x \in XS; (x, y) \in$ *pair-leq*; $(XS, YS) \in$ *min-weak*$]\!] \Longrightarrow (XS,$ *insert* $y\ YS) \in$ *min-weak*
⟨*proof*⟩

Reduction Pairs

lemma *max-ext-compat*:
assumes $R\ O\ S \subseteq R$
shows *max-ext* $R\ O$ (*max-ext* $S \cup \{(\{\},\{\})\}) \subseteq$ *max-ext* R
⟨*proof*⟩

lemma *max-rpair-set*: *reduction-pair* (*max-strict*, *max-weak*)
⟨*proof*⟩

lemma *min-ext-compat*:
assumes $R\ O\ S \subseteq R$
shows *min-ext* $R\ O$ (*min-ext* $S \cup \{(\{\},\{\})\}) \subseteq$ *min-ext* R
⟨*proof*⟩

lemma *min-rpair-set*: *reduction-pair* (*min-strict*, *min-weak*)
⟨*proof*⟩

25.8 Tool setup

⟨*ML*⟩

end

26 Extraction: Program extraction for HOL

theory *Extraction*
imports *Option*
uses *Tools/rewrite-hol-proof.ML*
begin

26.1 Setup

⟨*ML*⟩

lemmas [*extraction-expand*] =
meta-spec atomize-eq atomize-all atomize-imp atomize-conj
allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
induct-atomize induct-atomize' induct-rulify induct-rulify'
induct-rulify-fallback induct-trueI

True-implies-equals TrueE

lemmas [*extraction-expand-def*] =
induct-forall-def induct-implies-def induct-equal-def induct-conj-def
induct-true-def induct-false-def

datatype *sumbool* = *Left* | *Right*

26.2 Type of extracted program

extract-type
typeof (*Trueprop P*) ≡ *typeof P*

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ⟶ *Q*) ≡ *Type* (*TYPE*('*Q*))

typeof Q ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof* (*P* ⟶ *Q*) ≡ *Type* (*TYPE*(*Null*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ⟶ *Q*) ≡ *Type* (*TYPE*('*P* ⇒ '*Q*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*(*Null*))) ⟹
typeof (∀ *x*. *P x*) ≡ *Type* (*TYPE*(*Null*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*('*P*))) ⟹
typeof (∀ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a* ⇒ '*P*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*(*Null*))) ⟹
typeof (∃ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*('*P*))) ⟹
typeof (∃ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a* × '*P*))

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*(*Null*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*(*sumbool*))

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*('*Q option*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*(*Null*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*('*P option*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ∨ *Q*) ≡ *Type* (*TYPE*('*P* + '*Q*))

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ∧ *Q*) ≡ *Type* (*TYPE*('*Q*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*(*Null*)) ⟹

$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

26.3 Realizability

realizability

$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv (\forall x{::}'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P{::}bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

26.4 Computational content of basic inference rules

theorem *disjE-realizer*:
assumes *r*: $case\ x\ of\ Inl\ p \Rightarrow P\ p \mid Inr\ q \Rightarrow Q\ q$
and *r1*: $\bigwedge p.\ P\ p \Longrightarrow R\ (f\ p)$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows $R\ (case\ x\ of\ Inl\ p \Rightarrow f\ p \mid Inr\ q \Rightarrow g\ q)$
⟨*proof*⟩

theorem *disjE-realizer2*:
assumes *r*: $case\ x\ of\ None \Rightarrow P \mid Some\ q \Rightarrow Q\ q$
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows $R\ (case\ x\ of\ None \Rightarrow f \mid Some\ q \Rightarrow g\ q)$
⟨*proof*⟩

theorem *disjE-realizer3*:
assumes *r*: $case\ x\ of\ Left \Rightarrow P \mid Right \Rightarrow Q$
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $Q \Longrightarrow R\ g$
shows $R\ (case\ x\ of\ Left \Rightarrow f \mid Right \Rightarrow g)$
⟨*proof*⟩

theorem *conjI-realizer*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
⟨*proof*⟩

theorem *exI-realizer*:
$P\ y\ x \Longrightarrow P\ (snd\ (x,\ y))\ (fst\ (x,\ y))$ ⟨*proof*⟩

theorem *exE-realizer*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$

⟨*proof*⟩

theorem *exE-realizer'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow (\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q\ \langle proof\rangle$

realizers

impI (*P*, *Q*): $\lambda pq.\ pq$
  $\Lambda\ (c\text{: }\text{-})\ (d\text{: }\text{-})\ P\ Q\ pq\ (h\text{: }\text{-}).\ allI \cdot \text{-} \cdot c \cdot (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (h \cdot x))$

impI (*P*): *Null*
  $\Lambda\ (c\text{: }\text{-})\ P\ Q\ (h\text{: }\text{-}).\ allI \cdot \text{-} \cdot c \cdot (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (h \cdot x))$

impI (*Q*): $\lambda q.\ q\ \Lambda\ (c\text{: }\text{-})\ P\ Q\ q.\ impI \cdot \text{-} \cdot \text{-}$

impI: *Null impI*

mp (*P*, *Q*): $\lambda pq.\ pq$
  $\Lambda\ (c\text{: }\text{-})\ (d\text{: }\text{-})\ P\ Q\ pq\ (h\text{: }\text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (spec \cdot \text{-} \cdot p \cdot c \cdot h)$

mp (*P*): *Null*
  $\Lambda\ (c\text{: }\text{-})\ P\ Q\ (h\text{: }\text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (spec \cdot \text{-} \cdot p \cdot c \cdot h)$

mp (*Q*): $\lambda q.\ q\ \Lambda\ (c\text{: }\text{-})\ P\ Q\ q.\ mp \cdot \text{-} \cdot \text{-}$

mp: *Null mp*

allI (*P*): $\lambda p.\ p\ \Lambda\ (c\text{: }\text{-})\ P\ (d\text{: }\text{-})\ p.\ allI \cdot \text{-} \cdot d$

allI: *Null allI*

spec (*P*): $\lambda x\ p.\ p\ x\ \Lambda\ (c\text{: }\text{-})\ P\ x\ (d\text{: }\text{-})\ p.\ spec \cdot \text{-} \cdot x \cdot d$

spec: *Null spec*

exI (*P*): $\lambda x\ p.\ (x,\ p)\ \Lambda\ (c\text{: }\text{-})\ P\ x\ (d\text{: }\text{-})\ p.\ exI\text{-}realizer \cdot P \cdot p \cdot x \cdot c \cdot d$

exI: $\lambda x.\ x\ \Lambda\ P\ x\ (c\text{: }\text{-})\ (h\text{: }\text{-}).\ h$

exE (*P*, *Q*): $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
  $\Lambda\ (c\text{: }\text{-})\ (d\text{: }\text{-})\ P\ Q\ (e\text{: }\text{-})\ p\ (h\text{: }\text{-})\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h$

exE (*P*): *Null*
  $\Lambda\ (c\text{: }\text{-})\ P\ Q\ (d\text{: }\text{-})\ p.\ exE\text{-}realizer' \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot c \cdot d$

exE (*Q*): $\lambda x\ pq.\ pq\ x$
  $\Lambda\ (c\text{: }\text{-})\ P\ Q\ (d\text{: }\text{-})\ x\ (h1\text{: }\text{-})\ pq\ (h2\text{: }\text{-}).\ h2 \cdot x \cdot h1$

exE: *Null*
  $\Lambda\ P\ Q\ (c\text{: }\text{-})\ x\ (h1\text{: }\text{-})\ (h2\text{: }\text{-}).\ h2 \cdot x \cdot h1$

conjI (*P*, *Q*): *Pair*
Λ (*c*: -) (*d*: -) *P Q p* (*h*: -) *q*. *conjI-realizer* · *P* · *p* · *Q* · *q* • *c* • *d* • *h*

conjI (*P*): λ*p*. *p*
Λ (*c*: -) *P Q p*. *conjI* · - · -

conjI (*Q*): λ*q*. *q*
Λ (*c*: -) *P Q* (*h*: -) *q*. *conjI* · - · - • *h*

conjI: *Null conjI*

conjunct1 (*P*, *Q*): *fst*
Λ (*c*: -) (*d*: -) *P Q pq*. *conjunct1* · - · -

conjunct1 (*P*): λ*p*. *p*
Λ (*c*: -) *P Q p*. *conjunct1* · - · -

conjunct1 (*Q*): *Null*
Λ (*c*: -) *P Q q*. *conjunct1* · - · -

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
Λ (*c*: -) (*d*: -) *P Q pq*. *conjunct2* · - · -

conjunct2 (*P*): *Null*
Λ (*c*: -) *P Q p*. *conjunct2* · - · -

conjunct2 (*Q*): λ*p*. *p*
Λ (*c*: -) *P Q p*. *conjunct2* · - · -

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
Λ (*c*: -) (*d*: -) *P Q p*. *iffD2* · - · - • (*sum.cases-1* · *P* · - · *p* • *arity-type-bool* • *c* • *d*)

disjI1 (*P*): *Some*
Λ (*c*: -) *P Q p*. *iffD2* · - · - • (*option.cases-2* · - · *P* · *p* • *arity-type-bool* • *c*)

disjI1 (*Q*): *None*
Λ (*c*: -) *P Q*. *iffD2* · - · - • (*option.cases-1* · - · - • *arity-type-bool* • *c*)

disjI1: *Left*
Λ *P Q*. *iffD2* · - · - • (*sumbool.cases-1* · - · - • *arity-type-bool*)

disjI2 (*P*, *Q*): *Inr*
Λ (*d*: -) (*c*: -) *Q P q*. *iffD2* · - · - • (*sum.cases-2* · - · *Q* · *q* • *arity-type-bool* •

$c \cdot d$)

disjI2 (*P*): *None*
Λ (*c*: -) *Q P*. *iffD2* · - · - · (*option.cases-1* · - · - · *arity-type-bool* · *c*)

disjI2 (*Q*): *Some*
Λ (*c*: -) *Q P q*. *iffD2* · - · - · (*option.cases-2* · - · *Q* · *q* · *arity-type-bool* · *c*)

disjI2: *Right*
Λ *Q P*. *iffD2* · - · - · (*sumbool.cases-2* · - · - · *arity-type-bool*)

disjE (*P*, *Q*, *R*): λ*pq pr qr*.
(*case pq of Inl p* ⇒ *pr p* | *Inr q* ⇒ *qr q*)
Λ (*c*: -) (*d*: -) (*e*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
disjE-realizer · - · - · *pq* · *R* · *pr* · *qr* · *c* · *d* · *e* · *h1* · *h2*

disjE (*Q*, *R*): λ*pq pr qr*.
(*case pq of None* ⇒ *pr* | *Some q* ⇒ *qr q*)
Λ (*c*: -) (*d*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
disjE-realizer2 · - · - · *pq* · *R* · *pr* · *qr* · *c* · *d* · *h1* · *h2*

disjE (*P*, *R*): λ*pq pr qr*.
(*case pq of None* ⇒ *qr* | *Some p* ⇒ *pr p*)
Λ (*c*: -) (*d*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr* (*h3*: -).
disjE-realizer2 · - · - · *pq* · *R* · *qr* · *pr* · *c* · *d* · *h1* · *h3* · *h2*

disjE (*R*): λ*pq pr qr*.
(*case pq of Left* ⇒ *pr* | *Right* ⇒ *qr*)
Λ (*c*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
disjE-realizer3 · - · - · *pq* · *R* · *pr* · *qr* · *c* · *h1* · *h2*

disjE (*P*, *Q*): *Null*
Λ (*c*: -) (*d*: -) *P Q R pq*. *disjE-realizer* · - · - · *pq* · (λ*x*. *R*) · - · - · *c* · *d* · *arity-type-bool*

disjE (*Q*): *Null*
Λ (*c*: -) *P Q R pq*. *disjE-realizer2* · - · - · *pq* · (λ*x*. *R*) · - · - · *c* · *arity-type-bool*

disjE (*P*): *Null*
Λ (*c*: -) *P Q R pq* (*h1*: -) (*h2*: -) (*h3*: -).
disjE-realizer2 · - · - · *pq* · (λ*x*. *R*) · - · - · *c* · *arity-type-bool* · *h1* · *h3* · *h2*

disjE: *Null*
Λ *P Q R pq*. *disjE-realizer3* · - · - · *pq* · (λ*x*. *R*) · - · - · *arity-type-bool*

FalseE (*P*): *default*
Λ (*c*: -) *P*. *FalseE* · -

FalseE: *Null FalseE*

notI (*P*): *Null*
  Λ (*c*: -) *P* (*h*: -). *allI* · - • *c* • (Λ *x*. *notI* · - • (*h* · *x*))

notI: *Null notI*

notE (*P*, *R*): λ*p*. *default*
  Λ (*c*: -) (*d*: -) *P R* (*h*: -) *p*. *notE* · - - - • (*spec* · - · *p* • *c* • *h*)

notE (*P*): *Null*
  Λ (*c*: -) *P R* (*h*: -) *p*. *notE* · - - - • (*spec* · - · *p* • *c* • *h*)

notE (*R*): *default*
  Λ (*c*: -) *P R*. *notE* · - · -

notE: *Null notE*

subst (*P*): λ*s t ps*. *ps*
  Λ (*c*: -) *s t P* (*d*: -) (*h*: -) *ps*. *subst* · *s* · *t* · *P ps* • *d* • *h*

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
  Λ (*d*: -) (*c*: -) *Q P pq* (*h*: -) *p*.
    mp · - - - • (*spec* · - · *p* • *d* • (*conjunct1* · - - - • *h*))

iffD1 (*P*): λ*p*. *p*
  Λ (*c*: -) *Q P p* (*h*: -). *mp* · - - - • (*conjunct1* · - - - • *h*)

iffD1 (*Q*): *Null*
  Λ (*c*: -) *Q P q1* (*h*: -) *q2*.
    mp · - - - • (*spec* · - · *q2* • *c* • (*conjunct1* · - - - • *h*))

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
  Λ (*c*: -) (*d*: -) *P Q pq* (*h*: -) *q*.
    mp · - - - • (*spec* · - · *q* • *d* • (*conjunct2* · - - - • *h*))

iffD2 (*P*): λ*p*. *p*
  Λ (*c*: -) *P Q p* (*h*: -). *mp* · - - - • (*conjunct2* · - - - • *h*)

iffD2 (*Q*): *Null*
  Λ (*c*: -) *P Q q1* (*h*: -) *q2*.
    mp · - - - • (*spec* · - · *q2* • *c* • (*conjunct2* · - - - • *h*))

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*

Λ (c: -) (d: -) P Q pq ($h1$: -) qp ($h2$: -). *conjI-realizer* ·
(λpq. $\forall x$. P x $\longrightarrow$ Q (pq x)) · pq ·
(λqp. $\forall x$. Q x $\longrightarrow$ P (qp x)) · qp ·
(*arity-type-fun* · c · d) ·
(*arity-type-fun* · d · c) ·
(*allI* · - · c · (Λ x. *impI* · - · - · - · ($h1$ · x))) ·
(*allI* · - · d · (Λ x. *impI* · - · - · - · ($h2$ · x)))

iffI (P): λp. p
Λ (c: -) P Q ($h1$: -) p ($h2$: -). *conjI* · - · - ·
(*allI* · - · c · (Λ x. *impI* · - · - · - · ($h1$ · x))) ·
(*impI* · - · - · - · $h2$)

iffI (Q): λq. q
Λ (c: -) P Q q ($h1$: -) ($h2$: -). *conjI* · - · - · -
(*impI* · - · - · - · $h1$) ·
(*allI* · - · c · (Λ x. *impI* · - · - · - · ($h2$ · x)))

iffI: *Null iffI*

end

27 Plain: Plain HOL

theory *Plain*
imports *Datatype Record FunDef Extraction*
begin

Plain bootstrap of fundamental HOL tools and packages; does not include *Hilbert-Choice*.

⟨*ML*⟩

end

28 Big-Operators: Big operators and finite (non-empty) sets

theory *Big-Operators*
imports *Plain*
begin

28.1 Generic monoid operation over a set

no-notation *times* (**infixl** $*$ *70*)
no-notation *Groups.one* (*1*)

locale *comm-monoid-big* = *comm-monoid* +
  fixes *F* :: (*'b* ⇒ *'a*) ⇒ *'b set* ⇒ *'a*
  assumes *F-eq*: *F g A* = (*if finite A then fold-image* (*op* ∗) *g 1 A else 1*)

sublocale *comm-monoid-big* < *folding-image* ⟨*proof*⟩

context *comm-monoid-big*
begin

lemma *infinite* [*simp*]:
  ¬ *finite A* ⟹ *F g A* = *1*
  ⟨*proof*⟩

end

for ad-hoc proofs for *fold-image*

lemma (**in** *comm-monoid-add*) *comm-monoid-mult*:
  class.comm-monoid-mult (*op* +) *0*
⟨*proof*⟩

notation *times* (**infixl** ∗ *70*)
notation *Groups.one* (*1*)

28.2 Generalized summation over a set

definition (**in** *comm-monoid-add*) *setsum* :: (*'b* ⇒ *'a*) => *'b set* => *'a* **where**
  setsum f A = (*if finite A then fold-image* (*op* +) *f 0 A else 0*)

sublocale *comm-monoid-add* < *setsum*!: *comm-monoid-big op* + *0 setsum* ⟨*proof*⟩

abbreviation
  Setsum (∑ - [*1000*] *999*) **where**
  ∑ *A* == *setsum* (%*x*. *x*) *A*

Now: lot's of fancy syntax. First, *setsum* (λ*x*. *e*) *A* is written ∑ *x*∈*A*. *e*.

syntax
  -setsum :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-add* ((*3SUM* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
  -setsum :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
  -setsum :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
  SUM i:*A*. *b* == *CONST setsum* (%*i*. *b*) *A*
  ∑ *i*∈*A*. *b* == *CONST setsum* (%*i*. *b*) *A*

Instead of $\sum x{\in}\{x.\ P\}.\ e$ 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∑ - | (-)./ -) [0,0,10] 10)
syntax (*HTML* **output**)
-qsetsum :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∑ - | (-)./ -) [0,0,10] 10)

translations
SUM x|P. t => CONST setsum (%x. t) {x. P}
∑ x|P. t => CONST setsum (%x. t) {x. P}

⟨*ML*⟩

lemma *setsum-empty*:
setsum f {} = 0
⟨*proof*⟩

lemma *setsum-insert*:
finite F ==> a ∉ F ==> setsum f (insert a F) = f a + setsum f F
⟨*proof*⟩

lemma *setsum-infinite*:
~ finite A ==> setsum f A = 0
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-reindex*:
assumes *inj-on f B* **shows** *setsum h (f ' B) = setsum (h ∘ f) B*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-reindex-id*:
inj-on f B ==> setsum f B = setsum id (f ' B)
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-reindex-nonzero*:
assumes *fS*: *finite S*
and *nz*: *⋀ x y. x ∈ S ⟹ y ∈ S ⟹ x ≠ y ⟹ f x = f y ⟹ h (f x) = 0*
shows *setsum h (f ' S) = setsum (h o f) S*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-cong*:
A = B ==> (!!x. x:B ==> f x = g x) ==> setsum f A = setsum g B
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *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 (**in** *comm-monoid-add*) *setsum-cong2*: $[\![\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x]\!] \Longrightarrow$ *setsum f A = setsum g A*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-reindex-cong*:
[|*inj-on f A*; *B = f ' A*; !!*a*. *a*:*A* ⟹ *g a = h* (*f a*)|]
==> *setsum h B = setsum g A*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-0*[*simp*]: *setsum* (%*i*. *0*) *A = 0*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-0'*: *ALL a*:*A*. *f a = 0* ==> *setsum f A = 0*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *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 (**in** *comm-monoid-add*) *setsum-Un-disjoint*: *finite A* ==> *finite B*
==> *A Int B* = {} ==> *setsum g* (*A Un B*) = *setsum g A* + *setsum g B*
⟨*proof*⟩

lemma *setsum-mono-zero-left*:
assumes *fT*: *finite T* **and** *ST*: *S* ⊆ *T*
and *z*: ∀ *i* ∈ *T* − *S*. *f i = 0*
shows *setsum f S = setsum f T*
⟨*proof*⟩

lemma *setsum-mono-zero-right*:
finite T ⟹ *S* ⊆ *T* ⟹ ∀ *i* ∈ *T* − *S*. *f i = 0* ⟹ *setsum f T = setsum f S*
⟨*proof*⟩

lemma *setsum-mono-zero-cong-left*:
assumes *fT*: *finite T* **and** *ST*: *S* ⊆ *T*
and *z*: ∀ *i* ∈ *T* − *S*. *g i = 0*
and *fg*: $\bigwedge x.\ x \in S \Longrightarrow f\ x = g\ x$
shows *setsum f S = setsum g T*
⟨*proof*⟩

lemma *setsum-mono-zero-cong-right*:
assumes *fT*: *finite T* **and** *ST*: *S* ⊆ *T*
and *z*: ∀ *i* ∈ *T* − *S*. *f i = 0*
and *fg*: $\bigwedge x.\ x \in S \Longrightarrow f\ x = g\ x$
shows *setsum f T = setsum g S*
⟨*proof*⟩

lemma *setsum-delta*:
assumes *fS*: *finite S*
shows *setsum* (λ*k*. *if k=a then b k else 0*) *S* = (*if a* ∈ *S then b a else 0*)
⟨*proof*⟩
lemma *setsum-delta'*:
assumes *fS*: *finite S* **shows**
setsum (λ*k*. *if a* = *k then b k else 0*) *S* =
(*if a*∈ *S then b a else 0*)
⟨*proof*⟩

lemma *setsum-restrict-set*:
assumes *fA*: *finite A*
shows *setsum f* (*A* ∩ *B*) = *setsum* (λ*x*. *if x* ∈ *B then f x else 0*) *A*
⟨*proof*⟩

lemma *setsum-cases*:
assumes *fA*: *finite A*
shows *setsum* (λ*x*. *if P x then f x else g x*) *A* =
setsum f (*A* ∩ {*x*. *P x*}) + *setsum g* (*A* ∩ − {*x*. *P x*})
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-UN-disjoint*:
assumes *finite I* **and** *ALL i*:*I*. *finite* (*A i*)
and *ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* −−> *A i Int A j* = {}
shows *setsum f* (*UNION I A*) = (∑ *i*∈*I*. *setsum f* (*A i*))
⟨*proof*⟩

No need to assume that C is finite. If infinite, the rhs is directly 0, and $\bigcup C$ is also infinite, hence the lhs is also 0.

lemma *setsum-Union-disjoint*:
[| (*ALL A*:*C*. *finite A*);
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* −−> *A Int B* = {}) |]
==> *setsum f* (*Union C*) = *setsum* (*setsum f*) *C*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-Sigma*:
assumes *finite A* **and** *ALL x*:*A*. *finite* (*B x*)
shows (∑ *x*∈*A*. (∑ *y*∈*B x*. *f x y*)) = (∑ (*x*,*y*)∈(*SIGMA x*:*A*. *B x*). *f x y*)
⟨*proof*⟩

Here we can eliminate the finiteness assumptions, by cases.

lemma *setsum-cartesian-product*:
(∑ *x*∈*A*. (∑ *y*∈*B*. *f x y*)) = (∑ (*x*,*y*) ∈ *A* <*> *B*. *f x y*)
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-addf*: *setsum* (%*x*. *f x* + *g x*) *A* = (*setsum f A* + *setsum g A*)
⟨*proof*⟩

28.2.1 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-eq-Suc0-iff*: *finite A* ⟹
(*setsum f A* = *Suc 0*) = (*EX a*:*A*. *f a* = *Suc 0* & (*ALL b*:*A*. *a*≠*b* ⟶ *f b* = *0*))
⟨*proof*⟩

lemmas *setsum-eq-1-iff* = *setsum-eq-Suc0-iff*[*simplified One-nat-def*[*symmetric*]]

lemma *setsum-Un-nat*: *finite A* ==> *finite B* ==>
(*setsum f* (*A Un B*) :: *nat*) = *setsum f A* + *setsum f B* − *setsum f* (*A Int B*)
— For the natural numbers, we have subtraction.
⟨*proof*⟩

lemma *setsum-Un*: *finite A* ==> *finite B* ==>
(*setsum f* (*A Un B*) :: *'a* :: *ab-group-add*) =
setsum f A + *setsum f B* − *setsum f* (*A Int B*)
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-eq-general-reverses*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *kh*: ⋀*y*. *y* ∈ *T* ⟹ *k y* ∈ *S* ∧ *h* (*k y*) = *y*
and *hk*: ⋀*x*. *x* ∈ *S* ⟹ *h x* ∈ *T* ∧ *k* (*h x*) = *x* ∧ *g* (*h x*) = *f x*
shows *setsum f S* = *setsum g T*
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *setsum-Un-zero*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: ∀ *x* ∈ *S*∩*T*. *f x* = *0*
shows *setsum f* (*S* ∪ *T*) = *setsum f S* + *setsum f T*
⟨*proof*⟩

lemma *setsum-UNION-zero*:
assumes *fS*: *finite S* **and** *fSS*: ∀ *T* ∈ *S*. *finite T*
and *f0*: ⋀*T1 T2 x*. *T1*∈*S* ⟹ *T2*∈*S* ⟹ *T1* ≠ *T2* ⟹ *x* ∈ *T1* ⟹ *x* ∈ *T2* ⟹ *f x* = *0*
shows *setsum f* (⋃ *S*) = *setsum* (λ*T*. *setsum f T*) *S*
⟨*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-diff1-ring*: **assumes** *finite* A $a \in A$
shows $setsum\ f\ (A - \{a\}) = setsum\ f\ A - (f\ a{::}'a{::}ring)$
⟨*proof*⟩

lemma *setsum-diff-nat*:
assumes *finite* B **and** $B \subseteq A$
shows $(setsum\ f\ (A - B) :: nat) = (setsum\ f\ A) - (setsum\ f\ B)$
⟨*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*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i{::}'a) \leq ((g\ i){::}('b{::}\{comm\text{-}monoid\text{-}add,\ ordered\text{-}ab\text{-}semigroup\text{-}add\}))$
shows $(\sum i{\in}K.\ f\ i) \leq (\sum i{\in}K.\ g\ i)$
⟨*proof*⟩

lemma *setsum-strict-mono*:
fixes $f :: 'a \Rightarrow 'b{::}\{ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add,comm\text{-}monoid\text{-}add\}$
assumes *finite* A $A \neq \{\}$
and $!!x.\ x{:}A \Longrightarrow f\ x < g\ x$
shows $setsum\ f\ A < setsum\ g\ A$
⟨*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*: $\forall x{\in}A.\ (0{::}'a{::}\{ordered\text{-}ab\text{-}semigroup\text{-}add,comm\text{-}monoid\text{-}add\}) \leq f\ x$
shows $0 \leq setsum\ f\ A$
⟨*proof*⟩

lemma *setsum-nonpos*:
assumes *np*: $\forall x{\in}A.\ f\ x \leq (0{::}'a{::}\{ordered\text{-}ab\text{-}semigroup\text{-}add,comm\text{-}monoid\text{-}add\})$
shows $setsum\ f\ A \leq 0$
⟨*proof*⟩

lemma *setsum-nonneg-leq-bound*:
fixes $f :: 'a \Rightarrow 'b{::}\{ordered\text{-}ab\text{-}group\text{-}add\}$
assumes *finite* s $\bigwedge i.\ i \in s \Longrightarrow f\ i \geq 0$ $(\sum i \in s.\ f\ i) = B$ $i \in s$
shows $f\ i \leq B$
⟨*proof*⟩

lemma *setsum-nonneg-0*:
fixes $f :: 'a \Rightarrow 'b{::}\{ordered\text{-}ab\text{-}group\text{-}add\}$
assumes *finite* s **and** *pos*: $\bigwedge\ i.\ i \in s \Longrightarrow f\ i \geq 0$
and $(\sum\ i \in s.\ f\ i) = 0$ **and** i: $i \in s$
shows $f\ i = 0$
⟨*proof*⟩

lemma *setsum-mono2*:
fixes $f :: 'a \Rightarrow 'b :: ordered\text{-}comm\text{-}monoid\text{-}add$
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 <= $((f\ x){::}'a{::}\{comm\text{-}monoid\text{-}add,ordered\text{-}ab\text{-}semigroup\text{-}add\})$ ==>
$setsum\ f\ A$ <= $setsum\ f\ B$
⟨*proof*⟩

lemma *setsum-right-distrib*:
fixes $f :: 'a$ => $('b{::}semiring\text{-}0)$
shows $r * setsum\ f\ A = setsum\ (\%n.\ r * f\ n)\ A$
⟨*proof*⟩

lemma *setsum-left-distrib*:
$setsum\ f\ A * (r{::}'a{::}semiring\text{-}0) = (\sum n{\in}A.\ f\ n * r)$
⟨*proof*⟩

lemma *setsum-divide-distrib*:
$setsum\ f\ A\ /\ (r{::}'a{::}field) = (\sum n{\in}A.\ f\ n\ /\ r)$
⟨*proof*⟩

lemma *setsum-abs*[*iff*]:
fixes $f :: 'a => ('b::ordered\text{-}ab\text{-}group\text{-}add\text{-}abs)$
shows $abs\ (setsum\ f\ A) \leq setsum\ (\%i.\ abs(f\ i))\ A$
⟨*proof*⟩

lemma *setsum-abs-ge-zero*[*iff*]:
fixes $f :: 'a => ('b::ordered\text{-}ab\text{-}group\text{-}add\text{-}abs)$
shows $0 \leq setsum\ (\%i.\ abs(f\ i))\ A$
⟨*proof*⟩

lemma *abs-setsum-abs*[*simp*]:
fixes $f :: 'a => ('b::ordered\text{-}ab\text{-}group\text{-}add\text{-}abs)$
shows $abs\ (\sum a{\in}A.\ abs(f\ a)) = (\sum a{\in}A.\ abs(f\ a))$
⟨*proof*⟩

lemma *setsum-Plus*:
fixes $A :: 'a\ set$ **and** $B :: 'b\ set$
assumes *fin*: *finite A finite B*
shows $setsum\ f\ (A <+> B) = setsum\ (f \circ Inl)\ A + setsum\ (f \circ Inr)\ B$
⟨*proof*⟩

Commuting outer and inner summation

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*⟩

lemma *setsum-product*:
fixes $f :: 'a => ('b::semiring\text{-}0)$
shows $setsum\ f\ A * setsum\ g\ B = (\sum i{\in}A.\ \sum j{\in}B.\ f\ i * g\ j)$
⟨*proof*⟩

lemma *setsum-mult-setsum-if-inj*:
fixes $f :: 'a => ('b::semiring\text{-}0)$
shows $inj\text{-}on\ (\%(a,b).\ f\ a * g\ b)\ (A \times B) ==>$
$setsum\ f\ A * setsum\ g\ B = setsum\ id\ \{f\ a * g\ b | a\ b.\ a{:}A\ \&\ b{:}B\}$
⟨*proof*⟩

lemma *setsum-constant* [*simp*]: $(\sum x \in A.\ y) = of\text{-}nat(card\ A) * y$
⟨*proof*⟩

lemma *setsum-bounded*:
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\ i \leq (K::'a::\{semiring\text{-}1,\ ordered\text{-}ab\text{-}semigroup\text{-}add\})$
shows $setsum\ f\ A \leq of\text{-}nat(card\ A) * K$
⟨*proof*⟩

28.2.2 Cardinality as special case of *setsum*

lemma *card-eq-setsum*:

card A = setsum (λx. 1) A
⟨*proof*⟩

lemma *card-UN-disjoint*:
finite I ==> (ALL i:I. finite (A i)) ==>
(ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {}) ==> card (UNION I A) = (∑ i∈I. card(A i))
⟨*proof*⟩

lemma *card-Union-disjoint*:
finite C ==> (ALL A:C. finite A) ==>
(ALL A:C. ALL B:C. A ≠ B --> A Int B = {}) ==> card (Union C) = setsum card C
⟨*proof*⟩

The image of a finite set can be expressed using *fold-image*.

lemma *image-eq-fold-image*:
finite A ==> f ' A = fold-image (op Un) (%x. {f x}) {} A
⟨*proof*⟩

28.2.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
⟦ *finite A*; *ALL a:A. finite (B a)* ⟧
⟹ *card (SIGMA x: A. B x) = (∑ a∈A. card (B a))*
⟨*proof*⟩

lemma *card-cartesian-product*: *card (A <*> B) = card(A) * card(B)*
⟨*proof*⟩

lemma *card-cartesian-product-singleton*: *card({x} <*> A) = card(A)*
⟨*proof*⟩

28.3 Generalized product over a set

definition (**in** *comm-monoid-mult*) *setprod* :: *('b ⇒ 'a) => 'b set => 'a* **where**
*setprod f A = (if finite A then fold-image (op *) f 1 A else 1)*

sublocale *comm-monoid-mult < setprod*!: *comm-monoid-big op * 1 setprod* ⟨*proof*⟩

abbreviation
Setprod (∏ - [*1000*] *999*) **where**
∏ A == setprod (%x. x) A

syntax
-setprod :: *pttrn => 'a set => 'b => 'b::comm-monoid-mult* ((*3PROD -:-. -*) [*0*, *51*, *10*] *10*)

syntax (*xsymbols*)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*$\prod$ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*$\prod$ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
PROD i:*A*. *b* == *CONST setprod* (%*i*. *b*) *A*
$\prod$ *i*∈*A*. *b* == *CONST setprod* (%*i*. *b*) *A*

Instead of $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3PROD* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\prod$ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\prod$ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
PROD x|*P*. *t* => *CONST setprod* (%*x*. *t*) {*x*. *P*}
$\prod$ *x*|*P*. *t* => *CONST setprod* (%*x*. *t*) {*x*. *P*}

lemma *setprod-empty*: *setprod f* {} = *1*
⟨*proof*⟩

lemma *setprod-insert*: [| *finite A*; *a* ∉ *A* |] ==>
setprod f (*insert a A*) = *f a* ∗ *setprod f A*
⟨*proof*⟩

lemma *setprod-infinite*: ~ *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*[*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* ==>

$B = f \text{ ` } A \Longrightarrow g = h \circ f \Longrightarrow setprod\ h\ B = setprod\ g\ A$
⟨*proof*⟩

lemma *strong-setprod-reindex-cong*: **assumes** *i*: *inj-on f A*
and *B*: $B = f \text{ ` } A$ **and** *eq*: $\bigwedge x.\ x \in A \Longrightarrow g\ x = (h \circ f)\ x$
shows $setprod\ h\ B = setprod\ g\ A$
⟨*proof*⟩

lemma *setprod-Un-one*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: $\forall x \in S \cap T.\ f\ x = 1$
shows $setprod\ f\ (S \cup T) = setprod\ f\ S * setprod\ f\ T$
⟨*proof*⟩

lemma *setprod-1*: $setprod\ (\%i.\ 1)\ A = 1$
⟨*proof*⟩

lemma *setprod-1'*: $ALL\ a{:}F.\ f\ a = 1 \Longrightarrow setprod\ f\ F = 1$
⟨*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*
==> $A\ Int\ B = \{\} \Longrightarrow setprod\ g\ (A\ Un\ B) = setprod\ g\ A * setprod\ g\ B$
⟨*proof*⟩

lemma *setprod-mono-one-left*:
assumes *fT*: *finite T* **and** *ST*: $S \subseteq T$
and *z*: $\forall i \in T - S.\ f\ i = 1$
shows $setprod\ f\ S = setprod\ f\ T$
⟨*proof*⟩

lemmas *setprod-mono-one-right* = *setprod-mono-one-left* [*THEN sym*]

lemma *setprod-delta*:
assumes *fS*: *finite S*
shows $setprod\ (\lambda k.\ if\ k{=}a\ then\ b\ k\ else\ 1)\ S = (if\ a \in S\ then\ b\ a\ else\ 1)$
⟨*proof*⟩

lemma *setprod-delta'*:
assumes *fS*: *finite S* **shows**
$setprod\ (\lambda k.\ if\ a = k\ then\ b\ k\ else\ 1)\ S =$
$(if\ a{\in}\ S\ then\ b\ a\ else\ 1)$
⟨*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) ==>*
(∏ x∈A. (∏ y∈ B x. f x y)) =
(∏ (x,y)∈(SIGMA x:A. B x). f x y)
⟨*proof*⟩

Here we can eliminate the finiteness assumptions, by cases.

lemma *setprod-cartesian-product*:
(∏ x∈A. (∏ y∈ B. f x y)) = (∏ (x,y)∈(A <> B). f x y)*
⟨*proof*⟩

lemma *setprod-timesf*:
*setprod (%x. f x * g x) A = (setprod f A * setprod g A)*
⟨*proof*⟩

28.3.1 Properties in more restricted classes of structures

lemma *setprod-eq-1-iff* [*simp*]:
finite F ==> (setprod f F = 1) = (ALL a:F. f a = (1::nat))
⟨*proof*⟩

lemma *setprod-zero*:
finite A ==> EX x: A. f x = (0::'a::comm-semiring-1) ==> setprod f A = 0
⟨*proof*⟩

lemma *setprod-nonneg* [*rule-format*]:
(ALL x: A. (0::'a::linordered-semidom) ≤ f x) --> 0 ≤ setprod f A
⟨*proof*⟩

lemma *setprod-pos* [*rule-format*]: *(ALL x: A. (0::'a::linordered-semidom) < f x)*
--> 0 < setprod f A
⟨*proof*⟩

lemma *setprod-zero-iff*[*simp*]: *finite A ==>*
(setprod f A = (0::'a::{comm-semiring-1,no-zero-divisors})) =
(EX x: A. f x = 0)
⟨*proof*⟩

lemma *setprod-pos-nat*:
finite S ==> (ALL x : S. f x > (0::nat)) ==> setprod f S > 0
⟨*proof*⟩

lemma *setprod-pos-nat-iff* [*simp*]:
finite S ==> (setprod f S > 0) = (ALL x : S. f x > (0::nat))
⟨*proof*⟩

lemma *setprod-Un*: *finite A ==> finite B ==> (ALL x: A Int B. f x ≠ 0) ==>*
(setprod f (A Un B) :: 'a ::{field})
*= setprod f A * setprod f B / setprod f (A Int B)*
⟨*proof*⟩

lemma *setprod-diff1*: *finite A ==> f a ≠ 0 ==>*
(setprod f (A − {a}) :: 'a :: {field}) =
(if a:A then setprod f A / f a else setprod f A)
⟨*proof*⟩

lemma *setprod-inversef*:
fixes *f :: 'b ⇒ 'a::field-inverse-zero*
shows *finite A ==> setprod (inverse ∘ f) A = inverse (setprod f A)*
⟨*proof*⟩

lemma *setprod-dividef*:
fixes *f :: 'b ⇒ 'a::field-inverse-zero*
shows *finite A*
==> setprod (%x. f x / g x) A = setprod f A / setprod g A
⟨*proof*⟩

lemma *setprod-dvd-setprod* [*rule-format*]:
(ALL x : A. f x dvd g x) ⟶ setprod f A dvd setprod g A
⟨*proof*⟩

lemma *setprod-dvd-setprod-subset*:
finite B ⟹ A <= B ⟹ setprod f A dvd setprod f B
⟨*proof*⟩

lemma *setprod-dvd-setprod-subset2*:
finite B ⟹ A <= B ⟹ ALL x : A. (f x::'a::comm-semiring-1) dvd g x ⟹
setprod f A dvd setprod g B
⟨*proof*⟩

lemma *dvd-setprod*: *finite A ⟹ i:A ⟹*
(f i ::'a::comm-semiring-1) dvd setprod f A
⟨*proof*⟩

lemma *dvd-setsum* [*rule-format*]: *(ALL i : A. d dvd f i) ⟶*
(d::'a::comm-semiring-1) dvd (SUM x : A. f x)
⟨*proof*⟩

lemma *setprod-mono*:
fixes $f :: 'a \Rightarrow 'b$::*linordered-semidom*
assumes $\forall i{\in}A.\ 0 \leq f\ i \wedge f\ i \leq g\ i$
shows *setprod f A* $\leq$ *setprod g A*
⟨*proof*⟩

lemma *abs-setprod*:
fixes $f :: 'a \Rightarrow 'b$::{*linordered-field*,*abs*}
shows *abs* (*setprod f A*) = *setprod* ($\lambda x.$ *abs* (*f x*)) *A*
⟨*proof*⟩

lemma *setprod-constant*: *finite A* ==> ($\prod x{\in}$ *A*. (*y*::'*a*::{*comm-monoid-mult*}))
= *y* ^ (*card A*)
⟨*proof*⟩

lemma *setprod-gen-delta*:
assumes *fS*: *finite S*
shows *setprod* ($\lambda k.$ *if k*=*a then b k else c*) *S* = (*if a* ∈ *S then* (*b a* ::'*a*::{*comm-monoid-mult*})
∗ *c* ^ (*card S* − *1*) *else c* ^ *card S*)
⟨*proof*⟩

28.4 Versions of *inf* and *sup* on non-empty sets

no-notation *times* (**infixl** ∗ *70*)
no-notation *Groups.one* (*1*)

locale *semilattice-big* = *semilattice* +
fixes *F* :: '*a set* ⇒ '*a*
assumes *F-eq*: *finite A* ⟹ *F A* = *fold1* (*op* ∗) *A*

sublocale *semilattice-big* < *folding-one-idem* ⟨*proof*⟩

notation *times* (**infixl** ∗ *70*)
notation *Groups.one* (*1*)

context *lattice*
begin

definition *Inf-fin* :: '*a set* ⇒ '*a* ($\bigsqcap_{fin}$- [*900*] *900*) **where**
Inf-fin = *fold1 inf*

definition *Sup-fin* :: '*a set* ⇒ '*a* ($\bigsqcup_{fin}$- [*900*] *900*) **where**
Sup-fin = *fold1 sup*

end

sublocale *lattice* < *Inf-fin*!: *semilattice-big inf Inf-fin* ⟨*proof*⟩

sublocale *lattice* < *Sup-fin*!: *semilattice-big sup Sup-fin* ⟨*proof*⟩

context *semilattice-inf*
begin

lemma *ab-semigroup-idem-mult-inf*:
class.ab-semigroup-idem-mult inf
⟨*proof*⟩

lemma *fold-inf-insert*[*simp*]: *finite A* ⟹ *fold inf b* (*insert a A*) = *inf a* (*fold inf b A*)
⟨*proof*⟩

lemma *inf-le-fold-inf*: *finite A* ⟹ *ALL a*:*A*. *b* ≤ *a* ⟹ *inf b c* ≤ *fold inf c A*
⟨*proof*⟩

lemma *fold-inf-le-inf*: *finite A* ⟹ *a* ∈ *A* ⟹ *fold inf b A* ≤ *inf a b*
⟨*proof*⟩

lemma *below-fold1-iff*:
assumes *finite A A* ≠ {}
shows *x* ≤ *fold1 inf A* ⟷ (∀ *a*∈*A*. *x* ≤ *a*)
⟨*proof*⟩

lemma *fold1-belowI*:
assumes *finite A*
and *a* ∈ *A*
shows *fold1 inf A* ≤ *a*
⟨*proof*⟩

end

context *semilattice-sup*
begin

lemma *ab-semigroup-idem-mult-sup*: *class.ab-semigroup-idem-mult sup*
⟨*proof*⟩

lemma *fold-sup-insert*[*simp*]: *finite A* ⟹ *fold sup b* (*insert a A*) = *sup a* (*fold sup b A*)
⟨*proof*⟩

lemma *fold-sup-le-sup*: *finite A* ⟹ *ALL a*:*A*. *a* ≤ *b* ⟹ *fold sup c A* ≤ *sup b c*
⟨*proof*⟩

lemma *sup-le-fold-sup*: *finite A* ⟹ *a* ∈ *A* ⟹ *sup a b* ≤ *fold sup b A*
⟨*proof*⟩

end

context *lattice*
begin

lemma *Inf-le-Sup* [*simp*]: ⟦ *finite A*; $A \neq \{\}$ ⟧ ⟹ $\bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
⟨*proof*⟩

lemma *sup-Inf-absorb* [*simp*]:
finite A ⟹ $a \in A$ ⟹ *sup a* $(\bigsqcap_{fin} A) = a$
⟨*proof*⟩

lemma *inf-Sup-absorb* [*simp*]:
finite A ⟹ $a \in A$ ⟹ *inf a* $(\bigsqcup_{fin} A) = a$
⟨*proof*⟩

end

context *distrib-lattice*
begin

lemma *sup-Inf1-distrib*:
assumes *finite A*
and $A \neq \{\}$
shows *sup x* $(\bigsqcap_{fin} A) = \bigsqcap_{fin} \{sup\ x\ a | a.\ a \in A\}$
⟨*proof*⟩

lemma *sup-Inf2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *sup* $(\bigsqcap_{fin} A)$ $(\bigsqcap_{fin} B) = \bigsqcap_{fin} \{sup\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
⟨*proof*⟩

lemma *inf-Sup1-distrib*:
assumes *finite A* **and** $A \neq \{\}$
shows *inf x* $(\bigsqcup_{fin} A) = \bigsqcup_{fin} \{inf\ x\ a | a.\ a \in A\}$
⟨*proof*⟩

lemma *inf-Sup2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *inf* $(\bigsqcup_{fin} A)$ $(\bigsqcup_{fin} B) = \bigsqcup_{fin} \{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
⟨*proof*⟩

end

context *complete-lattice*
begin

lemma *Inf-fin-Inf*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A = Inf\ A$

⟨*proof*⟩

lemma *Sup-fin-Sup*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcup_{fin} A = Sup\ A$
⟨*proof*⟩

end

28.5 Versions of *min* and *max* on non-empty sets

definition (**in** *linorder*) *Min* :: *'a set* ⇒ *'a* **where**
Min = *fold1 min*

definition (**in** *linorder*) *Max* :: *'a set* ⇒ *'a* **where**
Max = *fold1 max*

sublocale *linorder* < *Min*!: *semilattice-big min Min* ⟨*proof*⟩

sublocale *linorder* < *Max*!: *semilattice-big max Max* ⟨*proof*⟩

context *linorder*
begin

lemmas *Min-singleton* = *Min.singleton*
lemmas *Max-singleton* = *Max.singleton*

lemma *Min-insert*:
assumes *finite A* **and** $A \neq \{\}$
shows *Min* (*insert x A*) = *min x* (*Min A*)
⟨*proof*⟩

lemma *Max-insert*:
assumes *finite A* **and** $A \neq \{\}$
shows *Max* (*insert x A*) = *max x* (*Max A*)
⟨*proof*⟩

lemma *Min-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows *Min* ($A \cup B$) = *min* (*Min A*) (*Min B*)
⟨*proof*⟩

lemma *Max-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows *Max* ($A \cup B$) = *max* (*Max A*) (*Max B*)
⟨*proof*⟩

lemma *hom-Min-commute*:
assumes $\bigwedge x\ y.\ h\ (min\ x\ y) = min\ (h\ x)\ (h\ y)$

and *finite N* **and** $N \neq \{\}$
shows $h\ (Min\ N) = Min\ (h\ `\ N)$
⟨*proof*⟩

lemma *hom-Max-commute*:
assumes $\bigwedge x\ y.\ h\ (max\ x\ y) = max\ (h\ x)\ (h\ y)$
and *finite N* **and** $N \neq \{\}$
shows $h\ (Max\ N) = Max\ (h\ `\ N)$
⟨*proof*⟩

lemma *ab-semigroup-idem-mult-min*:
class.ab-semigroup-idem-mult min
⟨*proof*⟩

lemma *ab-semigroup-idem-mult-max*:
class.ab-semigroup-idem-mult max
⟨*proof*⟩

lemma *max-lattice*:
class.semilattice-inf $(op \geq)\ (op >)\ max$
⟨*proof*⟩

lemma *dual-max*:
$ord.max\ (op \geq) = min$
⟨*proof*⟩

lemma *dual-min*:
$ord.min\ (op \geq) = max$
⟨*proof*⟩

lemma *strict-below-fold1-iff*:
assumes *finite A* **and** $A \neq \{\}$
shows $x < fold1\ min\ A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
⟨*proof*⟩

lemma *fold1-below-iff*:
assumes *finite A* **and** $A \neq \{\}$
shows $fold1\ min\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *fold1-strict-below-iff*:
assumes *finite A* **and** $A \neq \{\}$
shows $fold1\ min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
⟨*proof*⟩

lemma *fold1-antimono*:
assumes $A \neq \{\}$ **and** $A \subseteq B$ **and** *finite B*
shows $fold1\ min\ B \leq fold1\ min\ A$
⟨*proof*⟩

lemma *Min-in* [*simp*]:
assumes *finite* A **and** $A \neq \{\}$
shows *Min* $A \in A$
⟨*proof*⟩

lemma *Max-in* [*simp*]:
assumes *finite* A **and** $A \neq \{\}$
shows *Max* $A \in A$
⟨*proof*⟩

lemma *Min-le* [*simp*]:
assumes *finite* A **and** $x \in A$
shows *Min* $A \leq x$
⟨*proof*⟩

lemma *Max-ge* [*simp*]:
assumes *finite* A **and** $x \in A$
shows $x \leq$ *Max* A
⟨*proof*⟩

lemma *Min-ge-iff* [*simp*, *no-atp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $x \leq$ *Min* $A \longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
⟨*proof*⟩

lemma *Max-le-iff* [*simp*, *no-atp*]:
assumes *finite* A **and** $A \neq \{\}$
shows *Max* $A \leq x \longleftrightarrow (\forall\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *Min-gr-iff* [*simp*, *no-atp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $x <$ *Min* $A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
⟨*proof*⟩

lemma *Max-less-iff* [*simp*, *no-atp*]:
assumes *finite* A **and** $A \neq \{\}$
shows *Max* $A < x \longleftrightarrow (\forall\, a{\in}A.\ a < x)$
⟨*proof*⟩

lemma *Min-le-iff* [*no-atp*]:
assumes *finite* A **and** $A \neq \{\}$
shows *Min* $A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *Max-ge-iff* [*no-atp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $x \leq$ *Max* $A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$

⟨*proof*⟩

lemma *Min-less-iff* [*no-atp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
⟨*proof*⟩

lemma *Max-gr-iff* [*no-atp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x < Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$
⟨*proof*⟩

lemma *Min-eqI*:
assumes *finite A*
assumes $\bigwedge y.\ y \in A \Longrightarrow y \geq x$
and $x \in A$
shows $Min\ A = x$
⟨*proof*⟩

lemma *Max-eqI*:
assumes *finite A*
assumes $\bigwedge y.\ y \in A \Longrightarrow y \leq x$
and $x \in A$
shows $Max\ A = x$
⟨*proof*⟩

lemma *Min-antimono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Min\ N \leq Min\ M$
⟨*proof*⟩

lemma *Max-mono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Max\ M \leq Max\ N$
⟨*proof*⟩

lemma *finite-linorder-max-induct*[*consumes 1*, *case-names empty insert*]:
assumes *fin*: *finite A*
and *empty*: $P\ \{\}$
and *insert*: $(!!b\ A.\ finite\ A \Longrightarrow ALL\ a{:}A.\ a < b \Longrightarrow P\ A \Longrightarrow P(insert\ b\ A))$
shows $P\ A$
⟨*proof*⟩

lemma *finite-linorder-min-induct*[*consumes 1*, *case-names empty insert*]:
$[\![finite\ A;\ P\ \{\};\ \bigwedge b\ A.\ [\![finite\ A;\ \forall\, a{\in}A.\ b < a;\ P\ A]\!] \Longrightarrow P\ (insert\ b\ A)]\!] \Longrightarrow P\ A$
⟨*proof*⟩

end

context *linordered-ab-semigroup-add*
begin

lemma *add-Min-commute*:
fixes k
assumes *finite* N **and** $N \neq \{\}$
shows $k + Min\ N = Min\ \{k + m \mid m.\ m \in N\}$
⟨*proof*⟩

lemma *add-Max-commute*:
fixes k
assumes *finite* N **and** $N \neq \{\}$
shows $k + Max\ N = Max\ \{k + m \mid m.\ m \in N\}$
⟨*proof*⟩

end

context *linordered-ab-group-add*
begin

lemma *minus-Max-eq-Min* [*simp*]:
finite $S \Longrightarrow S \neq \{\} \Longrightarrow -\ (Max\ S) = Min\ (uminus\ `\ S)$
⟨*proof*⟩

lemma *minus-Min-eq-Max* [*simp*]:
finite $S \Longrightarrow S \neq \{\} \Longrightarrow -\ (Min\ S) = Max\ (uminus\ `\ S)$
⟨*proof*⟩

end

end

29 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *Big-Operators Relation Plain*
begin

29.1 Equivalence relations

locale *equiv* =
fixes A **and** r
assumes *refl-on*: *refl-on* 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 ⊆ r
⟨*proof*⟩

lemma *refl-on-comp-subset*: *refl-on A r ==> r ⊆ 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*⟩

29.2 Equivalence classes

lemma *equiv-class-subset*:
equiv A r ==> (a, b) ∈ r ==> r``{a} ⊆ r``{b}
— lemma for the next result
⟨*proof*⟩

theorem *equiv-class-eq*: *equiv A r ==> (a, b) ∈ r ==> r``{a} = r``{b}*
⟨*proof*⟩

lemma *equiv-class-self*: *equiv A r ==> a ∈ A ==> a ∈ r``{a}*
⟨*proof*⟩

lemma *subset-equiv-class*:
equiv A r ==> r``{b} ⊆ r``{a} ==> b ∈ A ==> (a,b) ∈ r
— lemma for the next result
⟨*proof*⟩

lemma *eq-equiv-class*:
r``{a} = r``{b} ==> equiv A r ==> b ∈ A ==> (a, b) ∈ r
⟨*proof*⟩

lemma *equiv-class-nondisjoint*:
equiv A r ==> x ∈ (r``{a} ∩ r``{b}) ==> (a, b) ∈ r
⟨*proof*⟩

lemma *equiv-type*: *equiv A r ==> r ⊆ A × A*
⟨*proof*⟩

theorem *equiv-class-eq-iff*:
equiv A r ==> ((x, y) ∈ r) = (r``{x} = r``{y} & x ∈ A & y ∈ 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*⟩

29.3 Quotients

definition *quotient* :: *'a set* ⇒ *('a × 'a) set* ⇒ *'a set set* (**infixl** *'//'* *90*) **where**
[*code del*]: $A//r = (\bigcup x \in A.\ \{r``\{x\}\})$ — set of equiv classes

lemma *quotientI*: $x \in A$ ==> $r``\{x\} \in A//r$
⟨*proof*⟩

lemma *quotientE*:
$X \in A//r$ ==> $(!!x.\ X = r``\{x\}$ ==> $x \in A$ ==> $P)$ ==> P
⟨*proof*⟩

lemma *Union-quotient*: *equiv A r* ==> *Union* $(A//r) = A$
⟨*proof*⟩

lemma *quotient-disj*:
equiv A r ==> $X \in A//r$ ==> $Y \in A//r$ ==> $X = Y \mid (X \cap Y = \{\})$
⟨*proof*⟩

lemma *quotient-eqI*:
[|*equiv A r*; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$; $(x,y) \in r$|] ==> $X = Y$
⟨*proof*⟩

lemma *quotient-eq-iff*:
[|*equiv A r*; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$|] ==> $(X = Y) = ((x,y) \in r)$
⟨*proof*⟩

lemma *eq-equiv-class-iff2*:
⟦ *equiv A r*; $x \in A$; $y \in A$ ⟧ ⟹ $(\{x\}//r = \{y\}//r) = ((x,y) : r)$
⟨*proof*⟩

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
⟨*proof*⟩

lemma *quotient-is-empty* [*iff*]: $(A//r = \{\}) = (A = \{\})$
⟨*proof*⟩

lemma *quotient-is-empty2* [*iff*]: $(\{\} = A//r) = (A = \{\})$
⟨*proof*⟩

lemma *singleton-quotient*: $\{x\}//r = \{r\ ``\ \{x\}\}$
⟨*proof*⟩

lemma *quotient-diff1*:
⟦ *inj-on* (%*a*. {*a*}//*r*) *A*; *a* ∈ *A* ⟧ ⟹ (*A* − {*a*})//*r* = *A*//*r* − {*a*}//*r*
⟨*proof*⟩

29.4 Defining unary operations upon equivalence classes

A congruence-preserving function

locale *congruent* =
fixes *r* **and** *f*
assumes *congruent*: (*y*,*z*) ∈ *r* ==> *f y* = *f z*

abbreviation
RESPECTS :: ('*a* => '*b*) => ('*a* * '*a*) *set* => *bool*
(**infixr** *respects* 80) **where**
f respects r == *congruent r f*

lemma *UN-constant-eq*: *a* ∈ *A* ==> ∀ *y* ∈ *A*. *f y* = *c* ==> (⋃ *y* ∈ *A*. *f*(*y*))=*c*
— lemma required to prove *UN-equiv-class*
⟨*proof*⟩

lemma *UN-equiv-class*:
equiv A r ==> *f respects r* ==> *a* ∈ *A*
==> (⋃ *x* ∈ *r*``{*a*}. *f x*) = *f a*
— Conversion rule
⟨*proof*⟩

lemma *UN-equiv-class-type*:
equiv A r ==> *f respects r* ==> *X* ∈ *A*//*r* ==>
(!!*x*. *x* ∈ *A* ==> *f x* ∈ *B*) ==> (⋃ *x* ∈ *X*. *f x*) ∈ *B*
⟨*proof*⟩

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be !!*y*. *y* ∈ *A* ==> *f y* ∈ *B*.

lemma *UN-equiv-class-inject*:
equiv A r ==> *f respects r* ==>
(⋃ *x* ∈ *X*. *f x*) = (⋃ *y* ∈ *Y*. *f y*) ==> *X* ∈ *A*//*r* ==> *Y* ∈ *A*//*r*
==> (!!*x y*. *x* ∈ *A* ==> *y* ∈ *A* ==> *f x* = *f y* ==> (*x*, *y*) ∈ *r*)
==> *X* = *Y*
⟨*proof*⟩

29.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

locale *congruent2* =
fixes *r1* **and** *r2* **and** *f*
assumes *congruent2*:

$(y1,z1) \in r1 \Longrightarrow (y2,z2) \in r2 \Longrightarrow f\ y1\ y2 = f\ z1\ z2$

Abbreviation for the common case where the relations are identical

abbreviation
RESPECTS2:: $['a \Rightarrow 'a \Rightarrow 'b, ('a * 'a)\ set] \Rightarrow bool$
(**infixr** *respects2 80*) **where**
$f\ respects2\ r == congruent2\ r\ r\ f$

lemma *congruent2-implies-congruent*:
$equiv\ A\ r1 \Longrightarrow congruent2\ r1\ r2\ f \Longrightarrow a \in A \Longrightarrow congruent\ r2\ (f\ a)$
⟨*proof*⟩

lemma *congruent2-implies-congruent-UN*:
$equiv\ A1\ r1 \Longrightarrow equiv\ A2\ r2 \Longrightarrow congruent2\ r1\ r2\ f \Longrightarrow a \in A2 \Longrightarrow$
$congruent\ r1\ (\lambda x1.\ \bigcup x2 \in r2``\{a\}.\ f\ x1\ x2)$
⟨*proof*⟩

lemma *UN-equiv-class2*:
$equiv\ A1\ r1 \Longrightarrow equiv\ A2\ r2 \Longrightarrow congruent2\ r1\ r2\ f \Longrightarrow a1 \in A1 \Longrightarrow a2 \in A2$
$\Longrightarrow (\bigcup x1 \in r1``\{a1\}.\ \bigcup x2 \in r2``\{a2\}.\ f\ x1\ x2) = f\ a1\ a2$
⟨*proof*⟩

lemma *UN-equiv-class-type2*:
$equiv\ A1\ r1 \Longrightarrow equiv\ A2\ r2 \Longrightarrow congruent2\ r1\ r2\ f$
$\Longrightarrow X1 \in A1//r1 \Longrightarrow X2 \in A2//r2$
$\Longrightarrow (!!x1\ x2.\ x1 \in A1 \Longrightarrow x2 \in A2 \Longrightarrow f\ x1\ x2 \in B)$
$\Longrightarrow (\bigcup x1 \in X1.\ \bigcup x2 \in X2.\ f\ x1\ x2) \in B$
⟨*proof*⟩

lemma *UN-UN-split-split-eq*:
$(\bigcup (x1,\ x2) \in X.\ \bigcup (y1,\ y2) \in Y.\ A\ x1\ x2\ y1\ y2) =$
$(\bigcup x \in X.\ \bigcup y \in Y.\ (\lambda(x1,\ x2).\ (\lambda(y1,\ y2).\ A\ x1\ x2\ y1\ y2)\ y)\ x)$
— Allows a natural expression of binary operators,
— without explicit calls to *split*
⟨*proof*⟩

lemma *congruent2I*:
$equiv\ A1\ r1 \Longrightarrow equiv\ A2\ r2$
$\Longrightarrow (!!y\ z\ w.\ w \in A2 \Longrightarrow (y,z) \in r1 \Longrightarrow f\ y\ w = f\ z\ w)$
$\Longrightarrow (!!y\ z\ w.\ w \in A1 \Longrightarrow (y,z) \in r2 \Longrightarrow f\ w\ y = f\ w\ z)$
$\Longrightarrow congruent2\ r1\ r2\ f$
— 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*⟩

29.6 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*: *finite A ==> r ⊆ A × A ==> finite (A//r)*
— recall *equiv ?A ?r ⟹ ?r ⊆ ?A × ?A*
⟨*proof*⟩

lemma *finite-equiv-class*:
finite A ==> r ⊆ A × A ==> X ∈ A//r ==> finite X
⟨*proof*⟩

lemma *equiv-imp-dvd-card*:
finite A ==> equiv A r ==> ∀ X ∈ A//r. k dvd card X
==> k dvd card A
⟨*proof*⟩

lemma *card-quotient-disjoint*:
⟦ *finite A*; *inj-on* (λ*x*. {*x*} // *r*) *A* ⟧ ⟹ *card*(*A*//*r*) = *card A*
⟨*proof*⟩

end

30 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *Int*
imports *Equiv-Relations Nat Wellfounded*
uses
(*Tools/numeral.ML*)
(*Tools/numeral-syntax.ML*)
(*Tools/int-arith.ML*)
begin

30.1 The equivalence relation underlying the integers

definition *intrel* :: ((*nat* × *nat*) × (*nat* × *nat*)) *set* **where**
[*code del*]: *intrel* = {((*x*, *y*), (*u*, *v*)) | *x y u v*. *x* + *v* = *u* + *y* }

typedef (*Integ*)
int = *UNIV*//*intrel*
⟨*proof*⟩

instantiation *int* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *ord*, *abs*, *sgn*}
begin

definition
Zero-int-def [*code del*]: *0 = Abs-Integ* (*intrel* `` {(*0*, *0*)})

definition
One-int-def [*code del*]: *1 = Abs-Integ* (*intrel* `` {(*1*, *0*)})

definition
add-int-def [*code del*]: *z + w = Abs-Integ*
(⋃(*x*, *y*) ∈ *Rep-Integ z*. ⋃(*u*, *v*) ∈ *Rep-Integ w*.
intrel `` {(*x + u*, *y + v*)})

definition
minus-int-def [*code del*]:
− z = Abs-Integ (⋃(*x*, *y*) ∈ *Rep-Integ z*. *intrel* `` {(*y*, *x*)})

definition
diff-int-def [*code del*]: *z − w = z + (−w :: int)*

definition
mult-int-def [*code del*]: *z ∗ w = Abs-Integ*
(⋃(*x*, *y*) ∈ *Rep-Integ z*. ⋃(*u*,*v*) ∈ *Rep-Integ w*.
intrel `` {(*x∗u + y∗v*, *x∗v + y∗u*)})

definition
le-int-def [*code del*]:
z ≤ *w* ⟷ (∃ *x y u v*. *x+v* ≤ *u+y* ∧ (*x*, *y*) ∈ *Rep-Integ z* ∧ (*u*, *v*) ∈ *Rep-Integ w*)

definition
less-int-def [*code del*]: (*z*::*int*) < *w* ⟷ *z* ≤ *w* ∧ *z* ≠ *w*

definition
zabs-def: |*i*::*int*| = (*if i < 0 then − i else i*)

definition
zsgn-def: *sgn* (*i*::*int*) = (*if i=0 then 0 else if 0<i then 1 else − 1*)

instance ⟨*proof*⟩

end

30.2 Construction of the Integers

lemma *intrel-iff* [*simp*]: (((*x*,*y*),(*u*,*v*)) ∈ *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*) ∈ *intrel*)

lemmas *equiv-intrel-iff* [*simp*] = *eq-equiv-class-iff* [*OF equiv-intrel UNIV-I UNIV-I*]

All equivalence classes belong to set of representatives

lemma [*simp*]: *intrel*``{(*x*,*y*)} ∈ *Integ*
⟨*proof*⟩

Reduces equality on abstractions to equality on representatives: ⟦*x* ∈ *Integ*; *y* ∈ *Integ*⟧ ⟹ (*Abs-Integ x* = *Abs-Integ y*) = (*x* = *y*)

declare *Abs-Integ-inject* [*simp*,*no-atp*] *Abs-Integ-inverse* [*simp*,*no-atp*]

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*⟩

30.3 Arithmetic Operations

lemma *minus*: − *Abs-Integ*(*intrel*``{(*x*,*y*)}) = *Abs-Integ*(*intrel* `` {(*y*,*x*)})
⟨*proof*⟩

lemma *add*:
Abs-Integ (*intrel*``{(*x*,*y*)}) + *Abs-Integ* (*intrel*``{(*u*,*v*)}) =
Abs-Integ (*intrel*``{(*x*+*u*, *y*+*v*)})
⟨*proof*⟩

Congruence property for multiplication

lemma *mult-congruent2*:
(%*p1 p2*. (%(*x*,*y*). (%(*u*,*v*). *intrel*``{(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)}) *p2*) *p1*)
respects2 intrel
⟨*proof*⟩

lemma *mult*:
Abs-Integ((*intrel*``{(*x*,*y*)})) ∗ *Abs-Integ*((*intrel*``{(*u*,*v*)})) =
Abs-Integ(*intrel* `` {(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)})
⟨*proof*⟩

The integers form a *comm-ring-1*

instance *int* :: *comm-ring-1*
⟨*proof*⟩

lemma *int-def*: *of-nat m* = *Abs-Integ* (*intrel* `` {(*m*, *0*)})
⟨*proof*⟩

30.4 The $\leq$ Ordering

lemma *le*:
(*Abs-Integ*(*intrel*``{(*x*,*y*)}) $\leq$ *Abs-Integ*(*intrel*``{(*u*,*v*)})) = (*x*+*v* $\leq$ *u*+*y*)
⟨*proof*⟩

lemma *less*:
(*Abs-Integ*(*intrel*``{(*x*,*y*)}) < *Abs-Integ*(*intrel*``{(*u*,*v*)})) = (*x*+*v* < *u*+*y*)
⟨*proof*⟩

instance *int* :: *linorder*
⟨*proof*⟩

instantiation *int* :: *distrib-lattice*
begin

definition
(*inf* :: *int* ⇒ *int* ⇒ *int*) = *min*

definition
(*sup* :: *int* ⇒ *int* ⇒ *int*) = *max*

instance
⟨*proof*⟩

end

instance *int* :: *ordered-cancel-ab-semigroup-add*
⟨*proof*⟩

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on k¿0

lemma *zmult-zless-mono2-lemma*:
(*i*::*int*)<*j* ==> 0<*k* ==> *of-nat k* ∗ *i* < *of-nat k* ∗ *j*
⟨*proof*⟩

lemma *zero-le-imp-eq-int*: (0::*int*) $\leq$ *k* ==> ∃ *n*. *k* = *of-nat n*
⟨*proof*⟩

lemma *zero-less-imp-eq-int*: (0::*int*) < *k* ==> ∃ *n*>0. *k* = *of-nat n*
⟨*proof*⟩

lemma *zmult-zless-mono2*: [| *i*<*j*; (0::*int*) < *k* |] ==> *k*∗*i* < *k*∗*j*
⟨*proof*⟩

The integers form an ordered integral domain

instance *int* :: *linordered-idom*
⟨*proof*⟩

lemma *zless-imp-add1-zle*: $w < z \Longrightarrow w + (1::int) \leq z$
⟨*proof*⟩

lemma *zless-iff-Suc-zadd*:
$(w :: int) < z \longleftrightarrow (\exists n.\ z = w + of\text{-}nat\ (Suc\ n))$
⟨*proof*⟩

lemmas *int-distrib* =
left-distrib [*of z1*::*int z2 w*, *standard*]
right-distrib [*of w*::*int z1 z2*, *standard*]
left-diff-distrib [*of z1*::*int z2 w*, *standard*]
right-diff-distrib [*of w*::*int z1 z2*, *standard*]

30.5 Embedding of the Integers into any *ring-1*: *of-int*

context *ring-1*
begin

definition *of-int* :: *int* ⇒ *'a* **where**
[*code del*]: *of-int z* = *contents* ($\bigcup(i, j) \in Rep\text{-}Integ\ z.$ { *of-nat i* − *of-nat j* })

lemma *of-int*: *of-int* (*Abs-Integ* (*intrel* `` {(*i*,*j*)})) = *of-nat i* − *of-nat j*
⟨*proof*⟩

lemma *of-int-0* [*simp*]: *of-int 0* = *0*
⟨*proof*⟩

lemma *of-int-1* [*simp*]: *of-int 1* = *1*
⟨*proof*⟩

lemma *of-int-add* [*simp*]: *of-int* (*w*+*z*) = *of-int w* + *of-int z*
⟨*proof*⟩

lemma *of-int-minus* [*simp*]: *of-int* (−*z*) = − (*of-int z*)
⟨*proof*⟩

lemma *of-int-diff* [*simp*]: *of-int* (*w* − *z*) = *of-int w* − *of-int z*
⟨*proof*⟩

lemma *of-int-mult* [*simp*]: *of-int* (*w*∗*z*) = *of-int w* ∗ *of-int z*
⟨*proof*⟩

Collapse nested embeddings

lemma *of-int-of-nat-eq* [*simp*]: *of-int* (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *of-int-power*:
of-int (*z* ^ *n*) = *of-int z* ^ *n*
⟨*proof*⟩

end

Class for unital rings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *ring-char-0* = *ring-1* + *semiring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]:
of-int w = *of-int z* ⟷ *w* = *z*
⟨*proof*⟩

Special cases where either operand is zero

lemma *of-int-eq-0-iff* [*simp*]:
of-int z = *0* ⟷ *z* = *0*
⟨*proof*⟩

lemma *of-int-0-eq-iff* [*simp*]:
0 = *of-int z* ⟷ *z* = *0*
⟨*proof*⟩

end

context *linordered-idom*
begin

Every *linordered-idom* has characteristic zero.

subclass *ring-char-0* ⟨*proof*⟩

lemma *of-int-le-iff* [*simp*]:
of-int w ≤ *of-int z* ⟷ *w* ≤ *z*
⟨*proof*⟩

lemma *of-int-less-iff* [*simp*]:
of-int w < *of-int z* ⟷ *w* < *z*
⟨*proof*⟩

lemma *of-int-0-le-iff* [*simp*]:
0 ≤ *of-int z* ⟷ *0* ≤ *z*
⟨*proof*⟩

lemma *of-int-le-0-iff* [*simp*]:
of-int z ≤ *0* ⟷ *z* ≤ *0*
⟨*proof*⟩

lemma *of-int-0-less-iff* [*simp*]:
0 < *of-int z* ⟷ *0* < *z*
⟨*proof*⟩

lemma *of-int-less-0-iff* [*simp*]:
of-int z < *0* ⟷ *z* < *0*
⟨*proof*⟩

end

lemma *of-int-eq-id* [*simp*]: *of-int* = *id*
⟨*proof*⟩

30.6 Magnitude of an Integer, as a Natural Number: *nat*

definition
nat :: *int* ⇒ *nat*
where
[*code del*]: *nat z* = *contents* (⋃(*x*, *y*) ∈ *Rep-Integ z*. {*x*−*y*})

lemma *nat*: *nat* (*Abs-Integ* (*intrel*``{(*x*,*y*)})) = *x*−*y*
⟨*proof*⟩

lemma *nat-int* [*simp*]: *nat* (*of-nat n*) = *n*
⟨*proof*⟩

lemma *nat-zero* [*simp*]: *nat 0* = *0*
⟨*proof*⟩

lemma *int-nat-eq* [*simp*]: *of-nat* (*nat z*) = (*if 0* ≤ *z then z else 0*)
⟨*proof*⟩

corollary *nat-0-le*: *0* ≤ *z* ==> *of-nat* (*nat z*) = *z*
⟨*proof*⟩

lemma *nat-le-0* [*simp*]: *z* ≤ *0* ==> *nat z* = *0*
⟨*proof*⟩

lemma *nat-le-eq-zle*: *0* < *w* | *0* ≤ *z* ==> (*nat w* ≤ *nat z*) = (*w*≤*z*)
⟨*proof*⟩

An alternative condition is (*0*::′*a*) ≤ *w*

corollary *nat-mono-iff*: *0* < *z* ==> (*nat w* < *nat z*) = (*w* < *z*)
⟨*proof*⟩

corollary *nat-less-eq-zless*: *0* ≤ *w* ==> (*nat w* < *nat z*) = (*w*<*z*)
⟨*proof*⟩

lemma *zless-nat-conj* [*simp*]: (*nat w* < *nat z*) = (*0* < *z* & *w* < *z*)
⟨*proof*⟩

lemma *nonneg-eq-int*:

fixes $z :: int$
assumes $0 \leq z$ **and** $\bigwedge m.\ z = of\text{-}nat\ m \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *nat-eq-iff*: $(nat\ w = m) = (if\ 0 \leq w\ then\ w = of\text{-}nat\ m\ else\ m{=}0)$
⟨*proof*⟩

corollary *nat-eq-iff2*: $(m = nat\ w) = (if\ 0 \leq w\ then\ w = of\text{-}nat\ m\ else\ m{=}0)$
⟨*proof*⟩

lemma *nat-less-iff*: $0 \leq w \Longrightarrow (nat\ w < m) = (w < of\text{-}nat\ m)$
⟨*proof*⟩

lemma *nat-0-iff* [*simp*]: $nat(i{::}int) = 0 \longleftrightarrow i{\leq}0$
⟨*proof*⟩

lemma *int-eq-iff*: $(of\text{-}nat\ m = z) = (m = nat\ z\ \&\ 0 \leq z)$
⟨*proof*⟩

lemma *zero-less-nat-eq* [*simp*]: $(0 < nat\ z) = (0 < z)$
⟨*proof*⟩

lemma *nat-add-distrib*:
$[\!|\ (0{::}int) \leq z;\ \ 0 \leq z'\ |\!] \Longrightarrow nat\ (z{+}z') = nat\ z + nat\ z'$
⟨*proof*⟩

lemma *nat-diff-distrib*:
$[\!|\ (0{::}int) \leq z';\ \ z' \leq z\ |\!] \Longrightarrow nat\ (z{-}z') = nat\ z - nat\ z'$
⟨*proof*⟩

lemma *nat-zminus-int* [*simp*]: $nat\ (-\ (of\text{-}nat\ n)) = 0$
⟨*proof*⟩

lemma *zless-nat-eq-int-zless*: $(m < nat\ z) = (of\text{-}nat\ m < z)$
⟨*proof*⟩

context *ring-1*
begin

lemma *of-nat-nat*: $0 \leq z \Longrightarrow of\text{-}nat\ (nat\ z) = of\text{-}int\ z$
⟨*proof*⟩

end

For termination proofs:

lemma *measure-function-int*[*measure-function*]: *is-measure* $(nat\ o\ abs)$ ⟨*proof*⟩

30.7 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: $-\ (\mathit{of\text{-}nat}\ (\mathit{Suc}\ n)) < (0 :: \mathit{int})$
⟨*proof*⟩

lemma *negative-zless* [*iff*]: $-\ (\mathit{of\text{-}nat}\ (\mathit{Suc}\ n)) < (\mathit{of\text{-}nat}\ m :: \mathit{int})$
⟨*proof*⟩

lemma *negative-zle-0*: $-\ \mathit{of\text{-}nat}\ n \leq (0 :: \mathit{int})$
⟨*proof*⟩

lemma *negative-zle* [*iff*]: $-\ \mathit{of\text{-}nat}\ n \leq (\mathit{of\text{-}nat}\ m :: \mathit{int})$
⟨*proof*⟩

lemma *not-zle-0-negative* [*simp*]: $\sim\ (0 \leq -\ (\mathit{of\text{-}nat}\ (\mathit{Suc}\ n) :: \mathit{int}))$
⟨*proof*⟩

lemma *int-zle-neg*: $((\mathit{of\text{-}nat}\ n :: \mathit{int}) \leq -\ \mathit{of\text{-}nat}\ m) = (n = 0\ \&\ m = 0)$
⟨*proof*⟩

lemma *not-int-zless-negative* [*simp*]: $\sim\ ((\mathit{of\text{-}nat}\ n :: \mathit{int}) < -\ \mathit{of\text{-}nat}\ m)$
⟨*proof*⟩

lemma *negative-eq-positive* [*simp*]: $((-\ \mathit{of\text{-}nat}\ n :: \mathit{int}) = \mathit{of\text{-}nat}\ m) = (n = 0\ \&\ m = 0)$
⟨*proof*⟩

lemma *zle-iff-zadd*: $(w::\mathit{int}) \leq z \longleftrightarrow (\exists\, n.\ z = w + \mathit{of\text{-}nat}\ n)$
⟨*proof*⟩

lemma *zadd-int-left*: $\mathit{of\text{-}nat}\ m + (\mathit{of\text{-}nat}\ n + z) = \mathit{of\text{-}nat}\ (m + n) + (z::\mathit{int})$
⟨*proof*⟩

lemma *int-Suc0-eq-1*: $\mathit{of\text{-}nat}\ (\mathit{Suc}\ 0) = (1::\mathit{int})$
⟨*proof*⟩

This version is proved for all ordered rings, not just integers! It is proved here because attribute *arith-split* is not available in theory *Rings*. But is it really better than just rewriting with *abs-if*?

lemma *abs-split* [*arith-split*,*no-atp*]:
$P(\mathit{abs}(a::'a::\mathit{linordered\text{-}idom})) = ((0 \leq a \longrightarrow P\ a)\ \&\ (a < 0 \longrightarrow P(-a)))$
⟨*proof*⟩

lemma *negD*: $(x :: \mathit{int}) < 0 \Longrightarrow \exists\, n.\ x = -\ (\mathit{of\text{-}nat}\ (\mathit{Suc}\ n))$
⟨*proof*⟩

30.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

theorem *int-cases* [*cases type*: *int*, *case-names nonneg neg*]:
[|!! *n*. (*z* :: *int*) = *of-nat n* ==> *P*; !! *n*. *z* = − (*of-nat* (*Suc n*)) ==> *P* |] ==> *P*
⟨*proof*⟩

theorem *int-of-nat-induct* [*induct type*: *int*, *case-names nonneg neg*]:
[|!! *n*. *P* (*of-nat n* :: *int*); !!*n*. *P* (− (*of-nat* (*Suc n*))) |] ==> *P z*
⟨*proof*⟩

Contributed by Brian Huffman

theorem *int-diff-cases*:
obtains (*diff*) *m n* **where** (*z*::*int*) = *of-nat m* − *of-nat n*
⟨*proof*⟩

30.9 Binary representation

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Tools/twos−compl.ML*, function *int-of-binary*, for the numerical interpretation.

The representation expects that (*m mod 2*) is 0 or 1, even if m is negative; For instance, *−5 div 2* = *−3* and *−5 mod 2* = *1*; thus *−5* = (*−3*)∗*2* + *1*.

This two's complement binary representation derives from the paper "An Efficient Representation of Arithmetic for Term Rewriting" by Dave Cohen and Phil Watson, Rewriting Techniques and Applications, Springer LNCS 488 (240-251), 1991.

30.9.1 The constructors *Bit0*, *Bit1*, *Pls* and *Min*

definition
Pls :: *int* **where**
[*code del*]: *Pls* = *0*

definition
Min :: *int* **where**
[*code del*]: *Min* = − *1*

definition
Bit0 :: *int* ⇒ *int* **where**
[*code del*]: *Bit0 k* = *k* + *k*

definition
Bit1 :: *int* ⇒ *int* **where**

[*code del*]: *Bit1 k = 1 + k + k*

class *number* = — for numeric types: nat, int, real, ...
fixes *number-of* :: *int* ⇒ *'a*

⟨*ML*⟩

syntax
-Numeral :: *num-const* ⇒ *'a* (-)

⟨*ML*⟩

abbreviation
Numeral0 ≡ *number-of Pls*

abbreviation
Numeral1 ≡ *number-of* (*Bit1 Pls*)

lemma *Let-number-of* [*simp*]: *Let* (*number-of v*) *f* = *f* (*number-of v*)
— Unfold all *let*s involving constants
⟨*proof*⟩

definition
succ :: *int* ⇒ *int* **where**
[*code del*]: *succ k = k + 1*

definition
pred :: *int* ⇒ *int* **where**
[*code del*]: *pred k = k − 1*

lemmas
max-number-of [*simp*] = *max-def*
[*of number-of u number-of v*, *standard*]
and
min-number-of [*simp*] = *min-def*
[*of number-of u number-of v*, *standard*]
— unfolding *minx* and *max* on numerals

lemmas *numeral-simps* =
succ-def pred-def Pls-def Min-def Bit0-def Bit1-def

Removal of leading zeroes

lemma *Bit0-Pls* [*simp*, *code-post*]:
Bit0 Pls = Pls
⟨*proof*⟩

lemma *Bit1-Min* [*simp*, *code-post*]:
Bit1 Min = Min
⟨*proof*⟩

lemmas *normalize-bin-simps* =
Bit0-Pls Bit1-Min

30.9.2 Successor and predecessor functions

Successor

lemma *succ-Pls*:
succ Pls = *Bit1 Pls*
⟨*proof*⟩

lemma *succ-Min*:
succ Min = *Pls*
⟨*proof*⟩

lemma *succ-Bit0*:
succ (*Bit0 k*) = *Bit1 k*
⟨*proof*⟩

lemma *succ-Bit1*:
succ (*Bit1 k*) = *Bit0* (*succ k*)
⟨*proof*⟩

lemmas *succ-bin-simps* [*simp*] =
succ-Pls succ-Min succ-Bit0 succ-Bit1

Predecessor

lemma *pred-Pls*:
pred Pls = *Min*
⟨*proof*⟩

lemma *pred-Min*:
pred Min = *Bit0 Min*
⟨*proof*⟩

lemma *pred-Bit0*:
pred (*Bit0 k*) = *Bit1* (*pred k*)
⟨*proof*⟩

lemma *pred-Bit1*:
pred (*Bit1 k*) = *Bit0 k*
⟨*proof*⟩

lemmas *pred-bin-simps* [*simp*] =
pred-Pls pred-Min pred-Bit0 pred-Bit1

30.9.3 Binary arithmetic

Addition

lemma *add-Pls*:
Pls + k = k
⟨*proof*⟩

lemma *add-Min*:
Min + k = pred k
⟨*proof*⟩

lemma *add-Bit0-Bit0*:
(Bit0 k) + (Bit0 l) = Bit0 (k + l)
⟨*proof*⟩

lemma *add-Bit0-Bit1*:
(Bit0 k) + (Bit1 l) = Bit1 (k + l)
⟨*proof*⟩

lemma *add-Bit1-Bit0*:
(Bit1 k) + (Bit0 l) = Bit1 (k + l)
⟨*proof*⟩

lemma *add-Bit1-Bit1*:
(Bit1 k) + (Bit1 l) = Bit0 (k + succ l)
⟨*proof*⟩

lemma *add-Pls-right*:
k + Pls = k
⟨*proof*⟩

lemma *add-Min-right*:
k + Min = pred k
⟨*proof*⟩

lemmas *add-bin-simps* [*simp*] =
add-Pls add-Min add-Pls-right add-Min-right
add-Bit0-Bit0 add-Bit0-Bit1 add-Bit1-Bit0 add-Bit1-Bit1

Negation

lemma *minus-Pls*:
− Pls = Pls
⟨*proof*⟩

lemma *minus-Min*:
− Min = Bit1 Pls
⟨*proof*⟩

lemma *minus-Bit0*:
− (Bit0 k) = Bit0 (− k)
⟨*proof*⟩

lemma *minus-Bit1*:
$-\ (Bit1\ k) = Bit1\ (pred\ (-\ k))$
⟨*proof*⟩

lemmas *minus-bin-simps* [*simp*] =
minus-Pls minus-Min minus-Bit0 minus-Bit1

Subtraction

lemma *diff-bin-simps* [*simp*]:
$k - Pls = k$
$k - Min = succ\ k$
$Pls - (Bit0\ l) = Bit0\ (Pls - l)$
$Pls - (Bit1\ l) = Bit1\ (Min - l)$
$Min - (Bit0\ l) = Bit1\ (Min - l)$
$Min - (Bit1\ l) = Bit0\ (Min - l)$
$(Bit0\ k) - (Bit0\ l) = Bit0\ (k - l)$
$(Bit0\ k) - (Bit1\ l) = Bit1\ (pred\ k - l)$
$(Bit1\ k) - (Bit0\ l) = Bit1\ (k - l)$
$(Bit1\ k) - (Bit1\ l) = Bit0\ (k - l)$
⟨*proof*⟩

Multiplication

lemma *mult-Pls*:
$Pls * w = Pls$
⟨*proof*⟩

lemma *mult-Min*:
$Min * k = -\ k$
⟨*proof*⟩

lemma *mult-Bit0*:
$(Bit0\ k) * l = Bit0\ (k * l)$
⟨*proof*⟩

lemma *mult-Bit1*:
$(Bit1\ k) * l = (Bit0\ (k * l)) + l$
⟨*proof*⟩

lemmas *mult-bin-simps* [*simp*] =
mult-Pls mult-Min mult-Bit0 mult-Bit1

30.9.4 Binary comparisons

Preliminaries

lemma *even-less-0-iff*:
$a + a < 0 \longleftrightarrow a < (0::'a::linordered\text{-}idom)$
⟨*proof*⟩

lemma *le-imp-0-less*:
assumes *le*: $0 \leq z$
shows $(0::int) < 1 + z$
⟨*proof*⟩

lemma *odd-less-0-iff*:
$(1 + z + z < 0) = (z < (0::int))$
⟨*proof*⟩

lemma *bin-less-0-simps*:
$Pls < 0 \longleftrightarrow False$
$Min < 0 \longleftrightarrow True$
$Bit0\ w < 0 \longleftrightarrow w < 0$
$Bit1\ w < 0 \longleftrightarrow w < 0$
⟨*proof*⟩

lemma *less-bin-lemma*: $k < l \longleftrightarrow k - l < (0::int)$
⟨*proof*⟩

lemma *le-iff-pred-less*: $k \leq l \longleftrightarrow pred\ k < l$
⟨*proof*⟩

lemma *succ-pred*: $succ\ (pred\ x) = x$
⟨*proof*⟩

Less-than

lemma *less-bin-simps* [*simp*]:
$Pls < Pls \longleftrightarrow False$
$Pls < Min \longleftrightarrow False$
$Pls < Bit0\ k \longleftrightarrow Pls < k$
$Pls < Bit1\ k \longleftrightarrow Pls \leq k$
$Min < Pls \longleftrightarrow True$
$Min < Min \longleftrightarrow False$
$Min < Bit0\ k \longleftrightarrow Min < k$
$Min < Bit1\ k \longleftrightarrow Min < k$
$Bit0\ k < Pls \longleftrightarrow k < Pls$
$Bit0\ k < Min \longleftrightarrow k \leq Min$
$Bit1\ k < Pls \longleftrightarrow k < Pls$
$Bit1\ k < Min \longleftrightarrow k < Min$
$Bit0\ k < Bit0\ l \longleftrightarrow k < l$
$Bit0\ k < Bit1\ l \longleftrightarrow k \leq l$
$Bit1\ k < Bit0\ l \longleftrightarrow k < l$
$Bit1\ k < Bit1\ l \longleftrightarrow k < l$
⟨*proof*⟩

Less-than-or-equal

lemma *le-bin-simps* [*simp*]:
$Pls \leq Pls \longleftrightarrow True$
$Pls \leq Min \longleftrightarrow False$

$Pls \leq Bit0\ k \longleftrightarrow Pls \leq k$
$Pls \leq Bit1\ k \longleftrightarrow Pls \leq k$
$Min \leq Pls \longleftrightarrow True$
$Min \leq Min \longleftrightarrow True$
$Min \leq Bit0\ k \longleftrightarrow Min < k$
$Min \leq Bit1\ k \longleftrightarrow Min \leq k$
$Bit0\ k \leq Pls \longleftrightarrow k \leq Pls$
$Bit0\ k \leq Min \longleftrightarrow k \leq Min$
$Bit1\ k \leq Pls \longleftrightarrow k < Pls$
$Bit1\ k \leq Min \longleftrightarrow k \leq Min$
$Bit0\ k \leq Bit0\ l \longleftrightarrow k \leq l$
$Bit0\ k \leq Bit1\ l \longleftrightarrow k \leq l$
$Bit1\ k \leq Bit0\ l \longleftrightarrow k < l$
$Bit1\ k \leq Bit1\ l \longleftrightarrow k \leq l$
⟨*proof*⟩

Equality

lemma *eq-bin-simps* [*simp*]:
$Pls = Pls \longleftrightarrow True$
$Pls = Min \longleftrightarrow False$
$Pls = Bit0\ l \longleftrightarrow Pls = l$
$Pls = Bit1\ l \longleftrightarrow False$
$Min = Pls \longleftrightarrow False$
$Min = Min \longleftrightarrow True$
$Min = Bit0\ l \longleftrightarrow False$
$Min = Bit1\ l \longleftrightarrow Min = l$
$Bit0\ k = Pls \longleftrightarrow k = Pls$
$Bit0\ k = Min \longleftrightarrow False$
$Bit1\ k = Pls \longleftrightarrow False$
$Bit1\ k = Min \longleftrightarrow k = Min$
$Bit0\ k = Bit0\ l \longleftrightarrow k = l$
$Bit0\ k = Bit1\ l \longleftrightarrow False$
$Bit1\ k = Bit0\ l \longleftrightarrow False$
$Bit1\ k = Bit1\ l \longleftrightarrow k = l$
⟨*proof*⟩

30.10 Converting Numerals to Rings: *number-of*

class *number-ring* = *number* + *comm-ring-1* +
assumes *number-of-eq*: $number\text{-}of\ k = of\text{-}int\ k$

self-embedding of the integers

instantiation *int* :: *number-ring*
begin

definition *int-number-of-def* [*code del*]:
$number\text{-}of\ w = (of\text{-}int\ w :: int)$

instance ⟨*proof*⟩

end

lemma *number-of-is-id*:
number-of (*k*::*int*) = *k*
⟨*proof*⟩

lemma *number-of-succ*:
number-of (*succ k*) = (*1* + *number-of k* ::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-pred*:
number-of (*pred w*) = (− *1* + *number-of w* ::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-minus*:
number-of (*uminus w*) = (− (*number-of w*)::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-add*:
number-of (*v* + *w*) = (*number-of v* + *number-of w*::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-diff*:
number-of (*v* − *w*) = (*number-of v* − *number-of w*::'*a*::*number-ring*)
⟨*proof*⟩

lemma *number-of-mult*:
number-of (*v* ∗ *w*) = (*number-of v* ∗ *number-of w*::'*a*::*number-ring*)
⟨*proof*⟩

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

lemma *double-number-of-Bit0*:
(*1* + *1*) ∗ *number-of w* = (*number-of* (*Bit0 w*) ::'*a*::*number-ring*)
⟨*proof*⟩

Converting numerals 0 and 1 to their abstract versions.

lemma *numeral-0-eq-0* [*simp*, *code-post*]:
Numeral0 = (*0*::'*a*::*number-ring*)
⟨*proof*⟩

lemma *numeral-1-eq-1* [*simp*, *code-post*]:
Numeral1 = (*1*::'*a*::*number-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*]:
$-\ (number\text{-}of\ w) * z = number\text{-}of\ (uminus\ w) * (z::'a::number\text{-}ring)$
⟨*proof*⟩

Subtraction

lemma *diff-number-of-eq*:
number-of v − number-of w =
$(number\text{-}of\ (v + uminus\ w)::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *number-of-Pls*:
number-of Pls = $(0::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *number-of-Min*:
number-of Min = $(-\ 1::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *number-of-Bit0*:
$number\text{-}of\ (Bit0\ w) = (0::'a::number\text{-}ring) + (number\text{-}of\ w) + (number\text{-}of\ w)$
⟨*proof*⟩

lemma *number-of-Bit1*:
$number\text{-}of\ (Bit1\ w) = (1::'a::number\text{-}ring) + (number\text{-}of\ w) + (number\text{-}of\ w)$
⟨*proof*⟩

30.10.1 Equality of Binary Numbers

First version by Norbert Voelker

definition *iszero* :: $'a::semiring\text{-}1 \Rightarrow bool$ **where**
$iszero\ z \longleftrightarrow z = 0$

lemma *iszero-0*: *iszero 0*
⟨*proof*⟩

lemma *iszero-Numeral0*: $iszero\ (Numeral0 :: 'a::number\text{-}ring)$

⟨*proof*⟩

lemma *not-iszero-1*: ¬ *iszero 1*
⟨*proof*⟩

lemma *not-iszero-Numeral1*: ¬ *iszero* (*Numeral1* :: *'a*::*number-ring*)
⟨*proof*⟩

lemma *eq-number-of-eq* [*simp*]:
((*number-of x*::*'a*::*number-ring*) = *number-of y*) =
iszero (*number-of* (*x* + *uminus y*) :: *'a*)
⟨*proof*⟩

lemma *iszero-number-of-Pls*:
iszero ((*number-of Pls*)::*'a*::*number-ring*)
⟨*proof*⟩

lemma *nonzero-number-of-Min*:
~ *iszero* ((*number-of Min*)::*'a*::*number-ring*)
⟨*proof*⟩

30.10.2 Comparisons, for Ordered Rings

lemmas *double-eq-0-iff* = *double-zero*

lemma *odd-nonzero*:
1 + *z* + *z* ≠ (*0*::*int*)
⟨*proof*⟩

lemma *iszero-number-of-Bit0*:
iszero (*number-of* (*Bit0 w*)::*'a*) =
iszero (*number-of w*::*'a*::{*ring-char-0*,*number-ring*})
⟨*proof*⟩

lemma *iszero-number-of-Bit1*:
~ *iszero* (*number-of* (*Bit1 w*)::*'a*::{*ring-char-0*,*number-ring*})
⟨*proof*⟩

lemmas *iszero-simps* [*simp*] =
iszero-0 not-iszero-1
iszero-number-of-Pls nonzero-number-of-Min
iszero-number-of-Bit0 iszero-number-of-Bit1

30.10.3 The Less-Than Relation

lemma *double-less-0-iff*:
(*a* + *a* < *0*) = (*a* < (*0*::*'a*::*linordered-idom*))
⟨*proof*⟩

lemma *odd-less-0*:

$(1 + z + z < 0) = (z < (0::int))$
⟨*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*]

Absolute value (*abs*)

lemma *abs-number-of*:
$abs(number\text{-}of\ x::'a::\{linordered\text{-}idom, number\text{-}ring\}) =$
$(if\ number\text{-}of\ x < (0::'a)\ then\ -number\text{-}of\ x\ else\ number\text{-}of\ x)$
⟨*proof*⟩

Re-orientation of the equation nnn=x

lemma *number-of-reorient*:
$(number\text{-}of\ w = x) = (x = number\text{-}of\ w)$
⟨*proof*⟩

30.10.4 Simplification of arithmetic operations on integer constants.

lemmas *arith-extra-simps* [*standard*, *simp*] =
number-of-add [*symmetric*]
number-of-minus [*symmetric*]
numeral-m1-eq-minus-1 [*symmetric*]
number-of-mult [*symmetric*]
diff-number-of-eq abs-number-of

For making a minimal simpset, one must include these default simprules. Also include *simp-thms*.

lemmas *arith-simps* =
normalize-bin-simps pred-bin-simps succ-bin-simps
add-bin-simps minus-bin-simps mult-bin-simps
abs-zero abs-one arith-extra-simps

Simplification of relational operations

lemma *less-number-of* [*simp*]:
$(number\text{-}of\ x::'a::\{linordered\text{-}idom, number\text{-}ring\}) < number\text{-}of\ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *le-number-of* [*simp*]:
$(number\text{-}of\ x::'a::\{linordered\text{-}idom, number\text{-}ring\}) \leq number\text{-}of\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *eq-number-of* [*simp*]:

(*number-of* x::$'a$::{*ring-char-0*,*number-ring*}) = *number-of* $y \longleftrightarrow x = y$
⟨*proof*⟩

lemmas *rel-simps* =
less-number-of less-bin-simps
le-number-of le-bin-simps
eq-number-of-eq eq-bin-simps
iszero-simps

30.10.5 Simplification of arithmetic when nested to the right.

lemma *add-number-of-left* [*simp*]:
number-of v + (*number-of* w + z) =
(*number-of*(v + w) + z::$'a$::*number-ring*)
⟨*proof*⟩

lemma *mult-number-of-left* [*simp*]:
number-of v * (*number-of* w * z) =
(*number-of*(v * w) * z::$'a$::*number-ring*)
⟨*proof*⟩

lemma *add-number-of-diff1*:
number-of v + (*number-of* w − c) =
number-of(v + w) − (c::$'a$::*number-ring*)
⟨*proof*⟩

lemma *add-number-of-diff2* [*simp*]:
number-of v + (c − *number-of* w) =
number-of (v + *uminus* w) + (c::$'a$::*number-ring*)
⟨*proof*⟩

30.11 The Set of Integers

context *ring-1*
begin

definition *Ints* :: *'a set* **where**
[*code del*]: *Ints* = *range of-int*

notation (*xsymbols*)
Ints (ℤ)

lemma *Ints-of-int* [*simp*]: *of-int* z ∈ ℤ
⟨*proof*⟩

lemma *Ints-of-nat* [*simp*]: *of-nat* n ∈ ℤ
⟨*proof*⟩

lemma *Ints-0* [*simp*]: 0 ∈ ℤ
⟨*proof*⟩

lemma *Ints-1* [*simp*]: $1 \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-add* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a + b \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-minus* [*simp*]: $a \in \mathbb{Z} \Longrightarrow -a \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a - b \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-mult* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a * b \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-power* [*simp*]: $a \in \mathbb{Z} \Longrightarrow a \hat{} n \in \mathbb{Z}$
⟨*proof*⟩

lemma *Ints-cases* [*cases set*: *Ints*]:
assumes $q \in \mathbb{Z}$
obtains (*of-int*) z **where** $q = \textit{of-int}\ z$
⟨*proof*⟩

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
$q \in \mathbb{Z} \Longrightarrow (\bigwedge z.\ P\ (\textit{of-int}\ z)) \Longrightarrow P\ q$
⟨*proof*⟩

end

The premise involving $\mathbb{Z}$ prevents $a = (1::'a)\ /\ (2::'a)$.

lemma *Ints-double-eq-0-iff*:
assumes *in-Ints*: $a \in \textit{Ints}$
shows $(a + a = 0) = (a = (0::'a::\textit{ring-char-0}))$
⟨*proof*⟩

lemma *Ints-odd-nonzero*:
assumes *in-Ints*: $a \in \textit{Ints}$
shows $1 + a + a \neq (0::'a::\textit{ring-char-0})$
⟨*proof*⟩

lemma *Ints-number-of* [*simp*]:
$(\textit{number-of}\ w :: 'a::\textit{number-ring}) \in \textit{Ints}$
⟨*proof*⟩

lemma *Nats-number-of* [*simp*]:
$\textit{Int.Pls} \leq w \Longrightarrow (\textit{number-of}\ w :: 'a::\textit{number-ring}) \in \textit{Nats}$
⟨*proof*⟩

lemma *Ints-odd-less-0*:
assumes *in-Ints*: $a \in Ints$
shows $(1 + a + a < 0) = (a < (0::'a::linordered\text{-}idom))$
⟨*proof*⟩

30.12 *setsum* and *setprod*

lemma *of-nat-setsum*: *of-nat* $(setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}nat(f\ x))$
⟨*proof*⟩

lemma *of-int-setsum*: *of-int* $(setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}int(f\ x))$
⟨*proof*⟩

lemma *of-nat-setprod*: *of-nat* $(setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}nat(f\ x))$
⟨*proof*⟩

lemma *of-int-setprod*: *of-int* $(setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}int(f\ x))$
⟨*proof*⟩

lemmas *int-setsum* = *of-nat-setsum* [**where** *'a=int*]
lemmas *int-setprod* = *of-nat-setprod* [**where** *'a=int*]

30.13 Inequality Reasoning for the Arithmetic Simproc

lemma *add-numeral-0*: $Numeral0 + a = (a::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *add-numeral-0-right*: $a + Numeral0 = (a::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *mult-numeral-1*: $Numeral1 * a = (a::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *mult-numeral-1-right*: $a * Numeral1 = (a::'a::number\text{-}ring)$
⟨*proof*⟩

lemma *divide-numeral-1*: $a\ /\ Numeral1 = (a::'a::\{number\text{-}ring,field\})$
⟨*proof*⟩

lemma *inverse-numeral-1*:
$inverse\ Numeral1 = (Numeral1::'a::\{number\text{-}ring,field\})$
⟨*proof*⟩

Theorem lists for the cancellation simprocs. The use of binary numerals for 0 and 1 reduces the number of special cases.

lemmas *add-0s* = *add-numeral-0 add-numeral-0-right*
lemmas *mult-1s* = *mult-numeral-1 mult-numeral-1-right*
mult-minus1 mult-minus1-right

30.14 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*: $1 + 1 = (2::'a::number\text{-}ring)$
⟨*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 numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op <*, *OF less-number-of numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op <*, *OF less-number-of refl numeral-0-eq-0*, *standard*]
binop-eq [*of op <*, *OF less-number-of 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 numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op ≤*, *OF le-number-of numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op ≤*, *OF le-number-of refl numeral-0-eq-0*, *standard*]
binop-eq [*of op ≤*, *OF le-number-of refl numeral-1-eq-1*, *standard*]

lemmas *arith-special*[*simp*] =

add-special diff-special eq-special less-special le-special

Legacy theorems

lemmas *zle-int* = *of-nat-le-iff* [**where** *'a=int*]
lemmas *int-int-eq* = *of-nat-eq-iff* [**where** *'a=int*]

30.15 Setting up simplification procedures

lemmas *int-arith-rules* =
neg-le-iff-le numeral-0-eq-0 numeral-1-eq-1
minus-zero diff-minus left-minus right-minus
mult-zero-left mult-zero-right mult-Bit1 mult-1-left mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult-assoc
of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult
of-int-0 of-int-1 of-int-add of-int-mult

⟨*ML*⟩

30.16 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*::{*linordered-idom*,*number-ring*})
⟨*proof*⟩

lemma *abs-power-minus-one* [*simp*]:
abs(*−1* ^ *n*) = (*1*::*'a*::{*linordered-idom*,*number-ring*})
⟨*proof*⟩

lemma *of-int-number-of-eq* [*simp*]:
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*⟩

30.17 More Inequality Reasoning

lemma *zless-add1-eq*: (*w* < *z* + (*1*::*int*)) = (*w*<*z* | *w*=*z*)
⟨*proof*⟩

lemma *add1-zle-eq*: (*w* + (*1*::*int*) ≤ *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*⟩

30.18 The functions *nat* and *int*

Simplify the terms $int\ (0::'a)$, $int\ (Suc\ 0)$ and $w + - z$

declare *Zero-int-def* [*symmetric*, *simp*]
declare *One-int-def* [*symmetric*, *simp*]

lemmas *diff-int-def-symmetric* = *diff-int-def* [*symmetric*, *simp*]

lemma *nat-0*: $nat\ 0 = 0$
⟨*proof*⟩

lemma *nat-1*: $nat\ 1 = Suc\ 0$
⟨*proof*⟩

lemma *nat-2*: $nat\ 2 = Suc\ (Suc\ 0)$
⟨*proof*⟩

lemma *one-less-nat-eq* [*simp*]: $(Suc\ 0 < nat\ z) = (1 < z)$
⟨*proof*⟩

This simplifies expressions of the form $int\ n = z$ where z is an integer literal.

lemmas *int-eq-iff-number-of* [*simp*] = *int-eq-iff* [*of - number-of v*, *standard*]

lemma *split-nat* [*arith-split*]:
$P(nat(i::int)) = ((\forall n.\ i = of\text{-}nat\ n \longrightarrow P\ n)\ \&\ (i < 0 \longrightarrow P\ 0))$
(**is** $?P = (?L\ \&\ ?R)$)
⟨*proof*⟩

context *ring-1*
begin

lemma *of-int-of-nat* [*nitpick-simp*]:
$of\text{-}int\ k = (if\ k < 0\ then\ -\ of\text{-}nat\ (nat\ (-\ k))\ else\ of\text{-}nat\ (nat\ k))$
⟨*proof*⟩

end

lemma *nat-mult-distrib*:

fixes $z\ z'$ :: *int*
assumes $0 \leq z$
shows $nat\ (z * z') = nat\ z * nat\ z'$
⟨*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*⟩

30.19 Induction principles for int

Well-founded segments of the integers

definition
int-ge-less-than :: $int => (int * int)\ set$
where
$int\text{-}ge\text{-}less\text{-}than\ d = \{(z',z).\ d \leq z' \ \&\ z' < z\}$

theorem *wf-int-ge-less-than*: $wf\ (int\text{-}ge\text{-}less\text{-}than\ d)$
⟨*proof*⟩

This variant looks odd, but is typical of the relations suggested by RankFinder.

definition
int-ge-less-than2 :: $int => (int * int)\ set$
where
$int\text{-}ge\text{-}less\text{-}than2\ d = \{(z',z).\ d \leq z\ \&\ z' < z\}$

theorem *wf-int-ge-less-than2*: $wf\ (int\text{-}ge\text{-}less\text{-}than2\ d)$
⟨*proof*⟩

abbreviation
int :: $nat \Rightarrow int$
where
$int \equiv of\text{-}nat$

theorem *int-ge-induct* [*case-names base step*, *induct set*: *int*]:
fixes i :: *int*
assumes *ge*: $k \leq i$ **and**
base: $P\ k$ **and**
step: $\bigwedge i.\ k \leq i \Longrightarrow P\ i \Longrightarrow P\ (i + 1)$
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*⟩

theorem *int-induct* [*case-names base step1 step2*]:
fixes $k :: int$
assumes *base*: $P\ k$
and *step1*: $\bigwedge i.\ k \leq i \Longrightarrow P\ i \Longrightarrow P\ (i + 1)$
and *step2*: $\bigwedge i.\ k \geq i \Longrightarrow P\ i \Longrightarrow P\ (i - 1)$
shows $P\ i$
⟨*proof*⟩

30.20 Intermediate value theorems

lemma *int-val-lemma*:
$(\forall i{<}n::nat.\ abs(f(i+1) - f\ i) \leq 1)\ \text{--}\text{>}$
$f\ 0 \leq k\ \text{--}\text{>}\ k \leq f\ n\ \text{--}\text{>}\ (\exists i \leq n.\ f\ i = (k::int))$
⟨*proof*⟩

lemmas *nat0-intermed-int-val* = *int-val-lemma* [*rule-format* (*no-asm*)]

lemma *nat-intermed-int-val*:
$[\!|\ \forall i.\ m \leq i\ \&\ i < n\ \text{--}\text{>}\ abs(f(i + 1::nat) - f\ i) \leq 1;\ m < n;$
$f\ m \leq k;\ k \leq f\ n\ |\!]\ \text{==}\text{>}\ ?\ i.\ m \leq i\ \&\ i \leq n\ \&\ f\ i = (k::int)$
⟨*proof*⟩

30.21 Products and 1, by T. M. Rasmussen

lemma *zabs-less-one-iff* [*simp*]: $(|z| < 1) = (z = (0::int))$
⟨*proof*⟩

lemma *abs-zmult-eq-1*:
assumes *mn*: $|m * n| = 1$
shows $|m| = (1::int)$

⟨*proof*⟩

lemma *pos-zmult-eq-1-iff-lemma*: $(m * n = 1) ==> m = (1::int) \mid m = -1$
⟨*proof*⟩

lemma *pos-zmult-eq-1-iff*:
assumes $0 < (m::int)$ **shows** $(m * n = 1) = (m = 1 \ \& \ n = 1)$
⟨*proof*⟩

lemma *zmult-eq-1-iff*: $(m*n = (1::int)) = ((m = 1 \ \& \ n = 1) \mid (m = -1 \ \& \ n = -1))$
⟨*proof*⟩

lemma *infinite-UNIV-int*: $\neg$ *finite* $(UNIV::int\ set)$
⟨*proof*⟩

30.22 Further theorems on numerals

30.22.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

lemmas *left-distrib-number-of* [*simp*] =
left-distrib [*of - - number-of v*, *standard*]

lemmas *right-distrib-number-of* [*simp*] =
right-distrib [*of number-of v*, *standard*]

lemmas *left-diff-distrib-number-of* [*simp*] =
left-diff-distrib [*of - - number-of v*, *standard*]

lemmas *right-diff-distrib-number-of* [*simp*] =
right-diff-distrib [*of number-of v*, *standard*]

These are actually for fields, like real: but where else to put them?

lemmas *zero-less-divide-iff-number-of* [*simp*, *no-atp*] =
zero-less-divide-iff [*of number-of w*, *standard*]

lemmas *divide-less-0-iff-number-of* [*simp*, *no-atp*] =
divide-less-0-iff [*of number-of w*, *standard*]

lemmas *zero-le-divide-iff-number-of* [*simp*, *no-atp*] =
zero-le-divide-iff [*of number-of w*, *standard*]

lemmas *divide-le-0-iff-number-of* [*simp*, *no-atp*] =
divide-le-0-iff [*of number-of w*, *standard*]

Replaces *inverse #nn* by *1/#nn*. It looks strange, but then other simprocs simplify the quotient.

lemmas *inverse-eq-divide-number-of* [*simp*] =

inverse-eq-divide [*of number-of w*, *standard*]

These laws simplify inequalities, moving unary minus from a term into the literal.

lemmas *less-minus-iff-number-of* [*simp*, *no-atp*] =
less-minus-iff [*of number-of v*, *standard*]

lemmas *le-minus-iff-number-of* [*simp*, *no-atp*] =
le-minus-iff [*of number-of v*, *standard*]

lemmas *equation-minus-iff-number-of* [*simp*, *no-atp*] =
equation-minus-iff [*of number-of v*, *standard*]

lemmas *minus-less-iff-number-of* [*simp*, *no-atp*] =
minus-less-iff [*of - number-of v*, *standard*]

lemmas *minus-le-iff-number-of* [*simp*, *no-atp*] =
minus-le-iff [*of - number-of v*, *standard*]

lemmas *minus-equation-iff-number-of* [*simp*, *no-atp*] =
minus-equation-iff [*of - number-of v*, *standard*]

To Simplify Inequalities Where One Side is the Constant 1

lemma *less-minus-iff-1* [*simp*,*no-atp*]:
fixes *b*::'*b*::{*linordered-idom*,*number-ring*}
shows (*1* < − *b*) = (*b* < −*1*)
⟨*proof*⟩

lemma *le-minus-iff-1* [*simp*,*no-atp*]:
fixes *b*::'*b*::{*linordered-idom*,*number-ring*}
shows (*1* ≤ − *b*) = (*b* ≤ −*1*)
⟨*proof*⟩

lemma *equation-minus-iff-1* [*simp*,*no-atp*]:
fixes *b*::'*b*::*number-ring*
shows (*1* = − *b*) = (*b* = −*1*)
⟨*proof*⟩

lemma *minus-less-iff-1* [*simp*,*no-atp*]:
fixes *a*::'*b*::{*linordered-idom*,*number-ring*}
shows (− *a* < *1*) = (−*1* < *a*)
⟨*proof*⟩

lemma *minus-le-iff-1* [*simp*,*no-atp*]:
fixes *a*::'*b*::{*linordered-idom*,*number-ring*}
shows (− *a* ≤ *1*) = (−*1* ≤ *a*)
⟨*proof*⟩

lemma *minus-equation-iff-1* [*simp*,*no-atp*]:

fixes $a::'b::number\text{-}ring$
shows $(-\ a = 1) = (a = -1)$
⟨*proof*⟩

Cancellation of constant factors in comparisons (< and ≤)

lemmas *mult-less-cancel-left-number-of* [*simp*, *no-atp*] =
mult-less-cancel-left [*of number-of v*, *standard*]

lemmas *mult-less-cancel-right-number-of* [*simp*, *no-atp*] =
mult-less-cancel-right [*of - number-of v*, *standard*]

lemmas *mult-le-cancel-left-number-of* [*simp*, *no-atp*] =
mult-le-cancel-left [*of number-of v*, *standard*]

lemmas *mult-le-cancel-right-number-of* [*simp*, *no-atp*] =
mult-le-cancel-right [*of - number-of v*, *standard*]

Multiplying out constant divisors in comparisons (<, ≤ and =)

lemmas *le-divide-eq-number-of1* [*simp*] = *le-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-le-eq-number-of1* [*simp*] = *divide-le-eq* [*of - number-of w*, *standard*]
lemmas *less-divide-eq-number-of1* [*simp*] = *less-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-less-eq-number-of1* [*simp*] = *divide-less-eq* [*of - number-of w*, *standard*]
lemmas *eq-divide-eq-number-of1* [*simp*] = *eq-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-eq-eq-number-of1* [*simp*] = *divide-eq-eq* [*of - number-of w*, *standard*]

30.22.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas *le-divide-eq-number-of* = *le-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-le-eq-number-of* = *divide-le-eq* [*of - - number-of w*, *standard*]
lemmas *less-divide-eq-number-of* = *less-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-less-eq-number-of* = *divide-less-eq* [*of - - number-of w*, *standard*]
lemmas *eq-divide-eq-number-of* = *eq-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-eq-eq-number-of* = *divide-eq-eq* [*of - - number-of w*, *standard*]

Not good as automatic simprules because they cause case splits.

lemmas *divide-const-simps* =
le-divide-eq-number-of divide-le-eq-number-of less-divide-eq-number-of
divide-less-eq-number-of eq-divide-eq-number-of divide-eq-eq-number-of
le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1

Division By -1

lemma *divide-minus1* [*simp*]:
$x/-1 = -(x::'a::\{field\text{-}inverse\text{-}zero,\ number\text{-}ring\})$

⟨*proof*⟩

lemma *minus1-divide* [*simp*]:
  $-1 \;/\; (x::'a::\{\textit{field-inverse-zero}, \textit{number-ring}\}) = -\ (1/x)$
⟨*proof*⟩

lemma *half-gt-zero-iff*:
  $(0 < r/2) = (0 < (r::'a::\{\textit{linordered-field-inverse-zero},\textit{number-ring}\}))$
⟨*proof*⟩

lemmas *half-gt-zero* [*simp*] = *half-gt-zero-iff* [*THEN iffD2*, *standard*]

lemma *divide-Numeral1*:
  $(x::'a::\{\textit{field}, \textit{number-ring}\}) \;/\; \textit{Numeral1} = x$
  ⟨*proof*⟩

lemma *divide-Numeral0*:
  $(x::'a::\{\textit{field-inverse-zero}, \textit{number-ring}\}) \;/\; \textit{Numeral0} = 0$
  ⟨*proof*⟩

30.23 The divides relation

lemma *zdvd-antisym-nonneg*:
  $0 <= m \Longrightarrow 0 <= n \Longrightarrow m \;dvd\; n \Longrightarrow n \;dvd\; m \Longrightarrow m = (n::int)$
  ⟨*proof*⟩

lemma *zdvd-antisym-abs*: **assumes** $(a::int)\ dvd\ b$ **and** $b\ dvd\ a$
  shows $|a| = |b|$
⟨*proof*⟩

lemma *zdvd-zdiffD*: $k\ dvd\ m - n \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ (m::int)$
  ⟨*proof*⟩

lemma *zdvd-reduce*: $(k\ dvd\ n + k * m) = (k\ dvd\ (n::int))$
⟨*proof*⟩

lemma *dvd-imp-le-int*:
  fixes $d\ i :: int$
  assumes $i \neq 0$ **and** $d\ dvd\ i$
  shows $|d| \leq |i|$
⟨*proof*⟩

lemma *zdvd-not-zless*:
  fixes $m\ n :: int$
  assumes $0 < m$ **and** $m < n$
  shows $\neg\ n\ dvd\ m$
⟨*proof*⟩

lemma *zdvd-mult-cancel*: **assumes** $d{:}k * m\ dvd\ k * n$ **and** $kz{:}k \neq (0::int)$

shows *m dvd n*
⟨*proof*⟩

theorem *zdvd-int*: (*x dvd y*) = (*int x dvd int y*)
⟨*proof*⟩

lemma *zdvd1-eq*[*simp*]: (*x*::*int*) *dvd 1* = (|*x*| = *1*)
⟨*proof*⟩

lemma *zdvd-mult-cancel1*:
assumes *mp*:*m* ≠(*0*::*int*) **shows** (*m* ∗ *n dvd m*) = (|*n*| = *1*)
⟨*proof*⟩

lemma *int-dvd-iff*: (*int m dvd z*) = (*m dvd nat* (*abs z*))
⟨*proof*⟩

lemma *dvd-int-iff*: (*z dvd int m*) = (*nat* (*abs z*) *dvd m*)
⟨*proof*⟩

lemma *nat-dvd-iff*: (*nat z dvd m*) = (*if* 0 ≤ *z then* (*z dvd int m*) *else m* = *0*)
⟨*proof*⟩

lemma *eq-nat-nat-iff*:
0 ≤ *z* ⟹ *0* ≤ *z'* ⟹ *nat z* = *nat z'* ⟷ *z* = *z'*
⟨*proof*⟩

lemma *nat-power-eq*:
0 ≤ *z* ⟹ *nat* (*z* ^ *n*) = *nat z* ^ *n*
⟨*proof*⟩

lemma *zdvd-imp-le*: [| *z dvd n*; *0* < *n* |] ==> *z* ≤ (*n*::*int*)
⟨*proof*⟩

lemma *zdvd-period*:
fixes *a d* :: *int*
assumes *a dvd d*
shows *a dvd* (*x* + *t*) ⟷ *a dvd* ((*x* + *c* ∗ *d*) + *t*)
⟨*proof*⟩

30.24 Configuration of the code generator

code-datatype *Pls Min Bit0 Bit1 number-of* :: *int* ⇒ *int*

lemmas *pred-succ-numeral-code* [*code*] =
pred-bin-simps succ-bin-simps

lemmas *plus-numeral-code* [*code*] =
add-bin-simps
arith-extra-simps(*1*) [**where** *'a* = *int*]

lemmas *minus-numeral-code* [*code*] =
minus-bin-simps
arith-extra-simps(*2*) [**where** *'a* = *int*]
arith-extra-simps(*5*) [**where** *'a* = *int*]

lemmas *times-numeral-code* [*code*] =
mult-bin-simps
arith-extra-simps(*4*) [**where** *'a* = *int*]

instantiation *int* :: *eq*
begin

definition [*code del*]: *eq-class.eq k l* ⟷ *k* − *l* = (*0*::*int*)

instance ⟨*proof*⟩

end

lemma *eq-number-of-int-code* [*code*]:
eq-class.eq (*number-of k* :: *int*) (*number-of l*) ⟷ *eq-class.eq k l*
⟨*proof*⟩

lemma *eq-int-code* [*code*]:
eq-class.eq Int.Pls Int.Pls ⟷ *True*
eq-class.eq Int.Pls Int.Min ⟷ *False*
eq-class.eq Int.Pls (*Int.Bit0 k2*) ⟷ *eq-class.eq Int.Pls k2*
eq-class.eq Int.Pls (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq Int.Min Int.Pls ⟷ *False*
eq-class.eq Int.Min Int.Min ⟷ *True*
eq-class.eq Int.Min (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq Int.Min (*Int.Bit1 k2*) ⟷ *eq-class.eq Int.Min k2*
eq-class.eq (*Int.Bit0 k1*) *Int.Pls* ⟷ *eq-class.eq k1 Int.Pls*
eq-class.eq (*Int.Bit1 k1*) *Int.Pls* ⟷ *False*
eq-class.eq (*Int.Bit0 k1*) *Int.Min* ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) *Int.Min* ⟷ *eq-class.eq k1 Int.Min*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit0 k2*) ⟷ *eq-class.eq k1 k2*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit1 k2*) ⟷ *eq-class.eq k1 k2*
⟨*proof*⟩

lemma *eq-int-refl* [*code nbe*]:
eq-class.eq (*k*::*int*) *k* ⟷ *True*
⟨*proof*⟩

lemma *less-eq-number-of-int-code* [*code*]:
(*number-of k* :: *int*) ≤ *number-of l* ⟷ *k* ≤ *l*
⟨*proof*⟩

lemma *less-eq-int-code* [*code*]:
Int.Pls $\leq$ *Int.Pls* $\longleftrightarrow$ *True*
Int.Pls $\leq$ *Int.Min* $\longleftrightarrow$ *False*
Int.Pls $\leq$ *Int.Bit0 k* $\longleftrightarrow$ *Int.Pls* $\leq$ *k*
Int.Pls $\leq$ *Int.Bit1 k* $\longleftrightarrow$ *Int.Pls* $\leq$ *k*
Int.Min $\leq$ *Int.Pls* $\longleftrightarrow$ *True*
Int.Min $\leq$ *Int.Min* $\longleftrightarrow$ *True*
Int.Min $\leq$ *Int.Bit0 k* $\longleftrightarrow$ *Int.Min* $<$ *k*
Int.Min $\leq$ *Int.Bit1 k* $\longleftrightarrow$ *Int.Min* $\leq$ *k*
Int.Bit0 k $\leq$ *Int.Pls* $\longleftrightarrow$ *k* $\leq$ *Int.Pls*
Int.Bit1 k $\leq$ *Int.Pls* $\longleftrightarrow$ *k* $<$ *Int.Pls*
Int.Bit0 k $\leq$ *Int.Min* $\longleftrightarrow$ *k* $\leq$ *Int.Min*
Int.Bit1 k $\leq$ *Int.Min* $\longleftrightarrow$ *k* $\leq$ *Int.Min*
Int.Bit0 k1 $\leq$ *Int.Bit0 k2* $\longleftrightarrow$ *k1* $\leq$ *k2*
Int.Bit0 k1 $\leq$ *Int.Bit1 k2* $\longleftrightarrow$ *k1* $\leq$ *k2*
Int.Bit1 k1 $\leq$ *Int.Bit0 k2* $\longleftrightarrow$ *k1* $<$ *k2*
Int.Bit1 k1 $\leq$ *Int.Bit1 k2* $\longleftrightarrow$ *k1* $\leq$ *k2*
⟨*proof*⟩

lemma *less-number-of-int-code* [*code*]:
(*number-of k* :: *int*) $<$ *number-of l* $\longleftrightarrow$ *k* $<$ *l*
⟨*proof*⟩

lemma *less-int-code* [*code*]:
Int.Pls $<$ *Int.Pls* $\longleftrightarrow$ *False*
Int.Pls $<$ *Int.Min* $\longleftrightarrow$ *False*
Int.Pls $<$ *Int.Bit0 k* $\longleftrightarrow$ *Int.Pls* $<$ *k*
Int.Pls $<$ *Int.Bit1 k* $\longleftrightarrow$ *Int.Pls* $\leq$ *k*
Int.Min $<$ *Int.Pls* $\longleftrightarrow$ *True*
Int.Min $<$ *Int.Min* $\longleftrightarrow$ *False*
Int.Min $<$ *Int.Bit0 k* $\longleftrightarrow$ *Int.Min* $<$ *k*
Int.Min $<$ *Int.Bit1 k* $\longleftrightarrow$ *Int.Min* $<$ *k*
Int.Bit0 k $<$ *Int.Pls* $\longleftrightarrow$ *k* $<$ *Int.Pls*
Int.Bit1 k $<$ *Int.Pls* $\longleftrightarrow$ *k* $<$ *Int.Pls*
Int.Bit0 k $<$ *Int.Min* $\longleftrightarrow$ *k* $\leq$ *Int.Min*
Int.Bit1 k $<$ *Int.Min* $\longleftrightarrow$ *k* $<$ *Int.Min*
Int.Bit0 k1 $<$ *Int.Bit0 k2* $\longleftrightarrow$ *k1* $<$ *k2*
Int.Bit0 k1 $<$ *Int.Bit1 k2* $\longleftrightarrow$ *k1* $\leq$ *k2*
Int.Bit1 k1 $<$ *Int.Bit0 k2* $\longleftrightarrow$ *k1* $<$ *k2*
Int.Bit1 k1 $<$ *Int.Bit1 k2* $\longleftrightarrow$ *k1* $<$ *k2*
⟨*proof*⟩

definition
nat-aux :: *int* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
nat-aux i n = *nat i* + *n*

lemma [*code*]:
nat-aux i n = (*if i* $\leq$ *0 then n else nat-aux* (*i* $-$ *1*) (*Suc n*)) — tail recursive

⟨*proof*⟩

lemma [*code*]: *nat i = nat-aux i 0*
⟨*proof*⟩

hide-const (**open**) *nat-aux*

lemma *zero-is-num-zero* [*code*, *code-unfold-post*]:
(*0*::*int*) = *Numeral0*
⟨*proof*⟩

lemma *one-is-num-one* [*code*, *code-unfold-post*]:
(*1*::*int*) = *Numeral1*
⟨*proof*⟩

code-modulename *SML*
Int Arith

code-modulename *OCaml*
Int Arith

code-modulename *Haskell*
Int Arith

types-code
int (*int*)
attach (*term-of*) ⟪
val term-of-int = HOLogic.mk-number HOLogic.intT;
⟫
attach (*test*) ⟪
fun gen-int i =
let val j = one-of [*~1*, *1*] ** random-range 0 i*
in (*j*, *fn* () => *term-of-int j*) *end*;
⟫

⟨*ML*⟩

consts-code
number-of :: *int* ⇒ *int* ((-))
0 :: *int* (*0*)
1 :: *int* (*1*)
uminus :: *int* => *int* (~)
op + :: *int* => *int* => *int* ((- +/ -))
*op ** :: *int* => *int* => *int* ((- */ -))
op ≤ :: *int* => *int* => *bool* ((- <=/ -))
op < :: *int* => *int* => *bool* ((- </ -))

quickcheck-params [*default-type = int*]

hide-const (**open**) *Pls Min Bit0 Bit1 succ pred*

30.25 Legacy theorems

lemmas *zminus-zminus = minus-minus* [*of z::int*, *standard*]
lemmas *zminus-0 = minus-zero* [**where** *'a=int*]
lemmas *zminus-zadd-distrib = minus-add-distrib* [*of z::int w*, *standard*]
lemmas *zadd-commute = add-commute* [*of z::int w*, *standard*]
lemmas *zadd-assoc = add-assoc* [*of z1::int z2 z3*, *standard*]
lemmas *zadd-left-commute = add-left-commute* [*of x::int y z*, *standard*]
lemmas *zadd-ac = zadd-assoc zadd-commute zadd-left-commute*
lemmas *zmult-ac = mult-ac*
lemmas *zadd-0 = add-0-left* [*of z::int*, *standard*]
lemmas *zadd-0-right = add-0-right* [*of z::int*, *standard*]
lemmas *zadd-zminus-inverse2 = left-minus* [*of z::int*, *standard*]
lemmas *zmult-zminus = mult-minus-left* [*of z::int w*, *standard*]
lemmas *zmult-commute = mult-commute* [*of z::int w*, *standard*]
lemmas *zmult-assoc = mult-assoc* [*of z1::int z2 z3*, *standard*]
lemmas *zadd-zmult-distrib = left-distrib* [*of z1::int z2 w*, *standard*]
lemmas *zadd-zmult-distrib2 = right-distrib* [*of w::int z1 z2*, *standard*]
lemmas *zdiff-zmult-distrib = left-diff-distrib* [*of z1::int z2 w*, *standard*]
lemmas *zdiff-zmult-distrib2 = right-diff-distrib* [*of w::int z1 z2*, *standard*]

lemmas *zmult-1 = mult-1-left* [*of z::int*, *standard*]
lemmas *zmult-1-right = mult-1-right* [*of z::int*, *standard*]

lemmas *zle-refl = order-refl* [*of w::int*, *standard*]
lemmas *zle-trans = order-trans* [**where** *'a=int* **and** *x=i* **and** *y=j* **and** *z=k*, *standard*]
lemmas *zle-antisym = order-antisym* [*of z::int w*, *standard*]
lemmas *zle-linear = linorder-linear* [*of z::int w*, *standard*]
lemmas *zless-linear = linorder-less-linear* [**where** *'a = int*]

lemmas *zadd-left-mono = add-left-mono* [*of i::int j k*, *standard*]
lemmas *zadd-strict-right-mono = add-strict-right-mono* [*of i::int j k*, *standard*]
lemmas *zadd-zless-mono = add-less-le-mono* [*of w'::int w z'* *z*, *standard*]

lemmas *int-0-less-1 = zero-less-one* [**where** *'a=int*]
lemmas *int-0-neq-1 = zero-neq-one* [**where** *'a=int*]

lemmas *inj-int = inj-of-nat* [**where** *'a=int*]
lemmas *zadd-int = of-nat-add* [**where** *'a=int*, *symmetric*]
lemmas *int-mult = of-nat-mult* [**where** *'a=int*]
lemmas *zmult-int = of-nat-mult* [**where** *'a=int*, *symmetric*]
lemmas *int-eq-0-conv = of-nat-eq-0-iff* [**where** *'a=int* **and** *m=n*, *standard*]
lemmas *zless-int = of-nat-less-iff* [**where** *'a=int*]
lemmas *int-less-0-conv = of-nat-less-0-iff* [**where** *'a=int* **and** *m=k*, *standard*]
lemmas *zero-less-int-conv = of-nat-0-less-iff* [**where** *'a=int*]
lemmas *zero-zle-int = of-nat-0-le-iff* [**where** *'a=int*]

lemmas *int-le-0-conv* = *of-nat-le-0-iff* [**where** *'a=int* **and** *m=n*, *standard*]
lemmas *int-0* = *of-nat-0* [**where** *'a=int*]
lemmas *int-1* = *of-nat-1* [**where** *'a=int*]
lemmas *int-Suc* = *of-nat-Suc* [**where** *'a=int*]
lemmas *abs-int-eq* = *abs-of-nat* [**where** *'a=int* **and** *n=m*, *standard*]
lemmas *of-int-int-eq* = *of-int-of-nat-eq* [**where** *'a=int*]
lemmas *zdiff-int* = *of-nat-diff* [**where** *'a=int*, *symmetric*]
lemmas *zless-le* = *less-int-def*
lemmas *int-eq-of-nat* = *TrueI*

lemma *zpower-zadd-distrib*:
$x \hat{} (y + z) = ((x \hat{} y) * (x \hat{} z)::int)$
⟨*proof*⟩

lemma *zero-less-zpower-abs-iff*:
$(0 < abs\ x \hat{} n) \longleftrightarrow (x \neq (0::int) \mid n = 0)$
⟨*proof*⟩

lemma *zero-le-zpower-abs*: $(0::int) \leq abs\ x \hat{} n$
⟨*proof*⟩

lemma *zpower-zpower*:
$(x \hat{} y) \hat{} z = (x \hat{} (y * z)::int)$
⟨*proof*⟩

lemma *int-power*:
$int\ (m \hat{} n) = int\ m \hat{} n$
⟨*proof*⟩

lemmas *zpower-int* = *int-power* [*symmetric*]

end

31 Nat-Numeral: Binary numerals for the natural numbers

theory *Nat-Numeral*
imports *Int*
begin

31.1 Numerals for natural numbers

Arithmetic for naturals is reduced to that for the non-negative integers.

instantiation *nat* :: *number*
begin

definition

nat-number-of-def [*code-unfold*, *code del*]: *number-of v* = *nat* (*number-of v*)

instance ⟨*proof*⟩

end

lemma [*code-post*]:
nat (*number-of v*) = *number-of v*
⟨*proof*⟩

31.2 Special case: squares and cubes

lemma *numeral-2-eq-2*: *2* = *Suc* (*Suc 0*)
⟨*proof*⟩

lemma *numeral-3-eq-3*: *3* = *Suc* (*Suc* (*Suc 0*))
⟨*proof*⟩

context *power*
begin

abbreviation (*xsymbols*)
power2 :: *'a* ⇒ *'a* ((-^{2}) [*1000*] *999*) **where**
$x^2 \equiv x$ ^ *2*

notation (*latex* **output**)
power2 ((-^{2}) [*1000*] *999*)

notation (*HTML* **output**)
power2 ((-^{2}) [*1000*] *999*)

end

context *monoid-mult*
begin

lemma *power2-eq-square*: $a^2 = a * a$
⟨*proof*⟩

lemma *power3-eq-cube*: *a* ^ *3* = *a* ∗ *a* ∗ *a*
⟨*proof*⟩

lemma *power-even-eq*:
a ^ (*2*∗*n*) = (*a* ^ *n*) ^ *2*
⟨*proof*⟩

lemma *power-odd-eq*:
a ^ *Suc* (*2*∗*n*) = *a* ∗ (*a* ^ *n*) ^ *2*
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *zero-power2* [*simp*]: $0^2 = 0$
⟨*proof*⟩

lemma *one-power2* [*simp*]: $1^2 = 1$
⟨*proof*⟩

end

context *ring-1*
begin

lemma *power2-minus* [*simp*]:
$(-a)^2 = a^2$
⟨*proof*⟩

We cannot prove general results about the numeral $-1::'b$, so we have to use $-(1::'a)$ instead.

lemma *power-minus1-even* [*simp*]:
$(-1) \hat{} (2*n) = 1$
⟨*proof*⟩

lemma *power-minus1-odd*:
$(-1) \hat{} Suc\ (2*n) = -1$
⟨*proof*⟩

lemma *power-minus-even* [*simp*]:
$(-a) \hat{} (2*n) = a \hat{} (2*n)$
⟨*proof*⟩

end

context *ring-1-no-zero-divisors*
begin

lemma *zero-eq-power2* [*simp*]:
$a^2 = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *power2-eq-1-iff*:
$a^2 = 1 \longleftrightarrow a = 1 \vee a = -1$
⟨*proof*⟩

end

context *linordered-ring*
begin

lemma *sum-squares-ge-zero*:
$0 \leq x * x + y * y$
⟨*proof*⟩

lemma *not-sum-squares-lt-zero*:
$\neg\ x * x + y * y < 0$
⟨*proof*⟩

end

context *linordered-ring-strict*
begin

lemma *sum-squares-eq-zero-iff*:
$x * x + y * y = 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemma *sum-squares-le-zero-iff*:
$x * x + y * y \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemma *sum-squares-gt-zero-iff*:
$0 < x * x + y * y \longleftrightarrow x \neq 0 \vee y \neq 0$
⟨*proof*⟩

end

context *linordered-semidom*
begin

lemma *power2-le-imp-le*:
$x^2 \leq y^2 \Longrightarrow 0 \leq y \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *power2-less-imp-less*:
$x^2 < y^2 \Longrightarrow 0 \leq y \Longrightarrow x < y$
⟨*proof*⟩

lemma *power2-eq-imp-eq*:
$x^2 = y^2 \Longrightarrow 0 \leq x \Longrightarrow 0 \leq y \Longrightarrow x = y$
⟨*proof*⟩

end

context *linordered-idom*

begin

lemma *zero-le-power2* [*simp*]:
$0 \leq a^2$
⟨*proof*⟩

lemma *zero-less-power2* [*simp*]:
$0 < a^2 \longleftrightarrow a \neq 0$
⟨*proof*⟩

lemma *power2-less-0* [*simp*]:
$\neg\ a^2 < 0$
⟨*proof*⟩

lemma *abs-power2* [*simp*]:
$abs\ (a^2) = a^2$
⟨*proof*⟩

lemma *power2-abs* [*simp*]:
$(abs\ a)^2 = a^2$
⟨*proof*⟩

lemma *odd-power-less-zero*:
$a < 0 \Longrightarrow a\ \hat{}\ Suc\ (2{*}n) < 0$
⟨*proof*⟩

lemma *odd-0-le-power-imp-0-le*:
$0 \leq a\ \hat{}\ Suc\ (2{*}n) \Longrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-le-even-power'*[*simp*]:
$0 \leq a\ \hat{}\ (2{*}n)$
⟨*proof*⟩

lemma *sum-power2-ge-zero*:
$0 \leq x^2 + y^2$
⟨*proof*⟩

lemma *not-sum-power2-lt-zero*:
$\neg\ x^2 + y^2 < 0$
⟨*proof*⟩

lemma *sum-power2-eq-zero-iff*:
$x^2 + y^2 = 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemma *sum-power2-le-zero-iff*:
$x^2 + y^2 \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemma *sum-power2-gt-zero-iff*:
$0 < x^2 + y^2 \longleftrightarrow x \neq 0 \vee y \neq 0$
⟨*proof*⟩

end

lemma *power2-sum*:
fixes $x\ y :: 'a$::*number-ring*
shows $(x + y)^2 = x^2 + y^2 + 2 * x * y$
⟨*proof*⟩

lemma *power2-diff*:
fixes $x\ y :: 'a$::*number-ring*
shows $(x - y)^2 = x^2 + y^2 - 2 * x * y$
⟨*proof*⟩

31.3 Predicate for negative binary numbers

definition *neg* :: *int* ⇒ *bool* **where**
neg Z ⟷ *Z* < *0*

lemma *not-neg-int* [*simp*]: ~ *neg* (*of-nat n*)
⟨*proof*⟩

lemma *neg-zminus-int* [*simp*]: *neg* (− (*of-nat* (*Suc n*)))
⟨*proof*⟩

lemmas *neg-eq-less-0* = *neg-def*

lemma *not-neg-eq-ge-0*: (~*neg x*) = (*0* ≤ *x*)
⟨*proof*⟩

To simplify inequalities when Numeral1 can get simplified to 1

lemma *not-neg-0*: ~ *neg 0*
⟨*proof*⟩

lemma *not-neg-1*: ~ *neg 1*
⟨*proof*⟩

lemma *neg-nat*: *neg z* ==> *nat z* = *0*
⟨*proof*⟩

lemma *not-neg-nat*: ~ *neg z* ==> *of-nat* (*nat z*) = *z*
⟨*proof*⟩

If *Numeral0* is rewritten to 0 then this rule can't be applied: *Numeral0* IS *number-of Pls*

lemma *not-neg-number-of-Pls*: ~ *neg* (*number-of Int.Pls*)

⟨*proof*⟩

lemma *neg-number-of-Min*: *neg* (*number-of Int.Min*)
⟨*proof*⟩

lemma *neg-number-of-Bit0*:
neg (*number-of* (*Int.Bit0 w*)) = *neg* (*number-of w*)
⟨*proof*⟩

lemma *neg-number-of-Bit1*:
neg (*number-of* (*Int.Bit1 w*)) = *neg* (*number-of w*)
⟨*proof*⟩

lemmas *neg-simps* [*simp*] =
not-neg-0 not-neg-1
not-neg-number-of-Pls neg-number-of-Min
neg-number-of-Bit0 neg-number-of-Bit1

31.4 Function *nat*: Coercion from Type *int* to *nat*

declare *nat-1* [*simp*]

lemma *nat-number-of* [*simp*]: *nat* (*number-of w*) = *number-of w*
⟨*proof*⟩

lemma *nat-numeral-0-eq-0* [*simp*, *code-post*]: *Numeral0* = (*0*::*nat*)
⟨*proof*⟩

lemma *nat-numeral-1-eq-1* [*simp*]: *Numeral1* = (*1*::*nat*)
⟨*proof*⟩

lemma *Numeral1-eq1-nat*:
(*1*::*nat*) = *Numeral1*
⟨*proof*⟩

lemma *numeral-1-eq-Suc-0* [*code-post*]: *Numeral1* = *Suc 0*
⟨*proof*⟩

31.5 Function *int*: Coercion from Type *nat* to *int*

lemma *int-nat-number-of* [*simp*]:
int (*number-of v*) =
(*if neg* (*number-of v* :: *int*) *then 0*
else (*number-of v* :: *int*))
⟨*proof*⟩

31.5.1 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-of v* + *n*) =
(*if neg* (*number-of v* :: *int*) *then 1*+*n else number-of* (*Int.succ v*) + *n*)
⟨*proof*⟩

lemma *Suc-nat-number-of* [*simp*]:
Suc (*number-of v*) =
(*if neg* (*number-of v* :: *int*) *then 1 else number-of* (*Int.succ v*))
⟨*proof*⟩

31.5.2 Addition

lemma *add-nat-number-of* [*simp*]:
(*number-of v* :: *nat*) + *number-of v'* =
(*if v* < *Int.Pls then number-of v'*
else if v' < *Int.Pls then number-of v*
else number-of (*v* + *v'*))
⟨*proof*⟩

lemma *nat-number-of-add-1* [*simp*]:
number-of v + (*1*::*nat*) =
(*if v* < *Int.Pls then 1 else number-of* (*Int.succ v*))
⟨*proof*⟩

lemma *nat-1-add-number-of* [*simp*]:
(*1*::*nat*) + *number-of v* =
(*if v* < *Int.Pls then 1 else number-of* (*Int.succ v*))
⟨*proof*⟩

lemma *nat-1-add-1* [*simp*]: *1* + *1* = (*2*::*nat*)
⟨*proof*⟩

31.5.3 Subtraction

lemma *diff-nat-eq-if*:
nat z − *nat z'* =
(*if neg z'* *then nat z*
else let d = *z*−*z'* *in*
if neg d then 0 else nat d)
⟨*proof*⟩

lemma *diff-nat-number-of* [*simp*]:
(*number-of v* :: *nat*) − *number-of v'* =
(*if v'* < *Int.Pls then number-of v*
else let d = *number-of* (*v* + *uminus v'*) *in*
if neg d then 0 else nat d)
⟨*proof*⟩

lemma *nat-number-of-diff-1* [*simp*]:
number-of v − (1::nat) =
(if v ≤ Int.Pls then 0 else number-of (Int.pred v))
⟨*proof*⟩

31.5.4 Multiplication

lemma *mult-nat-number-of* [*simp*]:
(number-of v :: nat) ∗ number-of v′ =
(if v < Int.Pls then 0 else number-of (v ∗ v′))
⟨*proof*⟩

31.6 Comparisons

31.6.1 Equals (=)

lemma *eq-nat-number-of* [*simp*]:
((number-of v :: nat) = number-of v′) =
(if neg (number-of v :: int) then (number-of v′ :: int) ≤ 0
else if neg (number-of v′ :: int) then (number-of v :: int) = 0
else v = v′)
⟨*proof*⟩

31.6.2 Less-than (¡)

lemma *less-nat-number-of* [*simp*]:
(number-of v :: nat) < number-of v′ ⟷
(if v < v′ then Int.Pls < v′ else False)
⟨*proof*⟩

31.6.3 Less-than-or-equal

lemma *le-nat-number-of* [*simp*]:
(number-of v :: nat) ≤ number-of v′ ⟷
(if v ≤ v′ then True else v ≤ Int.Pls)
⟨*proof*⟩

lemmas *numerals = nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2*

31.7 Powers with Numeric Exponents

Squares of literal numerals will be evaluated.

lemmas *power2-eq-square-number-of* [*simp*] =
power2-eq-square [*of number-of w*, *standard*]

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

31.7.1 Nat

lemma *Suc-pred'*: $0 < n \Longrightarrow n = Suc(n - 1)$
⟨*proof*⟩

lemmas *expand-Suc* = *Suc-pred'* [*of number-of v*, *standard*]

31.7.2 Arith

lemma *Suc-eq-plus1*: $Suc\ n = n + 1$
⟨*proof*⟩

lemma *Suc-eq-plus1-left*: $Suc\ n = 1 + n$
⟨*proof*⟩

lemma *add-eq-if*: $(m::nat) + n = (if\ m=0\ then\ n\ else\ Suc\ ((m - 1) + n))$
⟨*proof*⟩

lemma *mult-eq-if*: $(m::nat) * n = (if\ m=0\ then\ 0\ else\ n + ((m - 1) * n))$
⟨*proof*⟩

lemma *power-eq-if*: $(p\ \hat{}\ m :: nat) = (if\ m=0\ then\ 1\ else\ p * (p\ \hat{}\ (m - 1)))$
⟨*proof*⟩

31.8 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 $= (0::nat) \longleftrightarrow v \leq Int.Pls$
⟨*proof*⟩

lemma *eq-0-number-of* [*simp*]:
$(0::nat) =$ *number-of v* $\longleftrightarrow v \leq Int.Pls$
⟨*proof*⟩

lemma *less-0-number-of* [*simp*]:
$(0::nat) <$ *number-of v* $\longleftrightarrow Int.Pls < v$
⟨*proof*⟩

lemma *neg-imp-number-of-eq-0*: *neg* (*number-of v* :: *int*) ==> *number-of v* = (*0*::*nat*)
⟨*proof*⟩

31.9 Comparisons involving *Suc*

lemma *eq-number-of-Suc* [*simp*]:
(*number-of v* = *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
⟨*proof*⟩

lemma *Suc-eq-number-of* [*simp*]:
(*Suc n* = *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
⟨*proof*⟩

lemma *less-number-of-Suc* [*simp*]:
(*number-of v* < *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv < *n*)
⟨*proof*⟩

lemma *less-Suc-number-of* [*simp*]:
(*Suc n* < *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else n < *nat pv*)
⟨*proof*⟩

lemma *le-number-of-Suc* [*simp*]:
(*number-of v* <= *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv <= *n*)
⟨*proof*⟩

lemma *le-Suc-number-of* [*simp*]:
(*Suc n* <= *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else n <= *nat pv*)
⟨*proof*⟩

lemma *eq-number-of-Pls-Min*: (*Numeral0* ::*int*) ~= *number-of Int.Min*
⟨*proof*⟩

31.10 Max and Min Combined with *Suc*

lemma *max-number-of-Suc* [*simp*]:
max (*Suc n*) (*number-of v*) =

(*let pv = number-of (Int.pred v) in*
if neg pv then Suc n else Suc(max n (nat pv)))
⟨*proof*⟩

lemma *max-Suc-number-of* [*simp*]:
max (number-of v) (Suc n) =
(*let pv = number-of (Int.pred v) in*
if neg pv then Suc n else Suc(max (nat pv) n))
⟨*proof*⟩

lemma *min-number-of-Suc* [*simp*]:
min (Suc n) (number-of v) =
(*let pv = number-of (Int.pred v) in*
if neg pv then 0 else Suc(min n (nat pv)))
⟨*proof*⟩

lemma *min-Suc-number-of* [*simp*]:
min (number-of v) (Suc n) =
(*let pv = number-of (Int.pred v) in*
if neg pv then 0 else Suc(min (nat pv) n))
⟨*proof*⟩

31.11 Literal arithmetic involving powers

lemma *power-nat-number-of*:
(*number-of v :: nat*) ^ *n =*
(*if neg (number-of v :: int) then 0^n else nat ((number-of v :: int) ^ n)*)
⟨*proof*⟩

lemmas *power-nat-number-of-number-of = power-nat-number-of* [*of - number-of w, standard*]
declare *power-nat-number-of-number-of* [*simp*]

For arbitrary rings

lemma *power-number-of-even*:
fixes *z :: 'a::number-ring*
shows *z ^ number-of (Int.Bit0 w) = (let w = z ^ (number-of w) in w * w)*
⟨*proof*⟩

lemma *power-number-of-odd*:
fixes *z :: 'a::number-ring*
shows *z ^ number-of (Int.Bit1 w) = (if (0::int) <= number-of w*
*then (let w = z ^ (number-of w) in z * w * w) else 1)*
⟨*proof*⟩

lemmas *zpower-number-of-even = power-number-of-even* [**where** *'a=int*]
lemmas *zpower-number-of-odd = power-number-of-odd* [**where** *'a=int*]

lemmas *power-number-of-even-number-of* [*simp*] =
power-number-of-even [*of number-of v*, *standard*]

lemmas *power-number-of-odd-number-of* [*simp*] =
power-number-of-odd [*of number-of v*, *standard*]

lemma *nat-number-of-Pls*: *Numeral0* = (*0*::*nat*)
⟨*proof*⟩

lemma *nat-number-of-Min*: *number-of Int.Min* = (*0*::*nat*)
⟨*proof*⟩

lemma *nat-number-of-Bit0*:
number-of (*Int.Bit0 w*) = (*let n*::*nat* = *number-of w in n* + *n*)
⟨*proof*⟩

lemma *nat-number-of-Bit1*:
number-of (*Int.Bit1 w*) =
(*if neg* (*number-of w* :: *int*) *then 0*
else let n = *number-of w in Suc* (*n* + *n*))
⟨*proof*⟩

lemmas *nat-number* =
nat-number-of-Pls nat-number-of-Min
nat-number-of-Bit0 nat-number-of-Bit1

lemmas *nat-number'* =
nat-number-of-Bit0 nat-number-of-Bit1

lemmas *nat-arith* =
add-nat-number-of
diff-nat-number-of
mult-nat-number-of
eq-nat-number-of
less-nat-number-of

lemmas *semiring-norm* =
Let-def arith-simps nat-arith rel-simps neg-simps if-False
if-True add-0 add-Suc add-number-of-left mult-number-of-left
numeral-1-eq-1 [*symmetric*] *Suc-eq-plus1*
numeral-0-eq-0 [*symmetric*] *numerals* [*symmetric*]
not-iszero-Numeral1

lemma *Let-Suc* [*simp*]: *Let* (*Suc n*) *f* == *f* (*Suc n*)
⟨*proof*⟩

lemma *power-m1-even*: (−*1*) ^ (*2*∗*n*) = (*1*::*'a*::{*number-ring*})
⟨*proof*⟩

lemma *power-m1-odd*: $(-1)\ \hat{}\ Suc(2*n) = (-1::'a::\{number\text{-}ring\})$
⟨*proof*⟩

lemma *nat-number-of-add-left*:
$number\text{-}of\ v + (number\text{-}of\ v' + (k::nat)) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ number\text{-}of\ v' + k$
$else\ if\ neg\ (number\text{-}of\ v' :: int)\ then\ number\text{-}of\ v + k$
$else\ number\text{-}of\ (v + v') + k)$
⟨*proof*⟩

lemma *nat-number-of-mult-left*:
$number\text{-}of\ v * (number\text{-}of\ v' * (k::nat)) =$
$(if\ v < Int.Pls\ then\ 0$
$else\ number\text{-}of\ (v * v') * k)$
⟨*proof*⟩

31.12 Literal arithmetic and *of-nat*

lemma *of-nat-double*:
$0 \leq x \Longrightarrow of\text{-}nat\ (nat\ (2 * x)) = of\text{-}nat\ (nat\ x) + of\text{-}nat\ (nat\ x)$
⟨*proof*⟩

lemma *nat-numeral-m1-eq-0*: $-1 = (0::nat)$
⟨*proof*⟩

lemma *of-nat-number-of-lemma*:
$of\text{-}nat\ (number\text{-}of\ v :: nat) =$
$(if\ 0 \leq (number\text{-}of\ v :: int)$
$then\ (number\text{-}of\ v :: 'a :: number\text{-}ring)$
$else\ 0)$
⟨*proof*⟩

lemma *of-nat-number-of-eq* [*simp*]:
$of\text{-}nat\ (number\text{-}of\ v :: nat) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ 0$
$else\ (number\text{-}of\ v :: 'a :: number\text{-}ring))$
⟨*proof*⟩

31.12.1 For simplifying *Suc m − K* and *K − Suc m*

Where K above is a literal

lemma *Suc-diff-eq-diff-pred*: $Numeral0 < n \Longrightarrow Suc\ m - n = m - (n - Numeral1)$
⟨*proof*⟩

Now just instantiating *n* to *number-of v* does the right simplification, but with some redundant inequality tests.

lemma *neg-number-of-pred-iff-0*:
$neg\ (number\text{-}of\ (Int.pred\ v)::int) = (number\text{-}of\ v = (0::nat))$

⟨*proof*⟩

No longer required as a simprule because of the *inverse-fold* simproc

lemma *Suc-diff-number-of*:
Int.Pls < *v* ==>
Suc m − (*number-of v*) = *m* − (*number-of* (*Int.pred v*))
⟨*proof*⟩

lemma *diff-Suc-eq-diff-pred*: *m* − *Suc n* = (*m* − *1*) − *n*
⟨*proof*⟩

31.12.2 For *nat-case* and *nat-rec*

lemma *nat-case-number-of* [*simp*]:
nat-case a f (*number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then a else f (*nat pv*))
⟨*proof*⟩

lemma *nat-case-add-eq-if* [*simp*]:
nat-case a f ((*number-of v*) + *n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then nat-case a f n else f (*nat pv* + *n*))
⟨*proof*⟩

lemma *nat-rec-number-of* [*simp*]:
nat-rec a f (*number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then a else f (*nat pv*) (*nat-rec a f* (*nat pv*)))
⟨*proof*⟩

lemma *nat-rec-add-eq-if* [*simp*]:
nat-rec a f (*number-of v* + *n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then nat-rec a f n
else f (*nat pv* + *n*) (*nat-rec a f* (*nat pv* + *n*)))
⟨*proof*⟩

31.12.3 Various Other Lemmas

lemma *card-UNIV-bool*[*simp*]: *card* (*UNIV* :: *bool set*) = *2*
⟨*proof*⟩

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*⟩

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*⟩

end

32 Nat-Transfer: Generic transfer machinery; specific transfer from nats to ints and back.

theory *Nat-Transfer*
imports *Nat-Numeral*
uses (*Tools/transfer.ML*)
begin

32.1 Generic transfer machinery

definition *transfer-morphism*:: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow bool) \Rightarrow bool$
  where $transfer\text{-}morphism\ f\ A \longleftrightarrow (\forall P.\ (\forall x.\ P\ x) \longrightarrow (\forall y.\ A\ y \longrightarrow P\ (f\ y)))$

lemma *transfer-morphismI*:
  assumes $\bigwedge P\ y.\ (\bigwedge x.\ P\ x) \Longrightarrow A\ y \Longrightarrow P\ (f\ y)$
  shows *transfer-morphism f A*
  ⟨*proof*⟩

⟨*ML*⟩

32.2 Set up transfer from nat to int

set up transfer direction

lemma *transfer-morphism-nat-int*: *transfer-morphism nat* (*op* <= (*0*::*int*))
  ⟨*proof*⟩

declare *transfer-morphism-nat-int* [*transfer add*
mode: *manual*
return: *nat-0-le*
labels: *nat-int*
]

basic functions and relations

lemma *transfer-nat-int-numerals* [*transfer key*: *transfer-morphism-nat-int*]:
$(0::nat) = nat\ 0$
$(1::nat) = nat\ 1$
$(2::nat) = nat\ 2$
$(3::nat) = nat\ 3$
⟨*proof*⟩

definition
tsub :: *int* ⇒ *int* ⇒ *int*
where
$tsub\ x\ y = (if\ x >= y\ then\ x - y\ else\ 0)$

lemma *tsub-eq*: $x >= y \Longrightarrow tsub\ x\ y = x - y$
⟨*proof*⟩

lemma *transfer-nat-int-functions* [*transfer key*: *transfer-morphism-nat-int*]:
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow (nat\ x) + (nat\ y) = nat\ (x + y)$
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow (nat\ x) * (nat\ y) = nat\ (x * y)$
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow (nat\ x) - (nat\ y) = nat\ (tsub\ x\ y)$
$(x::int) >= 0 \Longrightarrow (nat\ x)\hat{}n = nat\ (x\hat{}n)$
⟨*proof*⟩

lemma *transfer-nat-int-function-closures* [*transfer key*: *transfer-morphism-nat-int*]:
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow x + y >= 0$
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow x * y >= 0$
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow tsub\ x\ y >= 0$
$(x::int) >= 0 \Longrightarrow x\hat{}n >= 0$
$(0::int) >= 0$
$(1::int) >= 0$
$(2::int) >= 0$
$(3::int) >= 0$
$int\ z >= 0$
⟨*proof*⟩

lemma *transfer-nat-int-relations* [*transfer key*: *transfer-morphism-nat-int*]:
$x >= 0 \Longrightarrow y >= 0 \Longrightarrow$
$(nat\ (x::int) = nat\ y) = (x = y)$
$x >= 0 \Longrightarrow y >= 0 \Longrightarrow$
$(nat\ (x::int) < nat\ y) = (x < y)$
$x >= 0 \Longrightarrow y >= 0 \Longrightarrow$
$(nat\ (x::int) <= nat\ y) = (x <= y)$
$x >= 0 \Longrightarrow y >= 0 \Longrightarrow$

$(nat\ (x::int)\ dvd\ nat\ y) = (x\ dvd\ y)$
⟨*proof*⟩

first-order quantifiers

lemma *all-nat*: $(\forall x.\ P\ x) \longleftrightarrow (\forall x{\geq}0.\ P\ (nat\ x))$
⟨*proof*⟩

lemma *ex-nat*: $(\exists x.\ P\ x) \longleftrightarrow (\exists x.\ 0 \leq x \wedge P\ (nat\ x))$
⟨*proof*⟩

lemma *transfer-nat-int-quantifiers* [*transfer key*: *transfer-morphism-nat-int*]:
$(ALL\ (x::nat).\ P\ x) = (ALL\ (x::int).\ x >= 0 \longrightarrow P\ (nat\ x))$
$(EX\ (x::nat).\ P\ x) = (EX\ (x::int).\ x >= 0\ \&\ P\ (nat\ x))$
⟨*proof*⟩

lemma *all-cong*: $(\bigwedge x.\ Q\ x \Longrightarrow P\ x = P'\ x) \Longrightarrow$
$(ALL\ x.\ Q\ x \longrightarrow P\ x) = (ALL\ x.\ Q\ x \longrightarrow P'\ x)$
⟨*proof*⟩

lemma *ex-cong*: $(\bigwedge x.\ Q\ x \Longrightarrow P\ x = P'\ x) \Longrightarrow$
$(EX\ x.\ Q\ x \wedge P\ x) = (EX\ x.\ Q\ x \wedge P'\ x)$
⟨*proof*⟩

declare *transfer-morphism-nat-int* [*transfer add*
cong: *all-cong ex-cong*]

if

lemma *nat-if-cong* [*transfer key*: *transfer-morphism-nat-int*]:
$(if\ P\ then\ (nat\ x)\ else\ (nat\ y)) = nat\ (if\ P\ then\ x\ else\ y)$
⟨*proof*⟩

operations with sets

definition
nat-set :: *int set* ⇒ *bool*
where
$nat\text{-}set\ S = (ALL\ x{:}S.\ x >= 0)$

lemma *transfer-nat-int-set-functions*:
$card\ A = card\ (int\ `\ A)$
$\{\} = nat\ `\ (\{\}::int\ set)$
$A\ Un\ B = nat\ `\ (int\ `\ A\ Un\ int\ `\ B)$
$A\ Int\ B = nat\ `\ (int\ `\ A\ Int\ int\ `\ B)$
$\{x.\ P\ x\} = nat\ `\ \{x.\ x >= 0\ \&\ P(nat\ x)\}$
⟨*proof*⟩

lemma *transfer-nat-int-set-function-closures*:
nat-set {}
$nat\text{-}set\ A \Longrightarrow nat\text{-}set\ B \Longrightarrow nat\text{-}set\ (A\ Un\ B)$

nat-set A ⟹ nat-set B ⟹ nat-set (A Int B)
nat-set {x. x >= 0 & P x}
nat-set (int ' C)
nat-set A ⟹ x : A ⟹ x >= 0
⟨*proof*⟩

lemma *transfer-nat-int-set-relations*:
(*finite A*) = (*finite* (*int ' A*))
(*x : A*) = (*int x : int ' A*)
(*A = B*) = (*int ' A = int ' B*)
(*A < B*) = (*int ' A < int ' B*)
(*A <= B*) = (*int ' A <= int ' B*)
⟨*proof*⟩

lemma *transfer-nat-int-set-return-embed*: *nat-set A ⟹*
(*int ' nat ' A = A*)
⟨*proof*⟩

lemma *transfer-nat-int-set-cong*: (!!*x*. *x >= 0 ⟹ P x = P' x*) *⟹*
{(*x*::*int*). *x >= 0 & P x*} = {*x*. *x >= 0 & P' x*}
⟨*proof*⟩

declare *transfer-morphism-nat-int* [*transfer add*
return: *transfer-nat-int-set-functions*
transfer-nat-int-set-function-closures
transfer-nat-int-set-relations
transfer-nat-int-set-return-embed
cong: *transfer-nat-int-set-cong*
]

setsum and setprod

lemma *transfer-nat-int-sum-prod*:
setsum f A = setsum (%*x*. *f* (*nat x*)) (*int ' A*)
setprod f A = setprod (%*x*. *f* (*nat x*)) (*int ' A*)
⟨*proof*⟩

lemma *transfer-nat-int-sum-prod2*:
setsum f A = nat(*setsum* (%*x*. *int* (*f x*)) *A*)
setprod f A = nat(*setprod* (%*x*. *int* (*f x*)) *A*)
⟨*proof*⟩

lemma *transfer-nat-int-sum-prod-closure*:
nat-set A ⟹ (!!*x*. *x >= 0 ⟹ f x >=* (*0*::*int*)) *⟹ setsum f A >= 0*
nat-set A ⟹ (!!*x*. *x >= 0 ⟹ f x >=* (*0*::*int*)) *⟹ setprod f A >= 0*
⟨*proof*⟩

lemma *transfer-nat-int-sum-prod-cong*:
$A = B \Longrightarrow$ *nat-set* $B \Longrightarrow (!!x.\ x >= 0 \Longrightarrow f\ x = g\ x) \Longrightarrow$
setsum $f\ A =$ *setsum* $g\ B$
$A = B \Longrightarrow$ *nat-set* $B \Longrightarrow (!!x.\ x >= 0 \Longrightarrow f\ x = g\ x) \Longrightarrow$
setprod $f\ A =$ *setprod* $g\ B$
⟨*proof*⟩

declare *transfer-morphism-nat-int* [*transfer add*
return: *transfer-nat-int-sum-prod transfer-nat-int-sum-prod2*
transfer-nat-int-sum-prod-closure
cong: *transfer-nat-int-sum-prod-cong*]

32.3 Set up transfer from int to nat

set up transfer direction

lemma *transfer-morphism-int-nat*: *transfer-morphism int* ($\lambda n.$ *True*)
⟨*proof*⟩

declare *transfer-morphism-int-nat* [*transfer add*
mode: *manual*
return: *nat-int*
labels: *int-nat*
]

basic functions and relations

definition
is-nat :: *int* ⇒ *bool*
where
is-nat $x = (x >= 0)$

lemma *transfer-int-nat-numerals*:
$0 =$ *int* 0
$1 =$ *int* 1
$2 =$ *int* 2
$3 =$ *int* 3
⟨*proof*⟩

lemma *transfer-int-nat-functions*:
(*int* x) + (*int* y) = *int* $(x + y)$
(*int* x) ∗ (*int* y) = *int* $(x * y)$
tsub (*int* x) (*int* y) = *int* $(x - y)$
(*int* x) ^n = *int* $(x\hat{}n)$
⟨*proof*⟩

lemma *transfer-int-nat-function-closures*:
is-nat $x \Longrightarrow$ *is-nat* $y \Longrightarrow$ *is-nat* $(x + y)$
is-nat $x \Longrightarrow$ *is-nat* $y \Longrightarrow$ *is-nat* $(x * y)$

is-nat x $\Longrightarrow$ *is-nat y* $\Longrightarrow$ *is-nat* (*tsub x y*)
is-nat x $\Longrightarrow$ *is-nat* (*x^n*)
is-nat 0
is-nat 1
is-nat 2
is-nat 3
is-nat (*int z*)
⟨*proof*⟩

lemma *transfer-int-nat-relations*:
(*int x* = *int y*) = (*x* = *y*)
(*int x* < *int y*) = (*x* < *y*)
(*int x* <= *int y*) = (*x* <= *y*)
(*int x dvd int y*) = (*x dvd y*)
⟨*proof*⟩

declare *transfer-morphism-int-nat* [*transfer add return*:
transfer-int-nat-numerals
transfer-int-nat-functions
transfer-int-nat-function-closures
transfer-int-nat-relations
]

first-order quantifiers

lemma *transfer-int-nat-quantifiers*:
(*ALL* (*x*::*int*) >= *0*. *P x*) = (*ALL* (*x*::*nat*). *P* (*int x*))
(*EX* (*x*::*int*) >= *0*. *P x*) = (*EX* (*x*::*nat*). *P* (*int x*))
⟨*proof*⟩

declare *transfer-morphism-int-nat* [*transfer add*
return: *transfer-int-nat-quantifiers*]

if

lemma *int-if-cong*: (*if P then* (*int x*) *else* (*int y*)) =
int (*if P then x else y*)
⟨*proof*⟩

declare *transfer-morphism-int-nat* [*transfer add return*: *int-if-cong*]

operations with sets

lemma *transfer-int-nat-set-functions*:
nat-set A $\Longrightarrow$ *card A* = *card* (*nat* ‘ *A*)
{} = *int* ‘ ({}::*nat set*)
nat-set A $\Longrightarrow$ *nat-set B* $\Longrightarrow$ *A Un B* = *int* ‘ (*nat* ‘ *A Un nat* ‘ *B*)
nat-set A $\Longrightarrow$ *nat-set B* $\Longrightarrow$ *A Int B* = *int* ‘ (*nat* ‘ *A Int nat* ‘ *B*)
{*x*. *x* >= *0* & *P x*} = *int* ‘ {*x*. *P*(*int x*)}

⟨*proof*⟩

lemma *transfer-int-nat-set-function-closures*:
nat-set {}
nat-set A ⟹ *nat-set B* ⟹ *nat-set* (*A Un B*)
nat-set A ⟹ *nat-set B* ⟹ *nat-set* (*A Int B*)
nat-set {*x. x >= 0 & P x*}
nat-set (*int ' C*)
nat-set A ⟹ *x* : *A* ⟹ *is-nat x*
⟨*proof*⟩

lemma *transfer-int-nat-set-relations*:
nat-set A ⟹ *finite A* = *finite* (*nat ' A*)
is-nat x ⟹ *nat-set A* ⟹ (*x* : *A*) = (*nat x* : *nat ' A*)
nat-set A ⟹ *nat-set B* ⟹ (*A* = *B*) = (*nat ' A* = *nat ' B*)
nat-set A ⟹ *nat-set B* ⟹ (*A* < *B*) = (*nat ' A* < *nat ' B*)
nat-set A ⟹ *nat-set B* ⟹ (*A* <= *B*) = (*nat ' A* <= *nat ' B*)
⟨*proof*⟩

lemma *transfer-int-nat-set-return-embed*: *nat ' int ' A* = *A*
⟨*proof*⟩

lemma *transfer-int-nat-set-cong*: (!!*x. P x* = *P' x*) ⟹
{(*x*::*nat*). *P x*} = {*x. P' x*}
⟨*proof*⟩

declare *transfer-morphism-int-nat* [*transfer add*
return: *transfer-int-nat-set-functions*
transfer-int-nat-set-function-closures
transfer-int-nat-set-relations
transfer-int-nat-set-return-embed
cong: *transfer-int-nat-set-cong*
]

setsum and setprod

lemma *transfer-int-nat-sum-prod*:
nat-set A ⟹ *setsum f A* = *setsum* (%*x. f* (*int x*)) (*nat ' A*)
nat-set A ⟹ *setprod f A* = *setprod* (%*x. f* (*int x*)) (*nat ' A*)
⟨*proof*⟩

lemma *transfer-int-nat-sum-prod2*:
(!!*x. x*:*A* ⟹ *is-nat* (*f x*)) ⟹ *setsum f A* = *int*(*setsum* (%*x. nat* (*f x*)) *A*)
(!!*x. x*:*A* ⟹ *is-nat* (*f x*)) ⟹
setprod f A = *int*(*setprod* (%*x. nat* (*f x*)) *A*)
⟨*proof*⟩

declare *transfer-morphism-int-nat* [*transfer add*
return: *transfer-int-nat-sum-prod transfer-int-nat-sum-prod2*
cong: *setsum-cong setprod-cong*]

end

33 Divides: The division operators div and mod

theory *Divides*
imports *Nat-Numeral Nat-Transfer*
uses ~~/*src*/*Provers*/*Arith*/*cancel-div-mod.ML*
begin

33.1 Syntactic division operations

class *div* = *dvd* +
  fixes *div* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** *div* *70*)
    and *mod* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** *mod* *70*)

33.2 Abstract division in commutative semirings.

class *semiring-div* = *comm-semiring-1-cancel* + *no-zero-divisors* + *div* +
  assumes *mod-div-equality*: *a div b* ∗ *b* + *a mod b* = *a*
    and *div-by-0* [*simp*]: *a div 0* = *0*
    and *div-0* [*simp*]: *0 div a* = *0*
    and *div-mult-self1* [*simp*]: *b* ≠ *0* ⟹ (*a* + *c* ∗ *b*) *div b* = *c* + *a div b*
    and *div-mult-mult1* [*simp*]: *c* ≠ *0* ⟹ (*c* ∗ *a*) *div* (*c* ∗ *b*) = *a div b*
begin

op div and *op mod*

lemma *mod-div-equality2*: *b* ∗ (*a div b*) + *a mod b* = *a*
  ⟨*proof*⟩

lemma *mod-div-equality'*: *a mod b* + *a div b* ∗ *b* = *a*
  ⟨*proof*⟩

lemma *div-mod-equality*: ((*a div b*) ∗ *b* + *a mod b*) + *c* = *a* + *c*
  ⟨*proof*⟩

lemma *div-mod-equality2*: (*b* ∗ (*a div b*) + *a mod b*) + *c* = *a* + *c*
  ⟨*proof*⟩

lemma *mod-by-0* [*simp*]: *a mod 0* = *a*
  ⟨*proof*⟩

lemma *mod-0* [*simp*]: *0 mod a* = *0*
  ⟨*proof*⟩

lemma *div-mult-self2* [*simp*]:
  assumes *b* ≠ *0*
  shows (*a* + *b* ∗ *c*) *div b* = *c* + *a div b*
  ⟨*proof*⟩

lemma *mod-mult-self1* [*simp*]: $(a + c * b)\ mod\ b = a\ mod\ b$
⟨*proof*⟩

lemma *mod-mult-self2* [*simp*]: $(a + b * c)\ mod\ b = a\ mod\ b$
⟨*proof*⟩

lemma *div-mult-self1-is-id* [*simp*]: $b \neq 0 \Longrightarrow b * a\ div\ b = a$
⟨*proof*⟩

lemma *div-mult-self2-is-id* [*simp*]: $b \neq 0 \Longrightarrow a * b\ div\ b = a$
⟨*proof*⟩

lemma *mod-mult-self1-is-0* [*simp*]: $b * a\ mod\ b = 0$
⟨*proof*⟩

lemma *mod-mult-self2-is-0* [*simp*]: $a * b\ mod\ b = 0$
⟨*proof*⟩

lemma *div-by-1* [*simp*]: $a\ div\ 1 = a$
⟨*proof*⟩

lemma *mod-by-1* [*simp*]: $a\ mod\ 1 = 0$
⟨*proof*⟩

lemma *mod-self* [*simp*]: $a\ mod\ a = 0$
⟨*proof*⟩

lemma *div-self* [*simp*]: $a \neq 0 \Longrightarrow a\ div\ a = 1$
⟨*proof*⟩

lemma *div-add-self1* [*simp*]:
assumes $b \neq 0$
shows $(b + a)\ div\ b = a\ div\ b + 1$
⟨*proof*⟩

lemma *div-add-self2* [*simp*]:
assumes $b \neq 0$
shows $(a + b)\ div\ b = a\ div\ b + 1$
⟨*proof*⟩

lemma *mod-add-self1* [*simp*]:
$(b + a)\ mod\ b = a\ mod\ b$
⟨*proof*⟩

lemma *mod-add-self2* [*simp*]:
$(a + b)\ mod\ b = a\ mod\ b$
⟨*proof*⟩

lemma *mod-div-decomp*:
fixes *a b*
obtains *q r* **where** $q = a \mathbin{div} b$ **and** $r = a \bmod b$
and $a = q * b + r$
⟨*proof*⟩

lemma *dvd-eq-mod-eq-0* [*code*, *code-unfold*, *code-inline del*]: $a \mathbin{dvd} b \longleftrightarrow b \bmod a = 0$
⟨*proof*⟩

lemma *mod-div-trivial* [*simp*]: $a \bmod b \mathbin{div} b = 0$
⟨*proof*⟩

lemma *mod-mod-trivial* [*simp*]: $a \bmod b \bmod b = a \bmod b$
⟨*proof*⟩

lemma *dvd-imp-mod-0*: $a \mathbin{dvd} b \Longrightarrow b \bmod a = 0$
⟨*proof*⟩

lemma *dvd-div-mult-self*: $a \mathbin{dvd} b \Longrightarrow (b \mathbin{div} a) * a = b$
⟨*proof*⟩

lemma *dvd-mult-div-cancel*: $a \mathbin{dvd} b \Longrightarrow a * (b \mathbin{div} a) = b$
⟨*proof*⟩

lemma *dvd-div-mult*: $a \mathbin{dvd} b \Longrightarrow (b \mathbin{div} a) * c = b * c \mathbin{div} a$
⟨*proof*⟩

lemma *div-dvd-div*[*simp*]:
$a \mathbin{dvd} b \Longrightarrow a \mathbin{dvd} c \Longrightarrow (b \mathbin{div} a \mathbin{dvd} c \mathbin{div} a) = (b \mathbin{dvd} c)$
⟨*proof*⟩

lemma *dvd-mod-imp-dvd*: [| $k \mathbin{dvd} m \bmod n$; $k \mathbin{dvd} n$ |] ==> $k \mathbin{dvd} m$
⟨*proof*⟩

Addition respects modular equivalence.

lemma *mod-add-left-eq*: $(a + b) \bmod c = (a \bmod c + b) \bmod c$
⟨*proof*⟩

lemma *mod-add-right-eq*: $(a + b) \bmod c = (a + b \bmod c) \bmod c$
⟨*proof*⟩

lemma *mod-add-eq*: $(a + b) \bmod c = (a \bmod c + b \bmod c) \bmod c$
⟨*proof*⟩

lemma *mod-add-cong*:
assumes $a \bmod c = a' \bmod c$
assumes $b \bmod c = b' \bmod c$
shows $(a + b) \bmod c = (a' + b') \bmod c$

⟨*proof*⟩

lemma *div-add* [*simp*]: *z dvd x* ⟹ *z dvd y*
⟹ (*x* + *y*) *div z* = *x div z* + *y div z*
⟨*proof*⟩

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq*: (*a* ∗ *b*) *mod c* = ((*a mod c*) ∗ *b*) *mod c*
⟨*proof*⟩

lemma *mod-mult-right-eq*: (*a* ∗ *b*) *mod c* = (*a* ∗ (*b mod c*)) *mod c*
⟨*proof*⟩

lemma *mod-mult-eq*: (*a* ∗ *b*) *mod c* = ((*a mod c*) ∗ (*b mod c*)) *mod c*
⟨*proof*⟩

lemma *mod-mult-cong*:
assumes *a mod c* = *a*′ *mod c*
assumes *b mod c* = *b*′ *mod c*
shows (*a* ∗ *b*) *mod c* = (*a*′ ∗ *b*′) *mod c*
⟨*proof*⟩

lemma *mod-mod-cancel*:
assumes *c dvd b*
shows *a mod b mod c* = *a mod c*
⟨*proof*⟩

lemma *div-mult-div-if-dvd*:
y dvd x ⟹ *z dvd w* ⟹ (*x div y*) ∗ (*w div z*) = (*x* ∗ *w*) *div* (*y* ∗ *z*)
⟨*proof*⟩

lemma *div-mult-swap*:
assumes *c dvd b*
shows *a* ∗ (*b div c*) = (*a* ∗ *b*) *div c*
⟨*proof*⟩

lemma *div-mult-mult2* [*simp*]:
c ≠ *0* ⟹ (*a* ∗ *c*) *div* (*b* ∗ *c*) = *a div b*
⟨*proof*⟩

lemma *div-mult-mult1-if* [*simp*]:
(*c* ∗ *a*) *div* (*c* ∗ *b*) = (*if c* = *0 then 0 else a div b*)
⟨*proof*⟩

lemma *mod-mult-mult1*:
(*c* ∗ *a*) *mod* (*c* ∗ *b*) = *c* ∗ (*a mod b*)
⟨*proof*⟩

lemma *mod-mult-mult2*:

$(a * c)\ mod\ (b * c) = (a\ mod\ b) * c$
⟨*proof*⟩

lemma *dvd-mod*: $k\ dvd\ m \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ (m\ mod\ n)$
⟨*proof*⟩

lemma *dvd-mod-iff*: $k\ dvd\ n \Longrightarrow k\ dvd\ (m\ mod\ n) \longleftrightarrow k\ dvd\ m$
⟨*proof*⟩

lemma *div-power*:
$y\ dvd\ x \Longrightarrow (x\ div\ y)\ \hat{}\ n = x\ \hat{}\ n\ div\ y\ \hat{}\ n$
⟨*proof*⟩

lemma *dvd-div-eq-mult*:
assumes $a \neq 0$ **and** $a\ dvd\ b$
shows $b\ div\ a = c \longleftrightarrow b = c * a$
⟨*proof*⟩

lemma *dvd-div-div-eq-mult*:
assumes $a \neq 0\ c \neq 0$ **and** $a\ dvd\ b\ c\ dvd\ d$
shows $b\ div\ a = d\ div\ c \longleftrightarrow b * c = a * d$
⟨*proof*⟩

end

class *ring-div* = *semiring-div* + *comm-ring-1*
begin

subclass *ring-1-no-zero-divisors* ⟨*proof*⟩

Negation respects modular equivalence.

lemma *mod-minus-eq*: $(-\ a)\ mod\ b = (-\ (a\ mod\ b))\ mod\ b$
⟨*proof*⟩

lemma *mod-minus-cong*:
assumes $a\ mod\ b = a'\ mod\ b$
shows $(-\ a)\ mod\ b = (-\ a')\ mod\ b$
⟨*proof*⟩

Subtraction respects modular equivalence.

lemma *mod-diff-left-eq*: $(a - b)\ mod\ c = (a\ mod\ c - b)\ mod\ c$
⟨*proof*⟩

lemma *mod-diff-right-eq*: $(a - b)\ mod\ c = (a - b\ mod\ c)\ mod\ c$
⟨*proof*⟩

lemma *mod-diff-eq*: $(a - b)\ mod\ c = (a\ mod\ c - b\ mod\ c)\ mod\ c$
⟨*proof*⟩

lemma *mod-diff-cong*:
assumes $a\ mod\ c = a'\ mod\ c$
assumes $b\ mod\ c = b'\ mod\ c$
shows $(a - b)\ mod\ c = (a' - b')\ mod\ c$
⟨*proof*⟩

lemma *dvd-neg-div*: $y\ dvd\ x \Longrightarrow -x\ div\ y = -\ (x\ div\ y)$
⟨*proof*⟩

lemma *dvd-div-neg*: $y\ dvd\ x \Longrightarrow x\ div\ -y = -\ (x\ div\ y)$
⟨*proof*⟩

end

33.3 Division on *nat*

We define *op div* and *op mod* on *nat* by means of a characteristic relation with two input arguments m, n and two output arguments q(uotient) and r(emainder).

definition *divmod-nat-rel* :: $nat \Rightarrow nat \Rightarrow nat \times nat \Rightarrow bool$ **where**
divmod-nat-rel m n qr ⟷
$m = fst\ qr * n + snd\ qr \wedge$
(*if* $n = 0$ *then* $fst\ qr = 0$ *else if* $n > 0$ *then* $0 \leq snd\ qr \wedge snd\ qr < n$ *else* $n < snd\ qr \wedge snd\ qr \leq 0$)

divmod-nat-rel is total:

lemma *divmod-nat-rel-ex*:
obtains $q\ r$ **where** *divmod-nat-rel m n* (q, r)
⟨*proof*⟩

divmod-nat-rel is injective:

lemma *divmod-nat-rel-unique*:
assumes *divmod-nat-rel m n qr*
and *divmod-nat-rel m n* qr'
shows $qr = qr'$
⟨*proof*⟩

We instantiate divisibility on the natural numbers by means of *divmod-nat-rel*:

instantiation *nat* :: *semiring-div*
begin

definition *divmod-nat* :: $nat \Rightarrow nat \Rightarrow nat \times nat$ **where**
[*code del*]: *divmod-nat m n* = (*THE qr*. *divmod-nat-rel m n qr*)

lemma *divmod-nat-rel-divmod-nat*:
divmod-nat-rel m n (*divmod-nat m n*)
⟨*proof*⟩

lemma *divmod-nat-eq*:
assumes *divmod-nat-rel m n qr*
shows *divmod-nat m n = qr*
⟨*proof*⟩

definition *div-nat* **where**
m div n = fst (divmod-nat m n)

definition *mod-nat* **where**
m mod n = snd (divmod-nat m n)

lemma *divmod-nat-div-mod*:
divmod-nat m n = (m div n, m mod n)
⟨*proof*⟩

lemma *div-eq*:
assumes *divmod-nat-rel m n (q, r)*
shows *m div n = q*
⟨*proof*⟩

lemma *mod-eq*:
assumes *divmod-nat-rel m n (q, r)*
shows *m mod n = r*
⟨*proof*⟩

lemma *divmod-nat-rel*: *divmod-nat-rel m n (m div n, m mod n)*
⟨*proof*⟩

lemma *divmod-nat-zero*:
divmod-nat m 0 = (0, m)
⟨*proof*⟩

lemma *divmod-nat-base*:
assumes *m < n*
shows *divmod-nat m n = (0, m)*
⟨*proof*⟩

lemma *divmod-nat-step*:
assumes *0 < n* **and** *n ≤ m*
shows *divmod-nat m n = (Suc ((m − n) div n), (m − n) mod n)*
⟨*proof*⟩

The "recursion" equations for *op div* and *op mod*

lemma *div-less* [*simp*]:
fixes *m n* :: *nat*
assumes *m < n*
shows *m div n = 0*
⟨*proof*⟩

lemma *le-div-geq*:
fixes m n :: *nat*
assumes $0 < n$ **and** $n \leq m$
shows m *div* n = *Suc* $((m - n)$ *div* $n)$
⟨*proof*⟩

lemma *mod-less* [*simp*]:
fixes m n :: *nat*
assumes $m < n$
shows m *mod* n = m
⟨*proof*⟩

lemma *le-mod-geq*:
fixes m n :: *nat*
assumes $n \leq m$
shows m *mod* n = $(m - n)$ *mod* n
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *divmod-nat-if* [*code*]: *divmod-nat* m n = (*if* $n = 0 \vee m < n$ *then* $(0, m)$
else
let (q, r) = *divmod-nat* $(m - n)$ n *in* (*Suc* q, r))
⟨*proof*⟩

Simproc for cancelling *op div* and *op mod*

⟨*ML*⟩

33.3.1 Quotient

lemma *div-geq*: $0 < n \Longrightarrow \neg\ m < n \Longrightarrow m$ *div* n = *Suc* $((m - n)$ *div* $n)$
⟨*proof*⟩

lemma *div-if*: $0 < n \Longrightarrow m$ *div* n = (*if* $m < n$ *then* 0 *else Suc* $((m - n)$ *div* $n))$
⟨*proof*⟩

lemma *div-mult-self-is-m* [*simp*]: $0<n$ ==> $(m*n)$ *div* n = $(m$::*nat*$)$
⟨*proof*⟩

lemma *div-mult-self1-is-m* [*simp*]: $0<n$ ==> $(n*m)$ *div* n = $(m$::*nat*$)$
⟨*proof*⟩

33.3.2 Remainder

lemma *mod-less-divisor* [*simp*]:
fixes m n :: *nat*
assumes $n > 0$
shows m *mod* $n < (n$::*nat*$)$

⟨*proof*⟩

lemma *mod-less-eq-dividend* [*simp*]:
fixes *m n* :: *nat*
shows *m mod n* ≤ *m*
⟨*proof*⟩

lemma *mod-geq*: ¬ *m* < (*n*::*nat*) ⟹ *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-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 *mult-div-cancel*: (*n*::*nat*) ∗ (*m div n*) = *m* − (*m mod n*)
⟨*proof*⟩

lemma *mod-le-divisor*[*simp*]: *0* < *n* ⟹ *m mod n* ≤ (*n*::*nat*)
⟨*proof*⟩

33.3.3 Quotient and Remainder

lemma *divmod-nat-rel-mult1-eq*:
divmod-nat-rel b c (*q*, *r*) ⟹ *c* > *0*
⟹ *divmod-nat-rel* (*a* ∗ *b*) *c* (*a* ∗ *q* + *a* ∗ *r div c*, *a* ∗ *r mod c*)
⟨*proof*⟩

lemma *div-mult1-eq*:
(*a* ∗ *b*) *div c* = *a* ∗ (*b div c*) + *a* ∗ (*b mod c*) *div* (*c*::*nat*)
⟨*proof*⟩

lemma *divmod-nat-rel-add1-eq*:
divmod-nat-rel a c (*aq*, *ar*) ⟹ *divmod-nat-rel b c* (*bq*, *br*) ⟹ *c* > *0*
⟹ *divmod-nat-rel* (*a* + *b*) *c* (*aq* + *bq* + (*ar* + *br*) *div c*, (*ar* + *br*) *mod c*)
⟨*proof*⟩

lemma *div-add1-eq*:
(*a*+*b*) *div* (*c*::*nat*) = *a div c* + *b div c* + ((*a mod c* + *b mod c*) *div c*)
⟨*proof*⟩

lemma *mod-lemma*: [| (*0*::*nat*) < *c*; *r* < *b* |] ==> *b* * (*q* *mod* *c*) + *r* < *b* * *c*
⟨*proof*⟩

lemma *divmod-nat-rel-mult2-eq*:
divmod-nat-rel a b (*q*, *r*) ⟹ *0* < *b* ⟹ *0* < *c*
⟹ *divmod-nat-rel a* (*b* * *c*) (*q* *div* *c*, *b* *(*q* *mod* *c*) + *r*)
⟨*proof*⟩

lemma *div-mult2-eq*: *a* *div* (*b*∗*c*) = (*a* *div* *b*) *div* (*c*::*nat*)
⟨*proof*⟩

lemma *mod-mult2-eq*: *a* *mod* (*b*∗*c*) = *b*∗(*a* *div* *b* *mod* *c*) + *a* *mod* (*b*::*nat*)
⟨*proof*⟩

33.3.4 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m* *div* *Suc* *0* = *m*
⟨*proof*⟩

lemma *div-le-mono* [*rule-format* (*no-asm*)]:
∀ *m*::*nat*. *m* ≤ *n* −−> (*m* *div* *k*) ≤ (*n* *div* *k*)
⟨*proof*⟩

lemma *div-le-mono2*: !!*m*::*nat*. [| *0*<*m*; *m*≤*n* |] ==> (*k* *div* *n*) ≤ (*k* *div* *m*)
⟨*proof*⟩

lemma *div-le-dividend* [*simp*]: *m* *div* *n* ≤ (*m*::*nat*)
⟨*proof*⟩

lemma *div-less-dividend* [*rule-format*]:
!!*n*::*nat*. *1*<*n* ==> *0* < *m* −−> *m* *div* *n* < *m*
⟨*proof*⟩

declare *div-less-dividend* [*simp*]

A fact for the mutilated chess board

lemma *mod-Suc*: *Suc*(*m*) *mod* *n* = (*if* *Suc*(*m* *mod* *n*) = *n* *then* *0* *else* *Suc*(*m* *mod* *n*))
⟨*proof*⟩

lemma *mod-eq-0-iff*: (*m* *mod* *d* = *0*) = (∃ *q*::*nat*. *m* = *d*∗*q*)
⟨*proof*⟩

lemmas *mod-eq-0D* [*dest*!] = *mod-eq-0-iff* [*THEN* *iffD1*]

lemma *mod-eqD*: $(m\ mod\ d = r) ==> \exists\, q{::}nat.\ m = r + q{*}d$
⟨*proof*⟩

lemma *split-div*:
$P(n\ div\ k :: nat) =$
$((k = 0 \longrightarrow P\ 0) \wedge (k \neq 0 \longrightarrow (!i.\ !j{<}k.\ n = k{*}i + j \longrightarrow P\ i)))$
(**is** *?P* = *?Q* **is** - = (- ∧ (- ⟶ *?R*)))
⟨*proof*⟩

lemma *split-div-lemma*:
assumes $0 < n$
shows $n * q \leq m \wedge m < n * Suc\ q \longleftrightarrow q = ((m{::}nat)\ div\ n)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

theorem *split-div'*:
$P\ ((m{::}nat)\ div\ n) = ((n = 0 \wedge P\ 0) \vee$
$(\exists\, q.\ (n * q \leq m \wedge m < n * (Suc\ q)) \wedge P\ q))$
⟨*proof*⟩

lemma *split-mod*:
$P(n\ mod\ k :: nat) =$
$((k = 0 \longrightarrow P\ n) \wedge (k \neq 0 \longrightarrow (!i.\ !j{<}k.\ n = k{*}i + j \longrightarrow P\ j)))$
(**is** *?P* = *?Q* **is** - = (- ∧ (- ⟶ *?R*)))
⟨*proof*⟩

theorem *mod-div-equality'*: $(m{::}nat)\ mod\ n = m - (m\ div\ n) * n$
⟨*proof*⟩

lemma *div-mod-equality'*:
fixes $m\ n :: nat$
shows $m\ div\ n * n = m - m\ mod\ n$
⟨*proof*⟩

33.3.5 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*⟩

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*]: *0 < (m::nat) mod 2 ⟷ m mod 2 = 1*
⟨*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*]

lemma *Suc-times-mod-eq*: *1<k ==> Suc (k * m) mod k = 1*
⟨*proof*⟩

declare *Suc-times-mod-eq* [*of number-of w*, *standard*, *simp*]

lemma [*simp*]: *n div k ≤ (Suc n) div k*
⟨*proof*⟩

lemma *Suc-n-div-2-gt-zero* [*simp*]: *(0::nat) < n ==> 0 < (n + 1) div 2*
⟨*proof*⟩

lemma *div-2-gt-zero* [*simp*]: **assumes** *A*: *(1::nat) < n* **shows** *0 < n div 2*
⟨*proof*⟩

lemma *mod-mult-self3* [*simp*]: $(k*n + m)\ mod\ n = m\ mod\ (n::nat)$
⟨*proof*⟩

lemma *mod-mult-self4* [*simp*]: $Suc\ (k*n + m)\ mod\ n = Suc\ m\ mod\ n$
⟨*proof*⟩

lemma *mod-Suc-eq-Suc-mod*: $Suc\ m\ mod\ n = Suc\ (m\ mod\ n)\ mod\ n$
⟨*proof*⟩

33.4 Division on *int*

definition *divmod-int-rel* :: $int \Rightarrow int \Rightarrow int \times int \Rightarrow bool$ **where**
— definition of quotient and remainder
[*code*]: $divmod\text{-}int\text{-}rel\ a\ b = (\lambda(q,\ r).\ a = b * q + r \wedge$
$(if\ 0 < b\ then\ 0 \leq r \wedge r < b\ else\ b < r \wedge r \leq 0))$

definition *adjust* :: $int \Rightarrow int \times int \Rightarrow int \times int$ **where**
— for the division algorithm
[*code*]: $adjust\ b = (\lambda(q,\ r).\ if\ 0 \leq r - b\ then\ (2 * q + 1,\ r - b)$
$else\ (2 * q,\ r))$

algorithm for the case $a \geq 0,\ b > 0$

function *posDivAlg* :: $int \Rightarrow int \Rightarrow int \times int$ **where**
$posDivAlg\ a\ b = (if\ a < b \vee\ b \leq 0\ then\ (0,\ a)$
$else\ adjust\ b\ (posDivAlg\ a\ (2 * b)))$
⟨*proof*⟩
termination ⟨*proof*⟩

algorithm for the case $a < 0,\ b > 0$

function *negDivAlg* :: $int \Rightarrow int \Rightarrow int \times int$ **where**
$negDivAlg\ a\ b = (if\ 0 \leq a + b \vee b \leq 0\ \ then\ (-1,\ a + b)$
$else\ adjust\ b\ (negDivAlg\ a\ (2 * b)))$
⟨*proof*⟩
termination ⟨*proof*⟩

algorithm for the general case $b \neq (0::'a)$

definition *negateSnd* :: $int \times int \Rightarrow int \times int$ **where**
[*code-unfold*]: $negateSnd = apsnd\ uminus$

definition *divmod-int* :: $int \Rightarrow int \Rightarrow int \times int$ **where**
— The full division algorithm considers all possible signs for a, b including the special case $a=0,\ b<0$ because *negDivAlg* requires $a < (0::'a)$.
$divmod\text{-}int\ a\ b = (if\ 0 \leq a\ then\ if\ 0 \leq b\ then\ posDivAlg\ a\ b$
$else\ if\ a = 0\ then\ (0,\ 0)$
$else\ negateSnd\ (negDivAlg\ (-a)\ (-b))$
$else$
$if\ 0 < b\ then\ negDivAlg\ a\ b$

else negateSnd (posDivAlg (−a) (−b)))

instantiation *int* :: *Divides.div*
begin

definition
a div b = fst (divmod-int a b)

definition
a mod b = snd (divmod-int a b)

instance ⟨*proof*⟩

end

lemma *divmod-int-mod-div*:
divmod-int p q = (p div q, p mod q)
⟨*proof*⟩

Here is the division algorithm in ML:

```
fun posDivAlg (a,b) =
  if a<b then (0,a)
  else let val (q,r) = posDivAlg(a, 2*b)
           in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
       end

fun negDivAlg (a,b) =
  if 0\<le>a+b then (~1,a+b)
  else let val (q,r) = negDivAlg(a, 2*b)
           in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
       end;

fun negateSnd (q,r:int) = (q,~r);

fun divmod (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));
```

33.4.1 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*:
[| *divmod-int-rel a b* (q, r); *divmod-int-rel a b* (q', r'); $b \neq 0$ |]
==> $q = q'$
⟨*proof*⟩

lemma *unique-remainder*:
[| *divmod-int-rel a b* (q, r); *divmod-int-rel a b* (q', r'); $b \neq 0$ |]
==> $r = r'$
⟨*proof*⟩

33.4.2 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 $0 \leq$ *diff then* $(2*q + 1,$ *diff*$)$
else $(2*q, r)$)
⟨*proof*⟩

declare *posDivAlg.simps* [*simp del*]

use with a simproc to avoid repeatedly proving the premise

lemma *posDivAlg-eqn*:
$0 < b$ ==>
posDivAlg a b = (*if* $a<b$ *then* $(0,a)$ *else adjust b* (*posDivAlg a* $(2*b)$))
⟨*proof*⟩

Correctness of *posDivAlg*: it computes quotients correctly

theorem *posDivAlg-correct*:
assumes $0 \leq a$ **and** $0 < b$
shows *divmod-int-rel a b* (*posDivAlg a b*)
⟨*proof*⟩

33.4.3 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*:
$0 < b \Longrightarrow$
$\mathit{negDivAlg}\ a\ b =$
$(\mathit{if}\ 0 \leq a+b\ \mathit{then}\ (-1, a+b)\ \mathit{else}\ \mathit{adjust}\ b\ (\mathit{negDivAlg}\ a\ (2*b)))$
⟨*proof*⟩

lemma *negDivAlg-correct*:
assumes $a < 0$ **and** $b > 0$
shows *divmod-int-rel a b* (*negDivAlg a b*)
⟨*proof*⟩

33.4.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *divmod-int-rel-0*: $b \neq 0 \Longrightarrow \mathit{divmod\text{-}int\text{-}rel}\ 0\ b\ (0, 0)$
⟨*proof*⟩

lemma *posDivAlg-0* [*simp*]: $\mathit{posDivAlg}\ 0\ b = (0, 0)$
⟨*proof*⟩

lemma *negDivAlg-minus1* [*simp*]: $\mathit{negDivAlg}\ -1\ b = (-1, b - 1)$
⟨*proof*⟩

lemma *negateSnd-eq* [*simp*]: $\mathit{negateSnd}(q,r) = (q,-r)$
⟨*proof*⟩

lemma *divmod-int-rel-neg*: $\mathit{divmod\text{-}int\text{-}rel}\ (-a)\ (-b)\ qr \Longrightarrow \mathit{divmod\text{-}int\text{-}rel}\ a\ b\ (\mathit{negateSnd}\ qr)$
⟨*proof*⟩

lemma *divmod-int-correct*: $b \neq 0 \Longrightarrow \mathit{divmod\text{-}int\text{-}rel}\ a\ b\ (\mathit{divmod\text{-}int}\ a\ b)$
⟨*proof*⟩

Arbitrary definitions for division by zero. Useful to simplify certain equations.

lemma *DIVISION-BY-ZERO* [*simp*]: $a\ \mathit{div}\ (0::\mathit{int}) = 0\ \&\ a\ \mathit{mod}\ (0::\mathit{int}) = a$
⟨*proof*⟩

Basic laws about division and remainder

lemma *zmod-zdiv-equality*: $(a::\mathit{int}) = b * (a\ \mathit{div}\ b) + (a\ \mathit{mod}\ b)$

⟨*proof*⟩

lemma *zdiv-zmod-equality*: $(b * (a\ div\ b) + (a\ mod\ b)) + k = (a::int)+k$
⟨*proof*⟩

lemma *zdiv-zmod-equality2*: $((a\ div\ b) * b + (a\ mod\ b)) + k = (a::int)+k$
⟨*proof*⟩

Tool setup

⟨*ML*⟩

lemma *pos-mod-conj* : $(0::int) < b \Longrightarrow 0 \leq a\ mod\ b\ \&\ a\ mod\ b < b$
⟨*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* : $b < (0::int) \Longrightarrow a\ mod\ b \leq 0\ \&\ b < a\ mod\ b$
⟨*proof*⟩

lemmas *neg-mod-sign* [*simp*] = *neg-mod-conj* [*THEN conjunct1*, *standard*]
and *neg-mod-bound* [*simp*] = *neg-mod-conj* [*THEN conjunct2*, *standard*]

33.4.5 General Properties of div and mod

lemma *divmod-int-rel-div-mod*: $b \neq 0 \Longrightarrow divmod\text{-}int\text{-}rel\ a\ b\ (a\ div\ b,\ a\ mod\ b)$
⟨*proof*⟩

lemma *divmod-int-rel-div*: $[|\ divmod\text{-}int\text{-}rel\ a\ b\ (q,\ r);\ \ b \neq 0\ |] \Longrightarrow a\ div\ b = q$
⟨*proof*⟩

lemma *divmod-int-rel-mod*: $[|\ divmod\text{-}int\text{-}rel\ a\ b\ (q,\ r);\ \ b \neq 0\ |] \Longrightarrow a\ mod\ b = r$
⟨*proof*⟩

lemma *div-pos-pos-trivial*: $[|\ (0::int) \leq a;\ \ a < b\ |] \Longrightarrow a\ div\ b = 0$
⟨*proof*⟩

lemma *div-neg-neg-trivial*: $[|\ a \leq (0::int);\ \ b < a\ |] \Longrightarrow a\ div\ b = 0$
⟨*proof*⟩

lemma *div-pos-neg-trivial*: $[|\ (0::int) < a;\ \ a+b \leq 0\ |] \Longrightarrow a\ div\ b = -1$
⟨*proof*⟩

lemma *mod-pos-pos-trivial*: $[|\ (0::int) \leq a;\ \ a < b\ |] \Longrightarrow a\ mod\ b = a$
⟨*proof*⟩

lemma *mod-neg-neg-trivial*: [| $a \leq (0::int)$; $b < a$ |] ==> $a \bmod b = a$
⟨*proof*⟩

lemma *mod-pos-neg-trivial*: [| $(0::int) < a$; $a+b \leq 0$ |] ==> $a \bmod b = a+b$
⟨*proof*⟩

There is no *mod-neg-pos-trivial*.

lemma *zdiv-zminus-zminus* [*simp*]: $(-a) \operatorname{div} (-b) = a \operatorname{div} (b::int)$
⟨*proof*⟩

lemma *zmod-zminus-zminus* [*simp*]: $(-a) \bmod (-b) = -(a \bmod (b::int))$
⟨*proof*⟩

33.4.6 Laws for div and mod with Unary Minus

lemma *zminus1-lemma*:
$divmod\text{-}int\text{-}rel\ a\ b\ (q, r)$
$\Longrightarrow divmod\text{-}int\text{-}rel\ (-a)\ b\ (if\ r=0\ then\ -q\ else\ -q - 1,$
$if\ r=0\ then\ 0\ else\ b-r)$
⟨*proof*⟩

lemma *zdiv-zminus1-eq-if*:
$b \neq (0::int)$
$\Longrightarrow (-a) \operatorname{div} b =$
$(if\ a \bmod b = 0\ then\ -(a \operatorname{div} b)\ else\ -(a \operatorname{div} b) - 1)$
⟨*proof*⟩

lemma *zmod-zminus1-eq-if*:
$(-a::int) \bmod b = (if\ a \bmod b = 0\ then\ 0\ else\ b - (a \bmod b))$
⟨*proof*⟩

lemma *zmod-zminus1-not-zero*:
fixes $k\ l :: int$
shows $-k \bmod l \neq 0 \Longrightarrow k \bmod l \neq 0$
⟨*proof*⟩

lemma *zdiv-zminus2*: $a \operatorname{div} (-b) = (-a::int) \operatorname{div} b$
⟨*proof*⟩

lemma *zmod-zminus2*: $a \bmod (-b) = -((-a::int) \bmod b)$
⟨*proof*⟩

lemma *zdiv-zminus2-eq-if*:
$b \neq (0::int)$
$\Longrightarrow a \operatorname{div} (-b) =$
$(if\ a \bmod b = 0\ then\ -(a \operatorname{div} b)\ else\ -(a \operatorname{div} b) - 1)$
⟨*proof*⟩

lemma *zmod-zminus2-eq-if*:
$a\ mod\ (-b::int) = (if\ a\ mod\ b = 0\ then\ 0\ else\ (a\ mod\ b) - b)$
⟨*proof*⟩

lemma *zmod-zminus2-not-zero*:
fixes $k\ l :: int$
shows $k\ mod\ -l \neq 0 \Longrightarrow k\ mod\ l \neq 0$
⟨*proof*⟩

33.4.7 Division of a Number by Itself

lemma *self-quotient-aux1*: $[|\ (0::int) < a;\ a = r + a*q;\ r < a\ |] ==> 1 \leq q$
⟨*proof*⟩

lemma *self-quotient-aux2*: $[|\ (0::int) < a;\ a = r + a*q;\ 0 \leq r\ |] ==> q \leq 1$
⟨*proof*⟩

lemma *self-quotient*: $[|\ divmod\text{-}int\text{-}rel\ a\ a\ (q, r);\ \ a \neq (0::int)\ |] ==> q = 1$
⟨*proof*⟩

lemma *self-remainder*: $[|\ divmod\text{-}int\text{-}rel\ a\ a\ (q, r);\ \ a \neq (0::int)\ |] ==> r = 0$
⟨*proof*⟩

lemma *zdiv-self* [*simp*]: $a \neq 0 ==> a\ div\ a = (1::int)$
⟨*proof*⟩

lemma *zmod-self* [*simp*]: $a\ mod\ a = (0::int)$
⟨*proof*⟩

33.4.8 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 *zmod-minus1*: $(0::int) < b ==> -1\ mod\ b = b - 1$
⟨*proof*⟩

a positive, b positive

lemma *div-pos-pos*: $[|\ 0 < a;\ \ 0 \leq b\ |] ==> a\ div\ b = fst\ (posDivAlg\ a\ b)$
⟨*proof*⟩

lemma *mod-pos-pos*: $[|\ 0 < a;\ \ 0 \leq b\ |] ==> a\ mod\ b = snd\ (posDivAlg\ a\ b)$

⟨*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*: [| $a < 0$; $0 < b$ |] ==> a *mod* b = *snd* (*negDivAlg* a b)
⟨*proof*⟩

a positive, b negative

lemma *div-pos-neg*:
[| $0 < a$; $b < 0$ |] ==> a *div* b = *fst* (*negateSnd* (*negDivAlg* $(-a)$ $(-b)$))
⟨*proof*⟩

lemma *mod-pos-neg*:
[| $0 < a$; $b < 0$ |] ==> a *mod* b = *snd* (*negateSnd* (*negDivAlg* $(-a)$ $(-b)$))
⟨*proof*⟩

a negative, b negative

lemma *div-neg-neg*:
[| $a < 0$; $b \leq 0$ |] ==> a *div* b = *fst* (*negateSnd* (*posDivAlg* $(-a)$ $(-b)$))
⟨*proof*⟩

lemma *mod-neg-neg*:
[| $a < 0$; $b \leq 0$ |] ==> a *mod* b = *snd* (*negateSnd* (*posDivAlg* $(-a)$ $(-b)$))
⟨*proof*⟩

Simplify expresions in which div and mod combine numerical constants

lemma *divmod-int-relI*:
⟦$a == b * q + r$; *if* $0 < b$ *then* $0 \leq r \wedge r < b$ *else* $b < r \wedge r \leq 0$⟧
$\Longrightarrow$ *divmod-int-rel* a b (q, r)
⟨*proof*⟩

lemmas *divmod-int-rel-div-eq* = *divmod-int-relI* [*THEN divmod-int-rel-div*, *THEN eq-reflection*]
lemmas *divmod-int-rel-mod-eq* = *divmod-int-relI* [*THEN divmod-int-rel-mod*, *THEN eq-reflection*]
lemmas *arithmetic-simps* =
arith-simps
add-special
add-0-left
add-0-right
mult-zero-left
mult-zero-right
mult-1-left
mult-1-right

⟨*ML*⟩

lemmas *posDivAlg-eqn-number-of* [*simp*] =
posDivAlg-eqn [*of number-of v number-of w*, *standard*]

lemmas *negDivAlg-eqn-number-of* [*simp*] =
negDivAlg-eqn [*of number-of v number-of w*, *standard*]

Special-case simplification

lemma *zmod-minus1-right* [*simp*]: *a mod* (−*1*::*int*) = *0*
⟨*proof*⟩

lemma *zdiv-minus1-right* [*simp*]: *a div* (−*1*::*int*) = −*a*
⟨*proof*⟩

lemmas *div-pos-pos-1-number-of* [*simp*] =
div-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *div-pos-neg-1-number-of* [*simp*] =
div-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *mod-pos-pos-1-number-of* [*simp*] =
mod-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *mod-pos-neg-1-number-of* [*simp*] =
mod-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *posDivAlg-eqn-1-number-of* [*simp*] =
posDivAlg-eqn [*of concl*: *1 number-of w*, *standard*]

lemmas *negDivAlg-eqn-1-number-of* [*simp*] =
negDivAlg-eqn [*of concl*: *1 number-of w*, *standard*]

33.4.9 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*: [| *a* ≤ *a'*; *0* < (*b*::*int*) |] ==> *a div b* ≤ *a' div b*
⟨*proof*⟩

lemma *zdiv-mono1-neg*: [| *a* ≤ *a'*; (*b*::*int*) < *0* |] ==> *a' div b* ≤ *a div b*
⟨*proof*⟩

33.4.10 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
[| *0* ≤ *b'***q'* + *r'*; *r'* < *b'*; *0* < *b'* |] ==> *0* ≤ (*q'*::*int*)
⟨*proof*⟩

lemma *zdiv-mono2-lemma*:
[| $b*q + r = b'*q' + r'$; $0 \leq b'*q' + r'$;
$r' < b'$; $0 \leq r$; $0 < b'$; $b' \leq b$ |]
==> $q \leq (q'::int)$
⟨*proof*⟩

lemma *zdiv-mono2*:
[| $(0::int) \leq a$; $0 < b'$; $b' \leq b$ |] ==> $a \ div \ b \leq a \ div \ b'$
⟨*proof*⟩

lemma *q-neg-lemma*:
[| $b'*q' + r' < 0$; $0 \leq r'$; $0 < b'$ |] ==> $q' \leq (0::int)$
⟨*proof*⟩

lemma *zdiv-mono2-neg-lemma*:
[| $b*q + r = b'*q' + r'$; $b'*q' + r' < 0$;
$r < b$; $0 \leq r'$; $0 < b'$; $b' \leq b$ |]
==> $q' \leq (q::int)$
⟨*proof*⟩

lemma *zdiv-mono2-neg*:
[| $a < (0::int)$; $0 < b'$; $b' \leq b$ |] ==> $a \ div \ b' \leq a \ div \ b$
⟨*proof*⟩

33.4.11 More Algebraic Laws for div and mod

proving (a*b) div c = a * (b div c) + a * (b mod c)

lemma *zmult1-lemma*:
[| *divmod-int-rel* $b \ c \ (q, \ r)$; $c \neq 0$ |]
==> *divmod-int-rel* $(a * b) \ c \ (a*q + a*r \ div \ c, \ a*r \ mod \ c)$
⟨*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-zdiv-trivial*: $(a \ mod \ b) \ div \ b = (0::int)$
⟨*proof*⟩

proving (a+b) div c = a div c + b div c + ((a mod c + b mod c) div c)

lemma *zadd1-lemma*:
[| *divmod-int-rel* $a \ c \ (aq, \ ar)$; *divmod-int-rel* $b \ c \ (bq, \ br)$; $c \neq 0$ |]
==> *divmod-int-rel* $(a+b) \ c \ (aq + bq + (ar+br) \ div \ c, \ (ar+br) \ mod \ c)$
⟨*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*⟩

instance *int* :: *ring-div*
⟨*proof*⟩

lemma *posDivAlg-div-mod*:
assumes $k \geq 0$
and $l \geq 0$
shows *posDivAlg k l* = $(k$ *div* l, k *mod* $l)$
⟨*proof*⟩

lemma *negDivAlg-div-mod*:
assumes $k < 0$
and $l > 0$
shows *negDivAlg k l* = $(k$ *div* l, k *mod* $l)$
⟨*proof*⟩

lemma *zmod-eq-0-iff*: $(m$ *mod* $d = 0) = (EX\ q::int.\ m = d*q)$
⟨*proof*⟩

lemmas *zmod-eq-0D* [*dest*!] = *zmod-eq-0-iff* [*THEN iffD1*]

33.4.12 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*: [| *divmod-int-rel a b* (q, r); $b \neq 0$; $0 < c$ |]
==> *divmod-int-rel 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*⟩

33.4.13 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*) = (∀ *i j*. 0≤*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*) = (∀ *i j*. *k*<*j* & *j*≤0 & *n* = *k*∗*i* + *j* −−> *P i j*)
⟨*proof*⟩

lemma *split-zdiv*:
P(*n div k* :: *int*) =
((*k* = 0 −−> *P* 0) &
(0<*k* −−> (∀ *i j*. 0≤*j* & *j*<*k* & *n* = *k*∗*i* + *j* −−> *P i*)) &
(*k*<0 −−> (∀ *i j*. *k*<*j* & *j*≤0 & *n* = *k*∗*i* + *j* −−> *P i*)))
⟨*proof*⟩

lemma *split-zmod*:
P(*n mod k* :: *int*) =
((*k* = 0 −−> *P n*) &
(0<*k* −−> (∀ *i j*. 0≤*j* & *j*<*k* & *n* = *k*∗*i* + *j* −−> *P j*)) &
(*k*<0 −−> (∀ *i j*. *k*<*j* & *j*≤0 & *n* = *k*∗*i* + *j* −−> *P j*)))
⟨*proof*⟩

Enable (lin)arith to deal with *op div* and *op mod* when these are applied to some constant that is of the form *number-of k*:

declare *split-zdiv* [*of* - - *number-of k*, *standard*, *arith-split*]
declare *split-zmod* [*of* - - *number-of k*, *standard*, *arith-split*]

33.4.14 Speeding up the Division Algorithm with Shifting

computing div by shifting

lemma *pos-zdiv-mult-2*: (0::*int*) ≤ *a* ==> (1 + 2∗*b*) *div* (2∗*a*) = *b div a*
⟨*proof*⟩

lemma *neg-zdiv-mult-2*:
assumes *A*: *a* ≤ (0::*int*) **shows** (1 + 2∗*b*) *div* (2∗*a*) = (*b*+1) *div a*
⟨*proof*⟩

lemma *zdiv-number-of-Bit0* [*simp*]:

number-of (Int.Bit0 v) div number-of (Int.Bit0 w) =
number-of v div (number-of w :: int)
⟨*proof*⟩

lemma *zdiv-number-of-Bit1* [*simp*]:
number-of (Int.Bit1 v) div number-of (Int.Bit0 w) =
(if (0::int) ≤ number-of w
then number-of v div (number-of w)
else (number-of v + (1::int)) div (number-of w))
⟨*proof*⟩

33.4.15 Computing mod by Shifting (proofs resemble those for div)

lemma *pos-zmod-mult-2*:
fixes *a b* :: *int*
assumes *0 ≤ a*
shows *(1 + 2 * b) mod (2 * a) = 1 + 2 * (b mod a)*
⟨*proof*⟩

lemma *neg-zmod-mult-2*:
fixes *a b* :: *int*
assumes *a ≤ 0*
shows *(1 + 2 * b) mod (2 * a) = 2 * ((b + 1) mod a) - 1*
⟨*proof*⟩

lemma *zmod-number-of-Bit0* [*simp*]:
number-of (Int.Bit0 v) mod number-of (Int.Bit0 w) =
*(2::int) * (number-of v mod number-of w)*
⟨*proof*⟩

lemma *zmod-number-of-Bit1* [*simp*]:
number-of (Int.Bit1 v) mod number-of (Int.Bit0 w) =
(if (0::int) ≤ number-of w
*then 2 * (number-of v mod number-of w) + 1*
*else 2 * ((number-of v + (1::int)) mod number-of w) - 1)*
⟨*proof*⟩

33.4.16 Quotients of Signs

lemma *div-neg-pos-less0*: [| *a < (0::int); 0 < b* |] ==> *a div b < 0*
⟨*proof*⟩

lemma *div-nonneg-neg-le0*: [| *(0::int) ≤ a; b < 0* |] ==> *a div b ≤ 0*
⟨*proof*⟩

lemma *div-nonpos-pos-le0*: [| *(a::int) ≤ 0; b > 0* |] ==> *a div b ≤ 0*
⟨*proof*⟩

Now for some equivalences of the form *a div b >=< 0* ⟷ ... conditional

upon the sign of a or b. There are many more. They should all be simp rules unless that causes too much search.

lemma *pos-imp-zdiv-nonneg-iff*: $(0::int) < b \Longrightarrow (0 \leq a \ div \ b) = (0 \leq a)$
⟨*proof*⟩

lemma *neg-imp-zdiv-nonneg-iff*:
$b < (0::int) \Longrightarrow (0 \leq a \ div \ b) = (a \leq (0::int))$
⟨*proof*⟩

lemma *pos-imp-zdiv-neg-iff*: $(0::int) < b \Longrightarrow (a \ div \ b < 0) = (a < 0)$
⟨*proof*⟩

lemma *neg-imp-zdiv-neg-iff*: $b < (0::int) \Longrightarrow (a \ div \ b < 0) = (0 < a)$
⟨*proof*⟩

lemma *nonneg1-imp-zdiv-pos-iff*:
$(0::int) <= a \Longrightarrow (a \ div \ b > 0) = (a >= b \ \& \ b>0)$
⟨*proof*⟩

33.4.17 The Divides Relation

lemmas *zdvd-iff-zmod-eq-0-number-of* [*simp*] =
dvd-eq-mod-eq-0 [*of number-of x*::*int number-of y*::*int*, *standard*]

lemma *zdvd-zmod*: $f \ dvd \ m \Longrightarrow f \ dvd \ (n::int) \Longrightarrow f \ dvd \ m \ mod \ n$
⟨*proof*⟩

lemma *zdvd-zmod-imp-zdvd*: $k \ dvd \ m \ mod \ n \Longrightarrow k \ dvd \ n \Longrightarrow k \ dvd \ (m::int)$
⟨*proof*⟩

lemma *zmult-div-cancel*: $(n::int) * (m \ div \ n) = m - (m \ mod \ n)$
⟨*proof*⟩

lemma *zpower-zmod*: $((x::int) \ mod \ m) \hat{} y \ mod \ m = x \hat{} y \ mod \ m$
⟨*proof*⟩

lemma *zdiv-int*: $int \ (a \ div \ b) = (int \ a) \ div \ (int \ b)$
⟨*proof*⟩

lemma *zmod-int*: $int \ (a \ mod \ b) = (int \ a) \ mod \ (int \ b)$
⟨*proof*⟩

lemma *abs-div*: $(y::int) \ dvd \ x \Longrightarrow abs \ (x \ div \ y) = abs \ x \ div \ abs \ y$
⟨*proof*⟩

lemma *zdvd-mult-div-cancel*:$(n::int) \ dvd \ m \Longrightarrow n * (m \ div \ n) = m$
⟨*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*⟩

by Brian Huffman

lemma *zminus-zmod*: $-\ ((x::int)\ mod\ m)\ mod\ m = -\ x\ mod\ m$
⟨*proof*⟩

lemma *zdiff-zmod-left*: $(x\ mod\ m - y)\ mod\ m = (x - y)\ mod\ (m::int)$
⟨*proof*⟩

lemma *zdiff-zmod-right*: $(x - y\ mod\ m)\ mod\ m = (x - y)\ mod\ (m::int)$
⟨*proof*⟩

lemmas *zmod-simps* =
mod-add-left-eq [*symmetric*]
mod-add-right-eq [*symmetric*]
zmod-zmult1-eq [*symmetric*]
mod-mult-left-eq [*symmetric*]
zpower-zmod
zminus-zmod zdiff-zmod-left zdiff-zmod-right

Distributive laws for function *nat*.

lemma *nat-div-distrib*: $0 \le x \Longrightarrow nat\ (x\ div\ y) = nat\ x\ div\ nat\ y$
⟨*proof*⟩

lemma *nat-mod-distrib*:
$[\![0 \le x;\ 0 \le y]\!] \Longrightarrow nat\ (x\ mod\ y) = nat\ x\ mod\ nat\ y$
⟨*proof*⟩

transfer setup

lemma *transfer-nat-int-functions*:
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow (nat\ x)\ div\ (nat\ y) = nat\ (x\ div\ y)$
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow (nat\ x)\ mod\ (nat\ y) = nat\ (x\ mod\ y)$
⟨*proof*⟩

lemma *transfer-nat-int-function-closures*:
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow x\ div\ y >= 0$
$(x::int) >= 0 \Longrightarrow y >= 0 \Longrightarrow x\ mod\ y >= 0$
⟨*proof*⟩

declare *transfer-morphism-nat-int* [*transfer add return*:
transfer-nat-int-functions
transfer-nat-int-function-closures
]

lemma *transfer-int-nat-functions*:
  $(int\ x)\ div\ (int\ y) = int\ (x\ div\ y)$
  $(int\ x)\ mod\ (int\ y) = int\ (x\ mod\ y)$
⟨*proof*⟩

lemma *transfer-int-nat-function-closures*:
  $is\text{-}nat\ x \Longrightarrow is\text{-}nat\ y \Longrightarrow is\text{-}nat\ (x\ div\ y)$
  $is\text{-}nat\ x \Longrightarrow is\text{-}nat\ y \Longrightarrow is\text{-}nat\ (x\ mod\ y)$
⟨*proof*⟩

declare *transfer-morphism-int-nat* [*transfer add return*:
  transfer-int-nat-functions
  transfer-int-nat-function-closures
]

Suggested by Matthias Daum

lemma *int-div-less-self*: $[\![0 < x;\ 1 < k]\!] \Longrightarrow x\ div\ k < (x::int)$
⟨*proof*⟩

lemma *zmod-eq-dvd-iff*: $(x::int)\ mod\ n = y\ mod\ n \longleftrightarrow n\ dvd\ x - y$
⟨*proof*⟩

lemma *nat-mod-eq-lemma*: **assumes** *xyn*: $(x::nat)\ mod\ n = y\ \ mod\ n$ **and** *xy*:$y \leq x$
  shows $\exists\, q.\ x = y + n * q$
⟨*proof*⟩

lemma *nat-mod-eq-iff*: $(x::nat)\ mod\ n = y\ mod\ n \longleftrightarrow (\exists\, q1\ q2.\ x + n * q1 = y + n * q2)$
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *div-nat-number-of* [*simp*]:
    $(number\text{-}of\ v :: nat)\ \ div\ \ number\text{-}of\ v' =$
        $(if\ neg\ (number\text{-}of\ v :: int)\ then\ 0$
         $else\ nat\ (number\text{-}of\ v\ div\ number\text{-}of\ v'))$
⟨*proof*⟩

lemma *one-div-nat-number-of* [*simp*]:
    $Suc\ 0\ div\ number\text{-}of\ v' = nat\ (1\ div\ number\text{-}of\ v')$
⟨*proof*⟩

lemma *mod-nat-number-of* [*simp*]:
    $(number\text{-}of\ v :: nat)\ \ mod\ \ number\text{-}of\ v' =$
      $(if\ neg\ (number\text{-}of\ v :: int)\ then\ 0$
       $else\ if\ neg\ (number\text{-}of\ v' :: int)\ then\ number\text{-}of\ v$
       $else\ nat\ (number\text{-}of\ v\ mod\ number\text{-}of\ v'))$
⟨*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*⟩

lemmas *dvd-eq-mod-eq-0-number-of =*
dvd-eq-mod-eq-0 [*of number-of x number-of y, standard*]

declare *dvd-eq-mod-eq-0-number-of* [*simp*]

33.4.18 Nitpick

lemma *zmod-zdiv-equality′*:
(*m*::*int*) *mod n = m − (m div n) ∗ n*
⟨*proof*⟩

lemmas [*nitpick-def*] = *dvd-eq-mod-eq-0* [*THEN eq-reflection*]
mod-div-equality′ [*THEN eq-reflection*]
zmod-zdiv-equality′ [*THEN eq-reflection*]

33.4.19 Code generation

definition *pdivmod* :: *int ⇒ int ⇒ int × int* **where**
pdivmod k l = (|k| div |l|, |k| mod |l|)

lemma *pdivmod-posDivAlg* [*code*]:
pdivmod k l = (if l = 0 then (0, |k|) else posDivAlg |k| |l|)
⟨*proof*⟩

lemma *divmod-int-pdivmod*: *divmod-int k l = (if k = 0 then (0, 0) else if l = 0 then (0, k) else*
apsnd ((op ∗) (sgn l)) (if 0 < l ∧ 0 ≤ k ∨ l < 0 ∧ k < 0
then pdivmod k l
else (let (r, s) = pdivmod k l in
if s = 0 then (− r, 0) else (− r − 1, |l| − s))))
⟨*proof*⟩

lemma *divmod-int-code* [*code*]: *divmod-int k l = (if k = 0 then (0, 0) else if l = 0 then (0, k) else*
apsnd ((op ∗) (sgn l)) (if sgn k = sgn l
then pdivmod k l
else (let (r, s) = pdivmod k l in
if s = 0 then (− r, 0) else (− r − 1, |l| − s))))
⟨*proof*⟩

context *ring-1*
begin

lemma *of-int-num* [*code*]:

of-int k = (if k = 0 then 0 else if k < 0 then
− of-int (− k) else let
(l, m) = divmod-int k 2;
l′ = of-int l
in if m = 0 then l′ + l′ else l′ + l′ + 1)
⟨*proof*⟩

end

code-modulename *SML*
Divides Arith

code-modulename *OCaml*
Divides Arith

code-modulename *Haskell*
Divides Arith

end

34 Code-Numeral: Type of target language numerals

theory *Code-Numeral*
imports *Nat-Numeral Nat-Transfer Divides*
begin

Code numerals are isomorphic to HOL *nat* but mapped to target-language builtin numerals.

34.1 Datatype of target language numerals

typedef (**open**) *code-numeral = UNIV :: nat set*
morphisms *nat-of of-nat* ⟨*proof*⟩

lemma *of-nat-nat-of* [*simp*]:
of-nat (nat-of k) = k
⟨*proof*⟩

lemma *nat-of-of-nat* [*simp*]:
nat-of (of-nat n) = n
⟨*proof*⟩

lemma [*measure-function*]:
is-measure nat-of ⟨*proof*⟩

lemma *code-numeral*:

$(\bigwedge n{::}$*code-numeral*$.$ *PROP P n*$) \equiv (\bigwedge n{::}$*nat*$.$ *PROP P* (*of-nat n*))
⟨*proof*⟩

lemma *code-numeral-case*:
assumes $\bigwedge n.$ *k* = *of-nat n* $\Longrightarrow$ *P*
shows *P*
⟨*proof*⟩

lemma *code-numeral-induct-raw*:
assumes $\bigwedge n.$ *P* (*of-nat n*)
shows *P k*
⟨*proof*⟩

lemma *nat-of-inject* [*simp*]:
nat-of k = *nat-of l* $\longleftrightarrow$ *k* = *l*
⟨*proof*⟩

lemma *of-nat-inject* [*simp*]:
of-nat n = *of-nat m* $\longleftrightarrow$ *n* = *m*
⟨*proof*⟩

instantiation *code-numeral* :: *zero*
begin

definition [*simp*, *code del*]:
0 = *of-nat 0*

instance ⟨*proof*⟩

end

definition [*simp*]:
Suc-code-numeral k = *of-nat* (*Suc* (*nat-of k*))

rep-datatype *0* :: *code-numeral Suc-code-numeral*
⟨*proof*⟩

declare *code-numeral-case* [*case-names nat*, *cases type*: *code-numeral*]
declare *code-numeral.induct* [*case-names nat*, *induct type*: *code-numeral*]

lemma *code-numeral-decr* [*termination-simp*]:
k ≠ *of-nat 0* $\Longrightarrow$ *nat-of k* − *Suc 0* < *nat-of k*
⟨*proof*⟩

lemma [*simp*, *code*]:
code-numeral-size = *nat-of*
⟨*proof*⟩

lemma [*simp*, *code*]:

size = *nat-of*
⟨*proof*⟩

lemmas [*code del*] = *code-numeral.recs code-numeral.cases*

lemma [*code*]:
eq-class.eq k l ⟷ *eq-class.eq* (*nat-of k*) (*nat-of l*)
⟨*proof*⟩

lemma [*code nbe*]:
eq-class.eq (*k*::*code-numeral*) *k* ⟷ *True*
⟨*proof*⟩

34.2 Indices as datatype of ints

instantiation *code-numeral* :: *number*
begin

definition
number-of = *of-nat o nat*

instance ⟨*proof*⟩

end

lemma *nat-of-number* [*simp*]:
nat-of (*number-of k*) = *number-of k*
⟨*proof*⟩

code-datatype *number-of* :: *int* ⇒ *code-numeral*

34.3 Basic arithmetic

instantiation *code-numeral* :: {*minus*, *linordered-semidom*, *semiring-div*, *linorder*}
begin

definition [*simp*, *code del*]:
(*1*::*code-numeral*) = *of-nat 1*

definition [*simp*, *code del*]:
n + *m* = *of-nat* (*nat-of n* + *nat-of m*)

definition [*simp*, *code del*]:
n − *m* = *of-nat* (*nat-of n* − *nat-of m*)

definition [*simp*, *code del*]:
n ∗ *m* = *of-nat* (*nat-of n* ∗ *nat-of m*)

definition [*simp*, *code del*]:
n div m = *of-nat* (*nat-of n div nat-of m*)

definition [*simp*, *code del*]:
n mod m = of-nat (nat-of n mod nat-of m)

definition [*simp*, *code del*]:
$n \leq m \longleftrightarrow$ *nat-of n* $\leq$ *nat-of m*

definition [*simp*, *code del*]:
$n < m \longleftrightarrow$ *nat-of n* $<$ *nat-of m*

instance ⟨*proof*⟩

end

lemma *zero-code-numeral-code* [*code*, *code-unfold*]:
(*0*::*code-numeral*) = *Numeral0*
⟨*proof*⟩
lemma [*code-post*]: *Numeral0* = (*0*::*code-numeral*)
⟨*proof*⟩

lemma *one-code-numeral-code* [*code*, *code-unfold*]:
(*1*::*code-numeral*) = *Numeral1*
⟨*proof*⟩
lemma [*code-post*]: *Numeral1* = (*1*::*code-numeral*)
⟨*proof*⟩

lemma *plus-code-numeral-code* [*code nbe*]:
of-nat n + *of-nat m* = *of-nat* (*n* + *m*)
⟨*proof*⟩

definition *subtract-code-numeral* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
where
[*simp*, *code del*]: *subtract-code-numeral* = *op* −

lemma *subtract-code-numeral-code* [*code nbe*]:
subtract-code-numeral (*of-nat n*) (*of-nat m*) = *of-nat* (*n* − *m*)
⟨*proof*⟩

lemma *minus-code-numeral-code* [*code*]:
n − *m* = *subtract-code-numeral n m*
⟨*proof*⟩

lemma *times-code-numeral-code* [*code nbe*]:
of-nat n ∗ *of-nat m* = *of-nat* (*n* ∗ *m*)
⟨*proof*⟩

lemma *less-eq-code-numeral-code* [*code nbe*]:
of-nat n $\leq$ *of-nat m* $\longleftrightarrow$ *n* $\leq$ *m*
⟨*proof*⟩

lemma *less-code-numeral-code* [*code nbe*]:
of-nat n < of-nat m ⟷ n < m
⟨*proof*⟩

lemma *code-numeral-zero-minus-one*:
(0::code-numeral) − 1 = 0
⟨*proof*⟩

lemma *Suc-code-numeral-minus-one*:
Suc-code-numeral n − 1 = n
⟨*proof*⟩

lemma *of-nat-code* [*code*]:
of-nat = Nat.of-nat
⟨*proof*⟩

lemma *code-numeral-not-eq-zero*: *i ≠ of-nat 0 ⟷ i ≥ 1*
⟨*proof*⟩

definition *nat-of-aux* :: *code-numeral ⇒ nat ⇒ nat* **where**
nat-of-aux i n = nat-of i + n

lemma *nat-of-aux-code* [*code*]:
nat-of-aux i n = (if i = 0 then n else nat-of-aux (i − 1) (Suc n))
⟨*proof*⟩

lemma *nat-of-code* [*code*]:
nat-of i = nat-of-aux i 0
⟨*proof*⟩

definition *div-mod-code-numeral* :: *code-numeral ⇒ code-numeral ⇒ code-numeral × code-numeral* **where**
[*code del*]: *div-mod-code-numeral n m = (n div m, n mod m)*

lemma [*code*]:
div-mod-code-numeral n m = (if m = 0 then (0, n) else (n div m, n mod m))
⟨*proof*⟩

lemma [*code*]:
n div m = fst (div-mod-code-numeral n m)
⟨*proof*⟩

lemma [*code*]:
n mod m = snd (div-mod-code-numeral n m)
⟨*proof*⟩

definition *int-of* :: *code-numeral ⇒ int* **where**
int-of = Nat.of-nat o nat-of

lemma *int-of-code* [*code*]:
int-of k = (*if k* = *0 then 0*
else (*if k mod 2* = *0 then 2* ∗ *int-of* (*k div 2*) *else 2* ∗ *int-of* (*k div 2*) + *1*))
⟨*proof*⟩

hide-const (**open**) *of-nat nat-of int-of*

34.3.1 Lazy Evaluation of an indexed function

function *iterate-upto* :: (*code-numeral* => *'a*) => *code-numeral* => *code-numeral* => *'a Predicate.pred*
where
iterate-upto f n m = *Predicate.Seq* (%*u*. *if n* > *m then Predicate.Empty else Predicate.Insert* (*f n*) (*iterate-upto f* (*n* + *1*) *m*))
⟨*proof*⟩

termination ⟨*proof*⟩

hide-const (**open**) *iterate-upto*

34.4 Code generator setup

Implementation of indices by bounded integers

code-type *code-numeral*
(*SML int*)
(*OCaml Big'-int.big'-int*)
(*Haskell Int*)
(*Scala Int*)

code-instance *code-numeral* :: *eq*
(*Haskell* −)

⟨*ML*⟩

code-reserved *SML Int int*
code-reserved *Scala Int*

code-const *op* + :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
(*SML Int.+*/ ((-),/ (-)))
(*OCaml Big'-int.add'-big'-int*)
(*Haskell* **infixl** *6* +)
(*Scala* **infixl** *7* +)

code-const *subtract-code-numeral* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
(*SML Int.max*/ (-/ −/ -,/ *0* : *int*))
(*OCaml Big'-int.max'-big'-int*/ (*Big'-int.sub'-big'-int*/ -/ -)/ *Big'-int.zero'-big'-int*)
(*Haskell max*/ (-/ −/ -)/ (*0* :: *Int*))
(*Scala* !(-/ −/ -).*max*(*0*))

code-const *op* $*$ :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
(*SML Int.*/ ((-),/ (-)))
(*OCaml Big'-int.mult'-big'-int*)
(*Haskell* **infixl** *7* $*$)
(*Scala* **infixl** *8* $*$)

code-const *div-mod-code-numeral*
(*SML* !(*fn n => fn m =>/ if m = 0/ then (0, n) else/ (n div m, n mod m)*))
(*OCaml Big'-int.quomod'-big'-int/ (Big'-int.abs'-big'-int -)/ (Big'-int.abs'-big'-int -)*)
(*Haskell divMod*)
(*Scala* !((*n*: *Int*) => (*m*: *Int*) =>/ *if* (*m* == *0*)/ (*0*, *n*) *else*/ (*n* '/ *m*, *n* % *m*)))

code-const *eq-class.eq* :: *code-numeral* ⇒ *code-numeral* ⇒ *bool*
(*SML* !((- : *Int.int*) = -))
(*OCaml Big'-int.eq'-big'-int*)
(*Haskell* **infixl** *4* ==)
(*Scala* **infixl** *5* ==)

code-const *op* ≤ :: *code-numeral* ⇒ *code-numeral* ⇒ *bool*
(*SML Int.<=/* ((-),/ (-)))
(*OCaml Big'-int.le'-big'-int*)
(*Haskell* **infix** *4* <=)
(*Scala* **infixl** *4* <=)

code-const *op* < :: *code-numeral* ⇒ *code-numeral* ⇒ *bool*
(*SML Int.</* ((-),/ (-)))
(*OCaml Big'-int.lt'-big'-int*)
(*Haskell* **infix** *4* <)
(*Scala* **infixl** *4* <)

end

35 Numeral-Simprocs: Combination and Cancellation Simprocs for Numeral Expressions

theory *Numeral-Simprocs*
imports *Divides*
uses
~~/*src*/*Provers*/*Arith*/*assoc-fold.ML*
~~/*src*/*Provers*/*Arith*/*cancel-numerals.ML*
~~/*src*/*Provers*/*Arith*/*combine-numerals.ML*
~~/*src*/*Provers*/*Arith*/*cancel-numeral-factor.ML*
~~/*src*/*Provers*/*Arith*/*extract-common-term.ML*
(*Tools*/*numeral-simprocs.ML*)

(*Tools/nat-numeral-simprocs.ML*)
begin

declare *split-div* [*of - - number-of k*, *standard*, *arith-split*]
declare *split-mod* [*of - - number-of k*, *standard*, *arith-split*]

For *combine-numerals*

lemma *left-add-mult-distrib*: $i*u + (j*u + k) = (i+j)*u + (k::nat)$
⟨*proof*⟩

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*⟩

For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: $(0::nat) < k ==> (k*m <= k*n) = (m<=n)$
⟨*proof*⟩

lemma *nat-mult-less-cancel1*: $(0::nat) < k ==> (k*m < k*n) = (m<n)$
⟨*proof*⟩

lemma *nat-mult-eq-cancel1*: $(0::nat) < k ==> (k*m = k*n) = (m=n)$
⟨*proof*⟩

lemma *nat-mult-div-cancel1*: $(0::nat) < k ==> (k*m)\ div\ (k*n) = (m\ div\ n)$
⟨*proof*⟩

lemma *nat-mult-dvd-cancel-disj*[*simp*]:
$(k*m)\ dvd\ (k*n) = (k=0 \mid m\ dvd\ (n::nat))$
⟨*proof*⟩

lemma *nat-mult-dvd-cancel1*: $0 < k \Longrightarrow (k*m)\ dvd\ (k*n::nat) = (m\ dvd\ n)$
⟨*proof*⟩

For *cancel-factor*

lemma *nat-mult-le-cancel-disj*: $(k*m <= k*n) = ((0::nat) < k \longrightarrow m<=n)$
⟨*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*[*simp*]:
$(k*m)\ div\ (k*n) = (if\ k = (0::nat)\ then\ 0\ else\ m\ div\ n)$
⟨*proof*⟩

⟨*ML*⟩

end

36 Semiring-Normalization: Semiring normalization

theory *Semiring-Normalization*
imports *Numeral-Simprocs Nat-Transfer*
uses
Tools/semiring-normalizer.ML
begin

Prelude

class *comm-semiring-1-cancel-crossproduct* = *comm-semiring-1-cancel* +
assumes *crossproduct-eq*: $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$
begin

lemma *crossproduct-noteq*:

$a \neq b \wedge c \neq d \longleftrightarrow a * c + b * d \neq a * d + b * c$
⟨*proof*⟩

lemma *add-scale-eq-noteq*:
$r \neq 0 \Longrightarrow a = b \wedge c \neq d \Longrightarrow a + r * c \neq b + r * d$
⟨*proof*⟩

lemma *add-0-iff*:
$b = b + a \longleftrightarrow a = 0$
⟨*proof*⟩

end

sublocale *idom* < *comm-semiring-1-cancel-crossproduct*
⟨*proof*⟩

instance *nat* :: *comm-semiring-1-cancel-crossproduct*
⟨*proof*⟩

Semiring normalization proper

⟨*ML*⟩

context *comm-semiring-1*
begin

lemma *normalizing-semiring-ops*:
shows *TERM* $(x + y)$ **and** *TERM* $(x * y)$ **and** *TERM* $(x \hat{} n)$
and *TERM 0* **and** *TERM 1* ⟨*proof*⟩

lemma *normalizing-semiring-rules*:
$(a * m) + (b * m) = (a + b) * m$
$(a * m) + m = (a + 1) * m$
$m + (a * m) = (a + 1) * m$
$m + m = (1 + 1) * m$
$0 + a = a$
$a + 0 = a$
$a * b = b * a$
$(a + b) * c = (a * c) + (b * c)$
$0 * a = 0$
$a * 0 = 0$
$1 * a = a$
$a * 1 = a$
$(lx * ly) * (rx * ry) = (lx * rx) * (ly * ry)$
$(lx * ly) * (rx * ry) = lx * (ly * (rx * ry))$
$(lx * ly) * (rx * ry) = rx * ((lx * ly) * ry)$
$(lx * ly) * rx = (lx * rx) * ly$
$(lx * ly) * rx = lx * (ly * rx)$
$lx * (rx * ry) = (lx * rx) * ry$
$lx * (rx * ry) = rx * (lx * ry)$

$(a + b) + (c + d) = (a + c) + (b + d)$
$(a + b) + c = a + (b + c)$
$a + (c + d) = c + (a + d)$
$(a + b) + c = (a + c) + b$
$a + c = c + a$
$a + (c + d) = (a + c) + d$
$(x \hat{} p) * (x \hat{} q) = x \hat{} (p + q)$
$x * (x \hat{} q) = x \hat{} (Suc\ q)$
$(x \hat{} q) * x = x \hat{} (Suc\ q)$
$x * x = x \hat{} 2$
$(x * y) \hat{} q = (x \hat{} q) * (y \hat{} q)$
$(x \hat{} p) \hat{} q = x \hat{} (p * q)$
$x \hat{} 0 = 1$
$x \hat{} 1 = x$
$x * (y + z) = (x * y) + (x * z)$
$x \hat{} (Suc\ q) = x * (x \hat{} q)$
$x \hat{} (2*n) = (x \hat{} n) * (x \hat{} n)$
$x \hat{} (Suc\ (2*n)) = x * ((x \hat{} n) * (x \hat{} n))$
⟨*proof*⟩

lemmas *normalizing-comm-semiring-1-axioms* =
comm-semiring-1-axioms [*normalizer*
semiring ops: *normalizing-semiring-ops*
semiring rules: *normalizing-semiring-rules*]

⟨*ML*⟩

end

context *comm-ring-1*
begin

lemma *normalizing-ring-ops*: **shows** *TERM* $(x-\ y)$ **and** *TERM* $(-\ x)$ ⟨*proof*⟩

lemma *normalizing-ring-rules*:
$-\ x = (-\ 1) * x$
$x - y = x + (-\ y)$
⟨*proof*⟩

lemmas *normalizing-comm-ring-1-axioms* =
comm-ring-1-axioms [*normalizer*
semiring ops: *normalizing-semiring-ops*
semiring rules: *normalizing-semiring-rules*
ring ops: *normalizing-ring-ops*
ring rules: *normalizing-ring-rules*]

⟨*ML*⟩

end

context *comm-semiring-1-cancel-crossproduct*
begin

declare
normalizing-comm-semiring-1-axioms [*normalizer del*]

lemmas
normalizing-comm-semiring-1-cancel-crossproduct-axioms =
comm-semiring-1-cancel-crossproduct-axioms [*normalizer*
semiring ops: *normalizing-semiring-ops*
semiring rules: *normalizing-semiring-rules*
idom rules: *crossproduct-noteq add-scale-eq-noteq*]

⟨*ML*⟩

end

context *idom*
begin

declare *normalizing-comm-ring-1-axioms* [*normalizer del*]

lemmas *normalizing-idom-axioms* = *idom-axioms* [*normalizer*
semiring ops: *normalizing-semiring-ops*
semiring rules: *normalizing-semiring-rules*
ring ops: *normalizing-ring-ops*
ring rules: *normalizing-ring-rules*
idom rules: *crossproduct-noteq add-scale-eq-noteq*
ideal rules: *right-minus-eq add-0-iff*]

⟨*ML*⟩

end

context *field*
begin

lemma *normalizing-field-ops*:
shows *TERM* (*x* / *y*) **and** *TERM* (*inverse x*) ⟨*proof*⟩

lemmas *normalizing-field-rules* = *divide-inverse inverse-eq-divide*

lemmas *normalizing-field-axioms* =
field-axioms [*normalizer*
semiring ops: *normalizing-semiring-ops*
semiring rules: *normalizing-semiring-rules*
ring ops: *normalizing-ring-ops*
ring rules: *normalizing-ring-rules*

field ops: *normalizing-field-ops*
field rules: *normalizing-field-rules*
idom rules: *crossproduct-noteq add-scale-eq-noteq*
ideal rules: *right-minus-eq add-0-iff*]

⟨*ML*⟩

end

hide-fact (**open**) *normalizing-comm-semiring-1-axioms*
normalizing-comm-semiring-1-cancel-crossproduct-axioms normalizing-semiring-ops
normalizing-semiring-rules

hide-fact (**open**) *normalizing-comm-ring-1-axioms*
normalizing-idom-axioms normalizing-ring-ops normalizing-ring-rules

hide-fact (**open**) *normalizing-field-axioms normalizing-field-ops normalizing-field-rules*

end

37 Groebner-Basis: Groebner bases

theory *Groebner-Basis*
imports *Semiring-Normalization*
uses
(*Tools/groebner.ML*)
begin

37.1 Groebner Bases

lemmas *bool-simps* = *simp-thms*(*1*−*34*)

lemma *dnf*:
$(P \,\&\, (Q \mid R)) = ((P\&Q) \mid (P\&R))$ $((Q \mid R) \,\&\, P) = ((Q\&P) \mid (R\&P))$
$(P \wedge Q) = (Q \wedge P)$ $(P \vee Q) = (Q \vee P)$
⟨*proof*⟩

lemmas *weak-dnf-simps* = *dnf bool-simps*

lemma *nnf-simps*:
$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$ $(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$ $(P \longrightarrow Q) = (\neg P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg\, Q))$ $(\neg\, \neg(P)) = P$
⟨*proof*⟩

lemma *PFalse*:
$P \equiv \mathit{False} \Longrightarrow \neg\, P$
$\neg\, P \Longrightarrow (P \equiv \mathit{False})$
⟨*proof*⟩

⟨*ML*⟩

declare *dvd-def*[*algebra*]
declare *dvd-eq-mod-eq-0*[*symmetric*, *algebra*]
declare *mod-div-trivial*[*algebra*]
declare *mod-mod-trivial*[*algebra*]
declare *conjunct1*[*OF DIVISION-BY-ZERO*, *algebra*]
declare *conjunct2*[*OF DIVISION-BY-ZERO*, *algebra*]
declare *zmod-zdiv-equality*[*symmetric*,*algebra*]
declare *zdiv-zmod-equality*[*symmetric*, *algebra*]
declare *zdiv-zminus-zminus*[*algebra*]
declare *zmod-zminus-zminus*[*algebra*]
declare *zdiv-zminus2*[*algebra*]
declare *zmod-zminus2*[*algebra*]
declare *zdiv-zero*[*algebra*]
declare *zmod-zero*[*algebra*]
declare *mod-by-1*[*algebra*]
declare *div-by-1*[*algebra*]
declare *zmod-minus1-right*[*algebra*]
declare *zdiv-minus1-right*[*algebra*]
declare *mod-div-trivial*[*algebra*]
declare *mod-mod-trivial*[*algebra*]
declare *mod-mult-self2-is-0*[*algebra*]
declare *mod-mult-self1-is-0*[*algebra*]
declare *zmod-eq-0-iff*[*algebra*]
declare *dvd-0-left-iff*[*algebra*]
declare *zdvd1-eq*[*algebra*]
declare *zmod-eq-dvd-iff*[*algebra*]
declare *nat-mod-eq-iff*[*algebra*]

end

38 SetInterval: Set intervals

theory *SetInterval*
imports *Int Nat-Transfer*
begin

context *ord*
begin
definition
  lessThan :: *'a* => *'a set* ((*1*{..<-})) **where**
  {..<*u*} == {*x*. *x* < *u*}

definition
  atMost :: *'a* => *'a set* ((*1*{..-})) **where**
  {..*u*} == {*x*. *x* ≤ *u*}

definition
greaterThan :: *'a => 'a set* ((*1*{-<..})) **where**
{*l*<..} == {*x*. *l*<*x*}

definition
atLeast :: *'a => 'a set* ((*1*{-..})) **where**
{*l*..} == {*x*. *l*≤*x*}

definition
greaterThanLessThan :: *'a => 'a => 'a set* ((*1*{-<..<-})) **where**
{*l*<..<*u*} == {*l*<..} *Int* {..<*u*}

definition
atLeastLessThan :: *'a => 'a => 'a set* ((*1*{-..<-})) **where**
{*l*..<*u*} == {*l*..} *Int* {..<*u*}

definition
greaterThanAtMost :: *'a => 'a => 'a set* ((*1*{-<..-})) **where**
{*l*<..*u*} == {*l*<..} *Int* {..*u*}

definition
atLeastAtMost :: *'a => 'a => 'a set* ((*1*{-..-})) **where**
{*l*..*u*} == {*l*..} *Int* {..*u*}

end

A note of warning when using {..<*n*} on type *nat*: it is equivalent to {*0*..<*n*} but some lemmas involving {*m*..<*n*} may not exist in {..<*n*}-form as well.

syntax
-UNION-le :: *'a => 'a => 'b set => 'b set* ((*3UN -<=-./ -*) [*0*, *0*, *10*] *10*)
-UNION-less :: *'a => 'a => 'b set => 'b set* ((*3UN -<-./ -*) [*0*, *0*, *10*] *10*)
-INTER-le :: *'a => 'a => 'b set => 'b set* ((*3INT -<=-./ -*) [*0*, *0*, *10*] *10*)
-INTER-less :: *'a => 'a => 'b set => 'b set* ((*3INT -<-./ -*) [*0*, *0*, *10*] *10*)

syntax (*xsymbols*)
-UNION-le :: *'a => 'a => 'b set => 'b set* ((*3⋃ -≤-./ -*) [*0*, *0*, *10*] *10*)
-UNION-less :: *'a => 'a => 'b set => 'b set* ((*3⋃ -<-./ -*) [*0*, *0*, *10*] *10*)
-INTER-le :: *'a => 'a => 'b set => 'b set* ((*3⋂ -≤-./ -*) [*0*, *0*, *10*] *10*)
-INTER-less :: *'a => 'a => 'b set => 'b set* ((*3⋂ -<-./ -*) [*0*, *0*, *10*] *10*)

syntax (*latex* **output**)
-UNION-le :: *'a ⇒ 'a => 'b set => 'b set* ((*3⋃(00- ≤ -)/ -*) [*0*, *0*, *10*] *10*)
-UNION-less :: *'a ⇒ 'a => 'b set => 'b set* ((*3⋃(00- < -)/ -*) [*0*, *0*, *10*] *10*)
-INTER-le :: *'a ⇒ 'a => 'b set => 'b set* ((*3⋂(00- ≤ -)/ -*) [*0*, *0*, *10*]

10)
-INTER-less :: *'a* ⇒ *'a* => *'b set* => *'b set* ((*3*⋂(*00-* < *-*)/ *-*) [*0*, *0*, *10*] *10*)

translations
UN i<=n. A == *UN i:{..n}. A*
UN i<n. A == *UN i:{..<n}. A*
INT i<=n. A == *INT i:{..n}. A*
INT i<n. A == *INT i:{..<n}. A*

38.1 Various equivalences

lemma (**in** *ord*) *lessThan-iff* [*iff*]: (*i*: *lessThan k*) = (*i*<*k*)
⟨*proof*⟩

lemma *Compl-lessThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*lessThan k* = *atLeast k*
⟨*proof*⟩

lemma *single-Diff-lessThan* [*simp*]: !!*k*:: *'a*::*order*. {*k*} − *lessThan k* = {*k*}
⟨*proof*⟩

lemma (**in** *ord*) *greaterThan-iff* [*iff*]: (*i*: *greaterThan k*) = (*k*<*i*)
⟨*proof*⟩

lemma *Compl-greaterThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*greaterThan k* = *atMost k*
⟨*proof*⟩

lemma *Compl-atMost* [*simp*]: !!*k*:: *'a*::*linorder*. −*atMost k* = *greaterThan k*
⟨*proof*⟩

lemma (**in** *ord*) *atLeast-iff* [*iff*]: (*i*: *atLeast k*) = (*k*<=*i*)
⟨*proof*⟩

lemma *Compl-atLeast* [*simp*]:
!!*k*:: *'a*::*linorder*. −*atLeast k* = *lessThan k*
⟨*proof*⟩

lemma (**in** *ord*) *atMost-iff* [*iff*]: (*i*: *atMost k*) = (*i*<=*k*)
⟨*proof*⟩

lemma *atMost-Int-atLeast*: !!*n*:: *'a*::*order*. *atMost n Int atLeast n* = {*n*}
⟨*proof*⟩

38.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-subset-iff* [*iff*]:
(*atLeast x* ⊆ *atLeast y*) = (*y* ≤ (*x*::*'a*::*order*))
⟨*proof*⟩

lemma *atLeast-eq-iff* [*iff*]:
(*atLeast x* = *atLeast y*) = (*x* = (*y*::'*a*::*linorder*))
⟨*proof*⟩

lemma *greaterThan-subset-iff* [*iff*]:
(*greaterThan x* ⊆ *greaterThan y*) = (*y* ≤ (*x*::'*a*::*linorder*))
⟨*proof*⟩

lemma *greaterThan-eq-iff* [*iff*]:
(*greaterThan x* = *greaterThan y*) = (*x* = (*y*::'*a*::*linorder*))
⟨*proof*⟩

lemma *atMost-subset-iff* [*iff*]: (*atMost x* ⊆ *atMost y*) = (*x* ≤ (*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* ⊆ *lessThan y*) = (*x* ≤ (*y*::'*a*::*linorder*))
⟨*proof*⟩

lemma *lessThan-eq-iff* [*iff*]:
(*lessThan x* = *lessThan y*) = (*x* = (*y*::'*a*::*linorder*))
⟨*proof*⟩

38.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [*simp*,*no-atp*]:
(*i* : {*l*<..<*u*}) = (*l* < *i* & *i* < *u*)
⟨*proof*⟩

lemma *atLeastLessThan-iff* [*simp*,*no-atp*]:
(*i* : {*l*..<*u*}) = (*l* <= *i* & *i* < *u*)
⟨*proof*⟩

lemma *greaterThanAtMost-iff* [*simp*,*no-atp*]:
(*i* : {*l*<..*u*}) = (*l* < *i* & *i* <= *u*)
⟨*proof*⟩

lemma *atLeastAtMost-iff* [*simp*,*no-atp*]:
(*i* : {*l*..*u*}) = (*l* <= *i* & *i* <= *u*)
⟨*proof*⟩

The above four lemmas could be declared as iffs. Unfortunately this breaks many proofs. Since it only helps blast, it is better to leave well alone

end

38.3.1 Emptyness, singletons, subset

context *order*
begin

lemma *atLeastatMost-empty*[*simp*]:
$b < a \Longrightarrow \{a..b\} = \{\}$
⟨*proof*⟩

lemma *atLeastatMost-empty-iff*[*simp*]:
$\{a..b\} = \{\} \longleftrightarrow (\sim\ a <= b)$
⟨*proof*⟩

lemma *atLeastatMost-empty-iff2*[*simp*]:
$\{\} = \{a..b\} \longleftrightarrow (\sim\ a <= b)$
⟨*proof*⟩

lemma *atLeastLessThan-empty*[*simp*]:
$b <= a \Longrightarrow \{a..<b\} = \{\}$
⟨*proof*⟩

lemma *atLeastLessThan-empty-iff*[*simp*]:
$\{a..<b\} = \{\} \longleftrightarrow (\sim\ a < b)$
⟨*proof*⟩

lemma *atLeastLessThan-empty-iff2*[*simp*]:
$\{\} = \{a..<b\} \longleftrightarrow (\sim\ a < b)$
⟨*proof*⟩

lemma *greaterThanAtMost-empty*[*simp*]: $l \leq k ==> \{k<..l\} = \{\}$
⟨*proof*⟩

lemma *greaterThanAtMost-empty-iff*[*simp*]: $\{k<..l\} = \{\} \longleftrightarrow \sim\ k < l$
⟨*proof*⟩

lemma *greaterThanAtMost-empty-iff2*[*simp*]: $\{\} = \{k<..l\} \longleftrightarrow \sim\ k < l$
⟨*proof*⟩

lemma *greaterThanLessThan-empty*[*simp*]:$l \leq k ==> \{k<..<l\} = \{\}$
⟨*proof*⟩

lemma *atLeastAtMost-singleton* [*simp*]: $\{a..a\} = \{a\}$
⟨*proof*⟩

lemma *atLeastAtMost-singleton'*: $a = b \Longrightarrow \{a\ ..\ b\} = \{a\}$ ⟨*proof*⟩

lemma *atLeastatMost-subset-iff*[*simp*]:

$\{a..b\} <= \{c..d\} \longleftrightarrow (\sim\ a <= b) \mid c <= a\ \&\ b <= d$
⟨*proof*⟩

lemma *atLeastatMost-psubset-iff*:
$\{a..b\} < \{c..d\} \longleftrightarrow$
$((\sim\ a <= b) \mid c <= a\ \&\ b <= d\ \&\ (c < a \mid b < d))\ \ \&\ \ c <= d$
⟨*proof*⟩

lemma *atLeastAtMost-singleton-iff*[*simp*]:
$\{a\ ..\ b\} = \{c\} \longleftrightarrow a = b \wedge b = c$
⟨*proof*⟩

end

lemma (**in** *linorder*) *atLeastLessThan-subset-iff*:
$\{a..<b\} <= \{c..<d\} \Longrightarrow b <= a \mid c<=a\ \&\ b<=d$
⟨*proof*⟩

38.3.2 Intersection

context *linorder*
begin

lemma *Int-atLeastAtMost*[*simp*]: $\{a..b\}\ Int\ \{c..d\} = \{max\ a\ c\ ..\ min\ b\ d\}$
⟨*proof*⟩

lemma *Int-atLeastAtMostR1*[*simp*]: $\{..b\}\ Int\ \{c..d\} = \{c\ ..\ min\ b\ d\}$
⟨*proof*⟩

lemma *Int-atLeastAtMostR2*[*simp*]: $\{a..\}\ Int\ \{c..d\} = \{max\ a\ c\ ..\ d\}$
⟨*proof*⟩

lemma *Int-atLeastAtMostL1*[*simp*]: $\{a..b\}\ Int\ \{..d\} = \{a\ ..\ min\ b\ d\}$
⟨*proof*⟩

lemma *Int-atLeastAtMostL2*[*simp*]: $\{a..b\}\ Int\ \{c..\} = \{max\ a\ c\ ..\ b\}$
⟨*proof*⟩

lemma *Int-atLeastLessThan*[*simp*]: $\{a..<b\}\ Int\ \{c..<d\} = \{max\ a\ c\ ..<\ min\ b\ d\}$
⟨*proof*⟩

lemma *Int-greaterThanAtMost*[*simp*]: $\{a<..b\}\ Int\ \{c<..d\} = \{max\ a\ c\ <..\ min\ b\ d\}$
⟨*proof*⟩

lemma *Int-greaterThanLessThan*[*simp*]: $\{a<..<b\}\ Int\ \{c<..<d\} = \{max\ a\ c\ <..<\ min\ b\ d\}$
⟨*proof*⟩

end

38.4 Intervals of natural numbers

38.4.1 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*⟩

38.4.2 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*⟩

38.4.3 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*⟩

38.4.4 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*⟩

38.4.5 The Constant *atLeastLessThan*

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

lemma *atLeast0LessThan*: $\{0{::}nat..<n\} = \{..<n\}$
⟨*proof*⟩

lemma *atLeast0AtMost*: $\{0..n{::}nat\} = \{..n\}$
⟨*proof*⟩

declare *atLeast0LessThan*[*symmetric*, *code-unfold*]
atLeast0AtMost[*symmetric*, *code-unfold*]

lemma *atLeastLessThan0*: $\{m..<0{::}nat\} = \{\}$
⟨*proof*⟩

38.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
$\{m..<Suc\ n\} = (if\ m \le n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$
⟨*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*: $m \le Suc\ n \Longrightarrow \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
⟨*proof*⟩

lemma *atLeastLessThan-add-Un*: $i \le j \Longrightarrow \{i..<j+k\} = \{i..<j\} \cup \{j..<j+k{::}nat\}$
⟨*proof*⟩

38.4.7 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*⟩

context *ordered-ab-group-add*
begin

lemma
fixes *x* :: *'a*
shows *image-uminus-greaterThan*[*simp*]: *uminus* ‘ {*x*<..} = {..<−*x*}
and *image-uminus-atLeast*[*simp*]: *uminus* ‘ {*x*..} = {..−*x*}
⟨*proof*⟩

lemma
fixes *x* :: *'a*
shows *image-uminus-lessThan*[*simp*]: *uminus* ‘ {..<*x*} = {−*x*<..}
and *image-uminus-atMost*[*simp*]: *uminus* ‘ {..*x*} = {−*x*..}
⟨*proof*⟩

lemma
fixes *x* :: *'a*
shows *image-uminus-atLeastAtMost*[*simp*]: *uminus* ‘ {*x*..*y*} = {−*y*..−*x*}
and *image-uminus-greaterThanAtMost*[*simp*]: *uminus* ‘ {*x*<..*y*} = {−*y*..<−*x*}
and *image-uminus-atLeastLessThan*[*simp*]: *uminus* ‘ {*x*..<*y*} = {−*y*<..−*x*}
and *image-uminus-greaterThanLessThan*[*simp*]: *uminus* ‘ {*x*<..<*y*} = {−*y*<..<−*x*}
⟨*proof*⟩
end

38.4.8 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*⟩

A bounded set of natural numbers is finite.

lemma *bounded-nat-set-is-finite*:
$(ALL\ i{:}N.\ i < (n{::}nat)) ==>$ *finite* N
⟨*proof*⟩

A set of natural numbers is finite iff it is bounded.

lemma *finite-nat-set-iff-bounded*:
finite$(N{::}nat\ set) = (EX\ m.\ ALL\ n{:}N.\ n<m)$ (**is** *?F = ?B*)
⟨*proof*⟩

lemma *finite-nat-set-iff-bounded-le*:
finite$(N{::}nat\ set) = (EX\ m.\ ALL\ n{:}N.\ n<=m)$
⟨*proof*⟩

lemma *finite-less-ub*:
$!!f{::}nat=>nat.\ (!!n.\ n \le f\ n) ==>$ *finite* $\{n.\ f\ n \le u\}$
⟨*proof*⟩

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
$A <= \{k..<k+card\ A\} \Longrightarrow A = \{k..<k+card\ A\}$ (**is** *PROP ?P*)
⟨*proof*⟩

38.4.9 Proving Inclusions and Equalities between Unions

lemma *UN-le-eq-Un0*:
$(\bigcup i{\le}n{::}nat.\ M\ i) = (\bigcup i{\in}\{1..n\}.\ M\ i) \cup M\ 0$ (**is** *?A = ?B*)
⟨*proof*⟩

lemma *UN-le-add-shift*:
$(\bigcup i{\leq}n{::}nat.\ M(i+k)) = (\bigcup i{\in}\{k..n+k\}.\ M\ i)$ (**is** *?A = ?B*)
⟨*proof*⟩

lemma *UN-UN-finite-eq*: $(\bigcup n{::}nat.\ \bigcup i{\in}\{0..{<}n\}.\ A\ i) = (\bigcup n.\ A\ n)$
⟨*proof*⟩

lemma *UN-finite-subset*: $(!!n{::}nat.\ (\bigcup i{\in}\{0..{<}n\}.\ A\ i) \subseteq C) \Longrightarrow (\bigcup n.\ A\ n) \subseteq C$
⟨*proof*⟩

lemma *UN-finite2-subset*:
$(!!n{::}nat.\ (\bigcup i{\in}\{0..{<}n\}.\ A\ i) \subseteq (\bigcup i{\in}\{0..{<}n+k\}.\ B\ i)) \Longrightarrow (\bigcup n.\ A\ n) \subseteq (\bigcup n.\ B\ n)$
⟨*proof*⟩

lemma *UN-finite2-eq*:
$(!!n{::}nat.\ (\bigcup i{\in}\{0..{<}n\}.\ A\ i) = (\bigcup i{\in}\{0..{<}n+k\}.\ B\ i)) \Longrightarrow (\bigcup n.\ A\ n) = (\bigcup n.\ B\ n)$
⟨*proof*⟩

38.4.10 Cardinality

lemma *card-lessThan* [*simp*]: $card\ \{..{<}u\} = u$
⟨*proof*⟩

lemma *card-atMost* [*simp*]: $card\ \{..u\} = Suc\ u$
⟨*proof*⟩

lemma *card-atLeastLessThan* [*simp*]: $card\ \{l..{<}u\} = u - l$
⟨*proof*⟩

lemma *card-atLeastAtMost* [*simp*]: $card\ \{l..u\} = Suc\ u - l$
⟨*proof*⟩

lemma *card-greaterThanAtMost* [*simp*]: $card\ \{l{<}..u\} = u - l$
⟨*proof*⟩

lemma *card-greaterThanLessThan* [*simp*]: $card\ \{l{<}..{<}u\} = u - Suc\ l$
⟨*proof*⟩

lemma *ex-bij-betw-nat-finite*:
$finite\ M \Longrightarrow \exists h.\ bij\text{-}betw\ h\ \{0..{<}card\ M\}\ M$
⟨*proof*⟩

lemma *ex-bij-betw-finite-nat*:
$finite\ M \Longrightarrow \exists h.\ bij\text{-}betw\ h\ M\ \{0..{<}card\ M\}$
⟨*proof*⟩

lemma *finite-same-card-bij*:
finite A ⟹ finite B ⟹ card A = card B ⟹ EX h. bij-betw h A B
⟨*proof*⟩

lemma *ex-bij-betw-nat-finite-1*:
finite M ⟹ ∃ h. bij-betw h {1 .. card M} M
⟨*proof*⟩

38.5 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*⟩

38.5.1 Finiteness

lemma *image-atLeastZeroLessThan-int*: *0* ≤ *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*⟩

38.5.2 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*⟩

lemma *finite-M-bounded-by-nat*: *finite* {*k*. *P k* ∧ *k* < (*i*::*nat*)}
⟨*proof*⟩

lemma *card-less*:
assumes *zero-in-M*: *0* ∈ *M*
shows *card* {*k* ∈ *M*. *k* < *Suc i*} ≠ *0*
⟨*proof*⟩

lemma *card-less-Suc2*: *0* ∉ *M* ⟹ *card* {*k*. *Suc k* ∈ *M* ∧ *k* < *i*} = *card* {*k* ∈ *M*. *k* < *Suc i*}
⟨*proof*⟩

lemma *card-less-Suc*:
assumes *zero-in-M*: *0* ∈ *M*
shows *Suc* (*card* {*k*. *Suc k* ∈ *M* ∧ *k* < *i*}) = *card* {*k* ∈ *M*. *k* < *Suc i*}
⟨*proof*⟩

38.6 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy

38.6.1 Disjoint Unions

Singletons and open intervals

lemma *ivl-disj-un-singleton*:
{*l*::'*a*::*linorder*} *Un* {*l*<..} = {*l*..}
{..<*u*} *Un* {*u*::'*a*::*linorder*} = {..*u*}
(*l*::'*a*::*linorder*) < *u* ==> {*l*} *Un* {*l*<..<*u*} = {*l*..<*u*}
(*l*::'*a*::*linorder*) < *u* ==> {*l*<..<*u*} *Un* {*u*} = {*l*<..*u*}
(*l*::'*a*::*linorder*) <= *u* ==> {*l*} *Un* {*l*<..*u*} = {*l*..*u*}
(*l*::'*a*::*linorder*) <= *u* ==> {*l*..<*u*} *Un* {*u*} = {*l*..*u*}
⟨*proof*⟩

One- and two-sided intervals

lemma *ivl-disj-un-one*:
(*l*::'*a*::*linorder*) < *u* ==> {..*l*} *Un* {*l*<..<*u*} = {..<*u*}
(*l*::'*a*::*linorder*) <= *u* ==> {..<*l*} *Un* {*l*..<*u*} = {..<*u*}
(*l*::'*a*::*linorder*) <= *u* ==> {..*l*} *Un* {*l*<..*u*} = {..*u*}
(*l*::'*a*::*linorder*) <= *u* ==> {..<*l*} *Un* {*l*..*u*} = {..*u*}
(*l*::'*a*::*linorder*) <= *u* ==> {*l*<..*u*} *Un* {*u*<..} = {*l*<..}

$(l::'a::linorder) < u ==> \{l<..<u\}\ Un\ \{u..\} = \{l<..\}$
$(l::'a::linorder) <= u ==> \{l..u\}\ Un\ \{u<..\} = \{l..\}$
$(l::'a::linorder) <= u ==> \{l..<u\}\ Un\ \{u..\} = \{l..\}$
⟨*proof*⟩

Two- and two-sided intervals

lemma *ivl-disj-un-two*:
$[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..<u\} = \{l<..<u\}$
$[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l<..m\}\ Un\ \{m<..<u\} = \{l<..<u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..<u\} = \{l..<u\}$
$[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l..m\}\ Un\ \{m<..<u\} = \{l..<u\}$
$[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..u\} = \{l<..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l<..m\}\ Un\ \{m<..u\} = \{l<..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..u\} = \{l..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..m\}\ Un\ \{m<..u\} = \{l..u\}$
⟨*proof*⟩

lemmas *ivl-disj-un = ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two*

38.6.2 Disjoint Intersections

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-one ivl-disj-int-two*

38.6.3 Some Differences

lemma *ivl-diff* [*simp*]:
$i \leq n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
⟨*proof*⟩

38.6.4 Some Subset Conditions

lemma *ivl-subset* [*simp*,*no-atp*]:
$(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \mid m \leq i \ \& \ j \leq (n::'a::linorder))$
⟨*proof*⟩

38.7 Summation indexed over intervals

syntax
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM - = -..-./ -*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM - = -..<-./ -*) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM -<-./ -*) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM -<=-./ -*) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *- = -..-./ -*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *- = -..<-./ -*) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *-<-./ -*) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *-≤-./ -*) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *- = -..-./ -*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *- = -..<-./ -*) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *-<-./ -*) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ *-≤-./ -*) [*0*,*0*,*10*] *10*)
syntax (*latex-sum* **output**)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*$\sum_{-}^{-} = \_$ *-*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*$\sum_{-}^{<-} = \_$ *-*) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*$\sum_{-} < \_$ *-*) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*$\sum_{-} \leq \_$ *-*) [*0*,*0*,*10*] *10*)

translations
$\sum x{=}a..b.\ t == CONST\ setsum\ (\%x.\ t)\ \{a..b\}$
$\sum x{=}a..{<}b.\ t == CONST\ setsum\ (\%x.\ t)\ \{a..{<}b\}$
$\sum i{\leq}n.\ t == CONST\ setsum\ (\lambda i.\ t)\ \{..n\}$
$\sum i{<}n.\ t == CONST\ setsum\ (\lambda i.\ t)\ \{..{<}n\}$

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	LaTeX
$\sum x{\in}\{a..b\}.\ e$	$\sum x = a..b.\ e$	$\sum_{x = a}^{b} e$
$\sum x{\in}\{a..<b\}.\ e$	$\sum x = a..<b.\ e$	$\sum_{x = a}^{<b} e$
$\sum x{\in}\{..b\}.\ e$	$\sum x{\leq}b.\ e$	$\sum_{x \leq b} e$
$\sum x{\in}\{..<b\}.\ e$	$\sum x{<}b.\ e$	$\sum_{x < b} e$

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*:
$[\![a = c;\ b = d;\ !!x.\ [\![c \leq x;\ x < d]\!] \Longrightarrow f\,x = g\,x]\!] \Longrightarrow$
$setsum\ f\ \{a..<b\} = setsum\ g\ \{c..<d\}$
⟨*proof*⟩

lemma *setsum-atMost-Suc*[*simp*]: $(\sum i \leq Suc\ n.\ f\,i) = (\sum i \leq n.\ f\,i) + f(Suc\ n)$
⟨*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-head*:
fixes n :: *nat*
assumes *mn*: $m <= n$
shows $(\sum x{\in}\{m..n\}.\ P\ x) = P\ m + (\sum x{\in}\{m<..n\}.\ P\ x)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *setsum-head-Suc*:
$m \leq n \Longrightarrow setsum\ f\ \{m..n\} = f\ m + setsum\ f\ \{Suc\ m..n\}$
⟨*proof*⟩

lemma *setsum-head-upt-Suc*:
$m < n \Longrightarrow setsum\ f\ \{m..<n\} = f\ m + setsum\ f\ \{Suc\ m..<n\}$
⟨*proof*⟩

lemma *setsum-ub-add-nat*: **assumes** $(m::nat) \leq n + 1$
shows $setsum\ f\ \{m..n + p\} = setsum\ f\ \{m..n\} + setsum\ f\ \{n + 1..n + p\}$
⟨*proof*⟩

lemma *setsum-add-nat-ivl*: $[\![\ m \leq n;\ n \leq p\ ]\!] \Longrightarrow$
$setsum\ f\ \{m..<n\} + setsum\ f\ \{n..<p\} = setsum\ f\ \{m..<p::nat\}$
⟨*proof*⟩

lemma *setsum-diff-nat-ivl*:
fixes $f :: nat \Rightarrow 'a::ab\text{-}group\text{-}add$
shows $[\![\ m \leq n;\ n \leq p\ ]\!] \Longrightarrow$
$setsum\ f\ \{m..<p\} - setsum\ f\ \{m..<n\} = setsum\ f\ \{n..<p\}$
⟨*proof*⟩

lemma *setsum-natinterval-difff*:
fixes $f::\ nat \Rightarrow ('a::ab\text{-}group\text{-}add)$
shows $setsum\ (\lambda k.\ f\ k - f(k + 1))\ \{(m::nat)\ ..\ n\} =$
$(if\ m <= n\ then\ f\ m - f(n + 1)\ else\ 0)$
⟨*proof*⟩

lemmas *setsum-restrict-set'* = *setsum-restrict-set*[*unfolded Int-def*]

lemma *setsum-setsum-restrict*:
$finite\ S \Longrightarrow finite\ T \Longrightarrow setsum\ (\lambda x.\ setsum\ (\lambda y.\ f\ x\ y)\ \{y.\ y \in T \wedge R\ x\ y\})\ S$
$= setsum\ (\lambda y.\ setsum\ (\lambda x.\ f\ x\ y)\ \{x.\ x \in S \wedge R\ x\ y\})\ T$
⟨*proof*⟩

lemma *setsum-image-gen*: **assumes** *fS*: *finite S*
shows $setsum\ g\ S = setsum\ (\lambda y.\ setsum\ g\ \{x.\ x \in S \wedge f\ x = y\})\ (f\ `\ S)$
⟨*proof*⟩

lemma *setsum-le-included*:
fixes $f :: 'a \Rightarrow 'b::ordered\text{-}comm\text{-}monoid\text{-}add$
assumes *finite s finite t*
and $\forall y \in t.\ 0 \leq g\ y\ (\forall x \in s.\ \exists y \in t.\ i\ y = x \wedge f\ x \leq g\ y)$
shows $setsum\ f\ s \leq setsum\ g\ t$
⟨*proof*⟩

lemma *setsum-multicount-gen*:
assumes *finite s finite t* $\forall j \in t.\ (card\ \{i \in s.\ R\ i\ j\} = k\ j)$
shows $setsum\ (\lambda i.\ (card\ \{j \in t.\ R\ i\ j\}))\ s = setsum\ k\ t$ (**is** *?l* = *?r*)

⟨*proof*⟩

lemma *setsum-multicount*:
assumes *finite S finite T* ∀*j*∈*T*. (*card* {*i*∈*S*. *R i j*} = *k*)
shows *setsum* (λ*i*. *card* {*j*∈*T*. *R i j*}) *S* = *k* ∗ *card T* (**is** *?l* = *?r*)
⟨*proof*⟩

38.8 Shifting bounds

lemma *setsum-shift-bounds-nat-ivl*:
setsum f {*m*+*k*..<*n*+*k*} = *setsum* (%*i*. *f*(*i* + *k*)){*m*..<*n*::*nat*}
⟨*proof*⟩

lemma *setsum-shift-bounds-cl-nat-ivl*:
setsum f {*m*+*k*..*n*+*k*} = *setsum* (%*i*. *f*(*i* + *k*)){*m*..*n*::*nat*}
⟨*proof*⟩

corollary *setsum-shift-bounds-cl-Suc-ivl*:
setsum f {*Suc m*..*Suc n*} = *setsum* (%*i*. *f*(*Suc i*)){*m*..*n*}
⟨*proof*⟩

corollary *setsum-shift-bounds-Suc-ivl*:
setsum f {*Suc m*..<*Suc n*} = *setsum* (%*i*. *f*(*Suc i*)){*m*..<*n*}
⟨*proof*⟩

lemma *setsum-shift-lb-Suc0-0*:
f(*0*::*nat*) = (*0*::*nat*) ⟹ *setsum f* {*Suc 0*..*k*} = *setsum f* {*0*..*k*}
⟨*proof*⟩

lemma *setsum-shift-lb-Suc0-0-upt*:
f(*0*::*nat*) = *0* ⟹ *setsum f* {*Suc 0*..<*k*} = *setsum f* {*0*..<*k*}
⟨*proof*⟩

38.9 The formula for geometric sums

lemma *geometric-sum*:
assumes *x* ≠ *1*
shows (∑ *i*=*0*..<*n*. *x* ^ *i*) = (*x* ^ *n* − *1*) / (*x* − *1*::*'a*::*field*)
⟨*proof*⟩

38.10 The formula for arithmetic sums

lemma *gauss-sum*:
((*1*::*'a*::*comm-semiring-1*) + *1*)∗(∑ *i*∈{*1*..*n*}. *of-nat i*) =
of-nat n∗((*of-nat n*)+*1*)
⟨*proof*⟩

theorem *arith-series-general*:
((*1*::*'a*::*comm-semiring-1*) + *1*) ∗ (∑ *i*∈{..<*n*}. *a* + *of-nat i* ∗ *d*) =
of-nat n ∗ (*a* + (*a* + *of-nat*(*n* − *1*)∗*d*))

⟨*proof*⟩

lemma *arith-series-nat*:
*Suc (Suc 0) * (∑ i∈{..<n}. a+i*d) = n * (a + (a+(n − 1)*d))*
⟨*proof*⟩

lemma *arith-series-int*:
*(2::int) * (∑ i∈{..<n}. a + of-nat i * d) =*
*of-nat n * (a + (a + of-nat(n − 1)*d))*
⟨*proof*⟩

lemma *sum-diff-distrib*:
fixes *P::nat⇒nat*
shows
∀ x. Q x ≤ P x ⟹
(∑ x<n. P x) − (∑ x<n. Q x) = (∑ x<n. P x − Q x)
⟨*proof*⟩

38.11 Products indexed over intervals

syntax
-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((PROD - = -..-./ -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((PROD - = -..<-./ -) [0,0,0,10] 10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((PROD -<-./ -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((PROD -<=-./ -) [0,0,10] 10)
syntax (*xsymbols*)
-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = -..-./ -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = -..<-./ -) [0,0,0,10] 10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -<-./ -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -≤-./ -) [0,0,10] 10)
syntax (*HTML* **output**)
-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = -..-./ -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = -..<-./ -) [0,0,0,10] 10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -<-./ -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -≤-./ -) [0,0,10] 10)
syntax (*latex-prod* **output**)
-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b
((∏_ = _ -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b
((∏_ <= _ -) [0,0,0,10] 10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b
((∏_ < _ -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b
((∏_ ≤ _ -) [0,0,10] 10)

translations

$\prod x{=}a..b.\ t == CONST\ setprod\ (\%x.\ t)\ \{a..b\}$
$\prod x{=}a..{<}b.\ t == CONST\ setprod\ (\%x.\ t)\ \{a..{<}b\}$
$\prod i{\leq}n.\ t == CONST\ setprod\ (\lambda i.\ t)\ \{..n\}$
$\prod i{<}n.\ t == CONST\ setprod\ (\lambda i.\ t)\ \{..{<}n\}$

38.12 Transfer setup

lemma *transfer-nat-int-set-functions*:
$\{..n\} = nat\ `\ \{0..int\ n\}$
$\{m..n\} = nat\ `\ \{int\ m..int\ n\}$
⟨*proof*⟩

lemma *transfer-nat-int-set-function-closures*:
$x >= 0 \Longrightarrow nat\text{-}set\ \{x..y\}$
⟨*proof*⟩

declare *transfer-morphism-nat-int*[*transfer add*
return: *transfer-nat-int-set-functions*
transfer-nat-int-set-function-closures
]

lemma *transfer-int-nat-set-functions*:
$is\text{-}nat\ m \Longrightarrow is\text{-}nat\ n \Longrightarrow \{m..n\} = int\ `\ \{nat\ m..nat\ n\}$
⟨*proof*⟩

lemma *transfer-int-nat-set-function-closures*:
$is\text{-}nat\ x \Longrightarrow nat\text{-}set\ \{x..y\}$
⟨*proof*⟩

declare *transfer-morphism-int-nat*[*transfer add*
return: *transfer-int-nat-set-functions*
transfer-int-nat-set-function-closures
]

end

39 Presburger: Decision Procedure for Presburger Arithmetic

theory *Presburger*
imports *Groebner-Basis SetInterval*
uses
Tools/Qelim/qelim.ML
Tools/Qelim/cooper-procedure.ML
(*Tools/Qelim/cooper.ML*)
begin

39.1 The $-\infty$ and $+\infty$ Properties

lemma *minf*:
$[\![\exists (z::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z::'a::\{linorder\}).\forall x<z.(x = t) = False$
$\exists (z::'a::\{linorder\}).\forall x<z.(x \neq t) = True$
$\exists (z::'a::\{linorder\}).\forall x<z.(x < t) = True$
$\exists (z::'a::\{linorder\}).\forall x<z.(x \leq t) = True$
$\exists (z::'a::\{linorder\}).\forall x<z.(x > t) = False$
$\exists (z::'a::\{linorder\}).\forall x<z.(x \geq t) = False$
$\exists z.\forall (x::'a::\{linorder,plus,Rings.dvd\})<z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Rings.dvd\})<z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x<z.\ F = F$
⟨*proof*⟩

lemma *pinf*:
$[\![\exists (z::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z::'a::\{linorder\}).\forall x>z.(x = t) = False$
$\exists (z::'a::\{linorder\}).\forall x>z.(x \neq t) = True$
$\exists (z::'a::\{linorder\}).\forall x>z.(x < t) = False$
$\exists (z::'a::\{linorder\}).\forall x>z.(x \leq t) = False$
$\exists (z::'a::\{linorder\}).\forall x>z.(x > t) = True$
$\exists (z::'a::\{linorder\}).\forall x>z.(x \geq t) = True$
$\exists z.\forall (x::'a::\{linorder,plus,Rings.dvd\})>z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Rings.dvd\})>z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x>z.\ F = F$
⟨*proof*⟩

lemma *inf-period*:
$[\![\forall x\ k.\ P\ x = P\ (x - k*D);\ \forall x\ k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \wedge Q\ x) = (P\ (x - k*D) \wedge Q\ (x - k*D))$
$[\![\forall x\ k.\ P\ x = P\ (x - k*D);\ \forall x\ k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \vee Q\ x) = (P\ (x - k*D) \vee Q\ (x - k*D))$
$(d::'a::\{comm\text{-}ring,Rings.dvd\})\ dvd\ D \Longrightarrow \forall x\ k.\ (d\ dvd\ x + t) = (d\ dvd\ (x - k*D) + t)$
$(d::'a::\{comm\text{-}ring,Rings.dvd\})\ dvd\ D \Longrightarrow \forall x\ k.\ (\neg d\ dvd\ x + t) = (\neg d\ dvd\ (x - k*D) + t)$
$\forall x\ k.\ F = F$
⟨*proof*⟩

39.2 The A and B sets

lemma *bset*:
$[\![\forall x.(\forall j \in \{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)\ ;$

$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x - D) \wedge Q\ (x - D))$
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ P\ x \longrightarrow P(x - D)$;
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (P\ x \vee Q\ x) \longrightarrow (P(x - D) \vee Q\ (x - D))$
$[\![D{>}0;\ t - 1{\in}\ B]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (x = t) \longrightarrow (x - D = t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (x \neq t) \longrightarrow (x - D \neq t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (x < t) \longrightarrow (x - D < t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (x \leq t) \longrightarrow (x - D \leq t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (x > t) \longrightarrow (x - D > t))$
$[\![D{>}0\ ;\ t - 1 \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (x \geq t) \longrightarrow (x - D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x - D) + t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow} (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x - D) + t))$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow F \longrightarrow F$
⟨*proof*⟩

lemma *aset*:
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ P\ x \longrightarrow P(x + D)$;
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x + D) \wedge Q\ (x + D))$
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ P\ x \longrightarrow P(x + D)$;
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (P\ x \vee Q\ x) \longrightarrow (P(x + D) \vee Q\ (x + D))$
$[\![D{>}0;\ t + 1{\in}\ A]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (x = t) \longrightarrow (x + D = t))$
$[\![D{>}0\ ;\ t \in A]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (x \neq t) \longrightarrow (x + D \neq t))$
$[\![D{>}0;\ t{\in}\ A]\!] \Longrightarrow (\forall (x{::}int).\ (\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (x < t) \longrightarrow (x + D < t))$
$[\![D{>}0;\ t + 1 \in A]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (x \leq t) \longrightarrow (x + D \leq t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (x > t) \longrightarrow (x + D > t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (x \geq t) \longrightarrow (x + D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow} (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x + D) + t))$

$d\ dvd\ D \Longrightarrow (\forall\,(x{::}int).(\forall\,j{\in}\{1\ ..\ D\}.\ \forall\,b{\in}A.\ x \neq b - j) \longrightarrow (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x + D) + t))$
$\forall\,x.(\forall\,j{\in}\{1\ ..\ D\}.\ \forall\,b{\in}A.\ x \neq b - j) \longrightarrow F \longrightarrow F$
⟨*proof*⟩

39.3 Cooper's Theorem $-\infty$ and $+\infty$ Version

39.3.1 First some trivial facts about periodic sets or predicates

lemma *periodic-finite-ex*:
assumes *dpos*: $(0{::}int) < d$ **and** *modd*: $ALL\ x\ k.\ P\ x = P(x - k{*}d)$
shows $(EX\ x.\ P\ x) = (EX\ j : \{1..d\}.\ P\ j)$
(**is** *?LHS = ?RHS*)
⟨*proof*⟩

39.3.2 The $-\infty$ Version

lemma *decr-lemma*: $0 < (d{::}int) \Longrightarrow x - (abs(x{-}z){+}1) * d < z$
⟨*proof*⟩

lemma *incr-lemma*: $0 < (d{::}int) \Longrightarrow z < x + (abs(x{-}z){+}1) * d$
⟨*proof*⟩

lemma *decr-mult-lemma*:
assumes *dpos*: $(0{::}int) < d$ **and** *minus*: $\forall\,x.\ P\ x \longrightarrow P(x - d)$ **and** *knneg*: $0 <= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x - k{*}d)$
⟨*proof*⟩

lemma *minusinfinity*:
assumes *dpos*: $0 < d$ **and**
P1eqP1: $ALL\ x\ k.\ P1\ x = P1(x - k{*}d)$ **and** *ePeqP1*: $EX\ z{::}int.\ ALL\ x.\ x < z \longrightarrow (P\ x = P1\ x)$
shows $(EX\ x.\ P1\ x) \longrightarrow (EX\ x.\ P\ x)$
⟨*proof*⟩

lemma *cpmi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists\,z.\ \forall\ x{<}\ z.\ P\ x = P'\ x$
and *nb*:$\forall\,x.(\forall\ j{\in}\ \{1..D\}.\ \forall\,(b{::}int) \in B.\ x \neq b{+}j) --> P\ (x) --> P\ (x - D)$
and *pd*: $\forall\ x\ k.\ P'\ x = P'\ (x{-}k{*}D)$
shows $(\exists\,x.\ P\ x) = ((\exists\ j{\in}\ \{1..D\}\ .\ P'\ j)\ |\ (\exists\ j\ \in \{1..D\}.\exists\ b{\in}\ B.\ P\ (b{+}j)))$
(**is** *?L* = (*?R1* ∨ *?R2*))
⟨*proof*⟩

39.3.3 The $+\infty$ Version

lemma *plusinfinity*:
assumes *dpos*: $(0{::}int) < d$ **and**
P1eqP1: $\forall\,x\ k.\ P'\ x = P'(x - k{*}d)$ **and** *ePeqP1*: $\exists\ z.\ \forall\ x{>}z.\ P\ x = P'\ x$

shows $(\exists\ x.\ P'\ x) \longrightarrow (\exists\ x.\ P\ x)$
⟨*proof*⟩

lemma *incr-mult-lemma*:
assumes *dpos*: $(0::int) < d$ **and** *plus*: $ALL\ x::int.\ P\ x \longrightarrow P(x + d)$ **and** *knneg*: $0 <= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x + k*d)$
⟨*proof*⟩

lemma *cppi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists z.\ \forall\ x> z.\ P\ x = P'\ x$
and *nb*:$\forall x.(\forall\ j\in \{1..D\}.\ \forall (b::int) \in A.\ x \neq b - j) --> P\ (x) --> P\ (x + D)$
and *pd*: $\forall\ x\ k.\ P'\ x= P'\ (x-k*D)$
shows $(\exists x.\ P\ x) = ((\exists\ j\in \{1..D\}\ .\ P'\ j) \mid (\exists\ j \in \{1..D\}.\exists\ b\in A.\ P\ (b - j)))$
(**is** $?L = (?R1 \vee ?R2)$)
⟨*proof*⟩

lemma *simp-from-to*: $\{i..j::int\} = (if\ j < i\ then\ \{\}\ else\ insert\ i\ \{i+1..j\})$
⟨*proof*⟩

theorem *unity-coeff-ex*: $(\exists (x::'a::\{semiring-0,Rings.dvd\}).\ P\ (l * x)) \equiv (\exists x.\ l\ dvd\ (x + 0) \wedge P\ x)$
⟨*proof*⟩

lemma *zdvd-mono*: **assumes** *not0*: $(k::int) \neq 0$
shows $((m::int)\ dvd\ t) \equiv (k*m\ dvd\ k*t)$
⟨*proof*⟩

lemma *uminus-dvd-conv*: $(d\ dvd\ (t::int)) \equiv (-d\ dvd\ t)\ (d\ dvd\ (t::int)) \equiv (d\ dvd\ -t)$
⟨*proof*⟩

Theorems for transforming predicates on nat to predicates on *int*

lemma *zdiff-int-split*: $P\ (int\ (x - y)) =$
$((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
⟨*proof*⟩

lemma *number-of1*: $(0::int) <= number\text{-}of\ n \Longrightarrow (0::int) <= number\text{-}of\ (Int.Bit0\ n) \wedge (0::int) <= number\text{-}of\ (Int.Bit1\ n)$
⟨*proof*⟩

lemma *number-of2*: $(0::int) <= Numeral0$ ⟨*proof*⟩

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*⟩

declare *dvd-eq-mod-eq-0*[*symmetric*, *presburger*]
declare *mod-1*[*presburger*]
declare *mod-0*[*presburger*]
declare *mod-by-1*[*presburger*]
declare *zmod-zero*[*presburger*]
declare *zmod-self*[*presburger*]
declare *mod-self*[*presburger*]
declare *mod-by-0*[*presburger*]
declare *mod-div-trivial*[*presburger*]
declare *div-mod-equality2*[*presburger*]
declare *div-mod-equality*[*presburger*]
declare *mod-div-equality2*[*presburger*]
declare *mod-div-equality*[*presburger*]
declare *mod-mult-self1*[*presburger*]
declare *mod-mult-self2*[*presburger*]
declare *zdiv-zmod-equality2*[*presburger*]
declare *zdiv-zmod-equality*[*presburger*]
declare *mod2-Suc-Suc*[*presburger*]
lemma [*presburger*]: $(a::int)\ div\ 0 = 0$ **and** [*presburger*]: $a\ mod\ 0 = a$
⟨*proof*⟩

lemma [*presburger*, *algebra*]: $m\ mod\ 2 = (1::nat) \longleftrightarrow \neg\ 2\ dvd\ m$ ⟨*proof*⟩
lemma [*presburger*, *algebra*]: $m\ mod\ 2 = Suc\ 0 \longleftrightarrow \neg\ 2\ dvd\ m$ ⟨*proof*⟩
lemma [*presburger*, *algebra*]: $m\ mod\ (Suc\ (Suc\ 0)) = (1::nat) \longleftrightarrow \neg\ 2\ dvd\ m$
⟨*proof*⟩
lemma [*presburger*, *algebra*]: $m\ mod\ (Suc\ (Suc\ 0)) = Suc\ 0 \longleftrightarrow \neg\ 2\ dvd\ m$
⟨*proof*⟩
lemma [*presburger*, *algebra*]: $m\ mod\ 2 = (1::int) \longleftrightarrow \neg\ 2\ dvd\ m$ ⟨*proof*⟩

end

40 Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *Nat Wellfounded Plain*
uses (*Tools/meson.ML*) (*Tools/choice-specification.ML*)
begin

40.1 Hilbert's epsilon

axiomatization *Eps* :: (*'a => bool*) => *'a* **where**
someI: *P x ==> P (Eps P)*

syntax (*epsilon*)
-Eps :: [*pttrn*, *bool*] => *'a* ((*3ϵ -./ -*) [*0*, *10*] *10*)
syntax (*HOL*)
-Eps :: [*pttrn*, *bool*] => *'a* ((*3@ -./ -*) [*0*, *10*] *10*)
syntax
-Eps :: [*pttrn*, *bool*] => *'a* ((*3SOME -./ -*) [*0*, *10*] *10*)
translations
SOME x. P == CONST Eps (%x. P)

⟨*ML*⟩

definition *inv-into* :: *'a set => ('a => 'b) => ('b => 'a)* **where**
inv-into A f == %x. SOME y. y : A & f y = x

abbreviation *inv* :: *('a => 'b) => ('b => 'a)* **where**
inv == inv-into UNIV

40.2 Hilbert's Epsilon-operator

Easier to apply than *someI* if the witness comes from an existential formula

lemma *someI-ex* [*elim?*]: *∃ x. P x ==> P (SOME x. P x)*
⟨*proof*⟩

Easier to apply than *someI* because the conclusion has only one occurrence of *P*.

lemma *someI2*: *[| P a; !!x. P x ==> Q x |] ==> Q (SOME x. P x)*
⟨*proof*⟩

Easier to apply than *someI2* if the witness comes from an existential formula

lemma *someI2-ex*: *[| ∃ a. P a; !!x. P x ==> Q x |] ==> Q (SOME x. P x)*
⟨*proof*⟩

lemma *some-equality* [*intro*]:
[| P a; !!x. P x ==> x=a |] ==> (SOME x. P x) = a
⟨*proof*⟩

lemma *some1-equality*: *[| EX!x. P x; P a |] ==> (SOME x. P x) = a*
⟨*proof*⟩

lemma *some-eq-ex*: *P (SOME x. P x) = (∃ x. P x)*
⟨*proof*⟩

lemma *some-eq-trivial* [*simp*]: *(SOME y. y=x) = x*
⟨*proof*⟩

lemma *some-sym-eq-trivial* [*simp*]: $(SOME\ y.\ x{=}y) = x$
⟨*proof*⟩

40.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

lemma *choice*: $\forall x.\ \exists y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall x.\ Q\ x\ (f\ x)$
⟨*proof*⟩

lemma *bchoice*: $\forall x{\in}S.\ \exists y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall x{\in}S.\ Q\ x\ (f\ x)$
⟨*proof*⟩

40.4 Function Inverse

lemma *inv-def*: $inv\ f = (\%y.\ SOME\ x.\ f\ x = y)$
⟨*proof*⟩

lemma *inv-into-into*: $x : f\ `\ A \Longrightarrow inv\text{-}into\ A\ f\ x : A$
⟨*proof*⟩

lemma *inv-id* [*simp*]: $inv\ id = id$
⟨*proof*⟩

lemma *inv-into-f-f* [*simp*]:
$[|\ inj\text{-}on\ f\ A;\ \ x : A\ |] \Longrightarrow inv\text{-}into\ A\ f\ (f\ x) = x$
⟨*proof*⟩

lemma *inv-f-f*: $inj\ f \Longrightarrow inv\ f\ (f\ x) = x$
⟨*proof*⟩

lemma *f-inv-into-f*: $y : f`A \Longrightarrow f\ (inv\text{-}into\ A\ f\ y) = y$
⟨*proof*⟩

lemma *inv-into-f-eq*: $[|\ inj\text{-}on\ f\ A;\ x : A;\ f\ x = y\ |] \Longrightarrow inv\text{-}into\ A\ f\ y = x$
⟨*proof*⟩

lemma *inv-f-eq*: $[|\ inj\ f;\ f\ x = y\ |] \Longrightarrow inv\ f\ y = x$
⟨*proof*⟩

lemma *inj-imp-inv-eq*: $[|\ inj\ f;\ ALL\ x.\ f(g\ x) = x\ |] \Longrightarrow inv\ f = g$
⟨*proof*⟩

But is it useful?

lemma *inj-transfer*:
assumes *injf*: $inj\ f$ **and** *minor*: $!!y.\ y \in range(f) \Longrightarrow P(inv\ f\ y)$
shows $P\ x$

⟨*proof*⟩

lemma *inj-iff*: $(inj\ f) = (inv\ f\ o\ f = id)$
⟨*proof*⟩

lemma *inv-o-cancel*[*simp*]: $inj\ f \Longrightarrow inv\ f\ o\ f = id$
⟨*proof*⟩

lemma *o-inv-o-cancel*[*simp*]: $inj\ f \Longrightarrow g\ o\ inv\ f\ o\ f = g$
⟨*proof*⟩

lemma *inv-into-image-cancel*[*simp*]:
$inj\text{-}on\ f\ A \Longrightarrow S <= A \Longrightarrow inv\text{-}into\ A\ f\ `\ f\ `\ S = S$
⟨*proof*⟩

lemma *inj-imp-surj-inv*: $inj\ f \Longrightarrow surj\ (inv\ f)$
⟨*proof*⟩

lemma *surj-f-inv-f*: $surj\ f \Longrightarrow f(inv\ f\ y) = y$
⟨*proof*⟩

lemma *inv-into-injective*:
assumes *eq*: $inv\text{-}into\ A\ f\ x = inv\text{-}into\ A\ f\ y$
and *x*: $x: f`A$
and *y*: $y: f`A$
shows $x=y$
⟨*proof*⟩

lemma *inj-on-inv-into*: $B <= f`A \Longrightarrow inj\text{-}on\ (inv\text{-}into\ A\ f)\ B$
⟨*proof*⟩

lemma *bij-betw-inv-into*: $bij\text{-}betw\ f\ A\ B \Longrightarrow bij\text{-}betw\ (inv\text{-}into\ A\ f)\ B\ A$
⟨*proof*⟩

lemma *surj-imp-inj-inv*: $surj\ f \Longrightarrow inj\ (inv\ f)$
⟨*proof*⟩

lemma *surj-iff*: $(surj\ f) = (f\ o\ inv\ f = id)$
⟨*proof*⟩

lemma *surj-imp-inv-eq*: $[|\ surj\ f; \forall x.\ g(f\ x) = x\ |] \Longrightarrow inv\ f = g$
⟨*proof*⟩

lemma *bij-imp-bij-inv*: $bij\ f \Longrightarrow bij\ (inv\ f)$
⟨*proof*⟩

lemma *inv-equality*: $[|\ !!x.\ g\ (f\ x) = x;\ \ !!y.\ f\ (g\ y) = y\ |] \Longrightarrow inv\ f = g$
⟨*proof*⟩

lemma *inv-inv-eq*: *bij f ==> inv (inv f) = f*
⟨*proof*⟩

lemma *inv-into-comp*:
[| inj-on f (g ' A); inj-on g A; x : f ' g ' A |] ==>
inv-into A (f o g) x = (inv-into A g o inv-into (g ' A) f) x
⟨*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*⟩

lemma *finite-fun-UNIVD1*:
assumes *fin*: *finite (UNIV :: ('a ⇒ 'b) set)*
and *card*: *card (UNIV :: 'b set) ≠ Suc 0*
shows *finite (UNIV :: 'a set)*
⟨*proof*⟩

40.5 Other Consequences of Hilbert's Epsilon

Hilbert's Epsilon and the *split* Operator

Looping simprule

lemma *split-paired-Eps*: *(SOME x. P x) = (SOME (a,b). P(a,b))*
⟨*proof*⟩

lemma *Eps-split*: *Eps (split P) = (SOME xy. P (fst xy) (snd xy))*
⟨*proof*⟩

lemma *Eps-split-eq* [*simp*]: *(@(x',y'). x = x' & y = y') = (x,y)*
⟨*proof*⟩

A relation is wellfounded iff it has no infinite descending chain

lemma *wf-iff-no-infinite-down-chain*:
wf r = (~(∃*f*. ∀*i*. (*f*(*Suc i*),*f i*) ∈ *r*))
⟨*proof*⟩

lemma *wf-no-infinite-down-chainE*:
assumes *wf r* **obtains** *k* **where** (*f* (*Suc k*), *f k*) ∉ *r*
⟨*proof*⟩

A dynamically-scoped fact for TFL

lemma *tfl-some*: ∀*P x*. *P x* −−> *P* (*Eps P*)
⟨*proof*⟩

40.6 Least value operator

definition
LeastM :: ['*a* => '*b*::*ord*, '*a* => *bool*] => '*a* **where**
LeastM m P == *SOME x*. *P x* & (∀*y*. *P y* −−> *m x* <= *m y*)

syntax
-*LeastM* :: [*pttrn*, '*a* => '*b*::*ord*, *bool*] => '*a* (*LEAST* - *WRT* -. - [*0*, *4*, *10*] *10*)
translations
LEAST x WRT m. *P* == *CONST LeastM m* (%*x*. *P*)

lemma *LeastMI2*:
P x ==> (!!*y*. *P y* ==> *m x* <= *m y*)
==> (!!*x*. *P x* ==> ∀*y*. *P y* −−> *m x* ≤ *m y* ==> *Q x*)
==> *Q* (*LeastM m P*)
⟨*proof*⟩

lemma *LeastM-equality*:
P k ==> (!!*x*. *P x* ==> *m k* <= *m x*)
==> *m* (*LEAST x WRT m*. *P x*) = (*m k*::'*a*::*order*)
⟨*proof*⟩

lemma *wf-linord-ex-has-least*:
wf r ==> ∀*x y*. ((*x*,*y*):*r*^+) = ((*y*,*x*)~:*r*^*) ==> *P k*
==> ∃*x*. *P x* & (!*y*. *P y* −−> (*m x*,*m y*):*r*^*)
⟨*proof*⟩

lemma *ex-has-least-nat*:
P k ==> ∃*x*. *P x* & (∀*y*. *P y* −−> *m x* <= (*m y*::*nat*))
⟨*proof*⟩

lemma *LeastM-nat-lemma*:
P k ==> *P* (*LeastM m P*) & (∀*y*. *P y* −−> *m* (*LeastM m P*) <= (*m y*::*nat*))
⟨*proof*⟩

lemmas *LeastM-natI* = *LeastM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *LeastM-nat-le*: $P\ x\ ==>\ m\ (LeastM\ m\ P) <= (m\ x::nat)$
⟨*proof*⟩

40.7 Greatest value operator

definition
$GreatestM :: ['a => 'b::ord, 'a => bool] => 'a$ **where**
$GreatestM\ m\ P == SOME\ x.\ P\ x\ \&\ (\forall y.\ P\ y\ -->\ m\ y <= m\ x)$

definition
$Greatest :: ('a::ord => bool) => 'a$ (**binder** $GREATEST\ \ 10$) **where**
$Greatest == GreatestM\ (\%x.\ x)$

syntax
$-GreatestM :: [pttrn, 'a => 'b::ord, bool] => 'a$
$(GREATEST\ -\ WRT\ -.\ -\ [0, 4, 10]\ 10)$
translations
$GREATEST\ x\ WRT\ m.\ P == CONST\ GreatestM\ m\ (\%x.\ P)$

lemma *GreatestMI2*:
$P\ x\ ==>\ (!!y.\ P\ y\ ==>\ m\ y <= m\ x)$
$==>\ (!!x.\ P\ x\ ==>\ \forall y.\ P\ y\ -->\ m\ y \leq m\ x\ ==>\ Q\ x)$
$==>\ Q\ (GreatestM\ m\ P)$
⟨*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*:
$P\ k\ ==>\ \forall x.\ P\ x\ -->\ (\exists y.\ P\ y\ \&\ \sim\ ((m\ y::nat) <= m\ x))$
$==>\ \exists y.\ P\ y\ \&\ \sim\ (m\ y < m\ k + n)$
⟨*proof*⟩

lemma *ex-has-greatest-nat*:
$P\ k\ ==>\ \forall y.\ P\ y\ -->\ m\ y < b$
$==>\ \exists x.\ P\ x\ \&\ (\forall y.\ P\ y\ -->\ (m\ y::nat) <= m\ x)$
⟨*proof*⟩

lemma *GreatestM-nat-lemma*:
$P\ k\ ==>\ \forall y.\ P\ y\ -->\ m\ y < b$
$==>\ P\ (GreatestM\ m\ P)\ \&\ (\forall y.\ P\ y\ -->\ (m\ y::nat) <= m\ (GreatestM\ m\ P))$

⟨*proof*⟩

lemmas *GreatestM-natI* = *GreatestM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *GreatestM-nat-le*:
P x ==> ∀ *y*. *P y* −−> *m y* < *b*
==> (*m x*::*nat*) <= *m* (*GreatestM m P*)
⟨*proof*⟩

Specialization to *GREATEST*.

lemma *GreatestI*: *P* (*k*::*nat*) ==> ∀ *y*. *P y* −−> *y* < *b* ==> *P* (*GREATEST x*. *P x*)
⟨*proof*⟩

lemma *Greatest-le*:
P x ==> ∀ *y*. *P y* −−> *y* < *b* ==> (*x*::*nat*) <= (*GREATEST x*. *P x*)
⟨*proof*⟩

40.8 The Meson proof procedure

40.8.1 Negation Normal Form

de Morgan laws

lemma *meson-not-conjD*: ~(*P*&*Q*) ==> ~*P* | ~*Q*
and *meson-not-disjD*: ~(*P*|*Q*) ==> ~*P* & ~*Q*
and *meson-not-notD*: ~~*P* ==> *P*
and *meson-not-allD*: !!*P*. ~(∀ *x*. *P*(*x*)) ==> ∃ *x*. ~*P*(*x*)
and *meson-not-exD*: !!*P*. ~(∃ *x*. *P*(*x*)) ==> ∀ *x*. ~*P*(*x*)
⟨*proof*⟩

Removal of −−> and <−> (positive and negative occurrences)

lemma *meson-imp-to-disjD*: *P*−−>*Q* ==> ~*P* | *Q*
and *meson-not-impD*: ~(*P*−−>*Q*) ==> *P* & ~*Q*
and *meson-iff-to-disjD*: *P*=*Q* ==> (~*P* | *Q*) & (~*Q* | *P*)
and *meson-not-iffD*: ~(*P*=*Q*) ==> (*P* | *Q*) & (~*P* | ~*Q*)
— Much more efficient than $P \wedge \neg Q \vee Q \wedge \neg P$ for computing CNF
and *meson-not-refl-disj-D*: *x* ~= *x* | *P* ==> *P*
⟨*proof*⟩

40.8.2 Pulling out the existential quantifiers

Conjunction

lemma *meson-conj-exD1*: !!*P Q*. (∃ *x*. *P*(*x*)) & *Q* ==> ∃ *x*. *P*(*x*) & *Q*
and *meson-conj-exD2*: !!*P Q*. *P* & (∃ *x*. *Q*(*x*)) ==> ∃ *x*. *P* & *Q*(*x*)
⟨*proof*⟩

Disjunction

lemma *meson-disj-exD*: $!!P\ Q.\ (\exists x.\ P(x)) \mid (\exists x.\ Q(x)) ==> \exists x.\ P(x) \mid Q(x)$
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and *meson-disj-exD1*: $!!P\ Q.\ (\exists x.\ P(x)) \mid Q ==> \exists x.\ P(x) \mid Q$
and *meson-disj-exD2*: $!!P\ Q.\ P \mid (\exists x.\ Q(x)) ==> \exists x.\ P \mid Q(x)$
⟨*proof*⟩

40.8.3 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*⟩

40.9 Lemmas for Meson, the Model Elimination Procedure

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

lemma *make-neg-rule*: $\sim P|Q ==> ((\sim P==>P) ==> Q)$
⟨*proof*⟩

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: $\sim P|Q ==> (P ==> Q)$
⟨*proof*⟩

P should be a literal

lemma *make-pos-rule*: $P|Q ==> ((P==>\sim P) ==> Q)$
⟨*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'*: $[|P|Q;\ \sim P|] ==> Q$
⟨*proof*⟩

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: $\sim P ==> ((\sim P==>P) ==> False)$
⟨*proof*⟩

lemma *make-pos-goal*: $P ==> ((P==>\sim P) ==> False)$
⟨*proof*⟩

40.9.1 Lemmas for Forward Proof

There is a similarity to congruence rules

lemma *conj-forward*: $[|\ P'\&Q';\ \ P' ==> P;\ \ Q' ==> Q\ |] ==> P\&Q$
⟨*proof*⟩

lemma *disj-forward*: $[|\ P'|Q';\ \ P' ==> P;\ \ Q' ==> Q\ |] ==> P|Q$
⟨*proof*⟩

lemma *disj-forward2*:
$[|\ P'|Q';\ \ P' ==> P;\ \ [|\ Q';\ P==>False\ |] ==> Q\ |] ==> P|Q$
⟨*proof*⟩

lemma *all-forward*: $[|\ \forall x.\ P'(x);\ \ !!x.\ P'(x) ==> P(x)\ |] ==> \forall x.\ P(x)$
⟨*proof*⟩

lemma *ex-forward*: $[|\ \exists x.\ P'(x);\ \ !!x.\ P'(x) ==> P(x)\ |] ==> \exists x.\ P(x)$
⟨*proof*⟩

40.10 Meson package

⟨*ML*⟩

40.11 Specification package – Hilbertized version

lemma *exE-some*: $[|\ Ex\ P\ ;\ c == Eps\ P\ |] ==> P\ c$
⟨*proof*⟩

⟨*ML*⟩

end

41 Sledgehammer: Sledgehammer: Isabelle–ATP Linkup

theory *Sledgehammer*
imports *Plain Hilbert-Choice*
uses
~~/*src*/*Tools*/*Metis*/*metis.ML*
Tools/*Sledgehammer*/*sledgehammer-util.ML*
(*Tools*/*Sledgehammer*/*sledgehammer-fol-clause.ML*)
(*Tools*/*Sledgehammer*/*sledgehammer-fact-preprocessor.ML*)
(*Tools*/*Sledgehammer*/*sledgehammer-hol-clause.ML*)
(*Tools*/*Sledgehammer*/*sledgehammer-proof-reconstruct.ML*)
(*Tools*/*Sledgehammer*/*sledgehammer-fact-filter.ML*)

(*Tools/ATP-Manager/atp-manager.ML*)
(*Tools/ATP-Manager/atp-systems.ML*)
(*Tools/Sledgehammer/sledgehammer-fact-minimizer.ML*)
(*Tools/Sledgehammer/sledgehammer-isar.ML*)
(*Tools/Sledgehammer/meson-tactic.ML*)
(*Tools/Sledgehammer/metis-tactics.ML*)
begin

definition *COMBI* :: $'a \Rightarrow 'a$ **where**
[*no-atp*]: *COMBI P* $\equiv$ *P*

definition *COMBK* :: $'a \Rightarrow 'b \Rightarrow 'a$ **where**
[*no-atp*]: *COMBK P Q* $\equiv$ *P*

definition *COMBB* :: $('b => 'c) \Rightarrow ('a => 'b) \Rightarrow 'a \Rightarrow 'c$ **where** [*no-atp*]:
COMBB P Q R $\equiv$ *P* (*Q R*)

definition *COMBC* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
[*no-atp*]: *COMBC P Q R* $\equiv$ *P R Q*

definition *COMBS* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ **where**
[*no-atp*]: *COMBS P Q R* $\equiv$ *P R* (*Q R*)

definition *fequal* :: $'a \Rightarrow 'a \Rightarrow$ *bool* **where** [*no-atp*]:
fequal X Y $\equiv$ (*X* = *Y*)

lemma *fequal-imp-equal* [*no-atp*]: *fequal X Y* $\Longrightarrow$ *X* = *Y*
⟨*proof*⟩

lemma *equal-imp-fequal* [*no-atp*]: *X* = *Y* $\Longrightarrow$ *fequal X Y*
⟨*proof*⟩

These two represent the equivalence between Boolean equality and iff. They can't be converted to clauses automatically, as the iff would be expanded...

lemma *iff-positive*: *P* $\vee$ *Q* $\vee$ *P* = *Q*
⟨*proof*⟩

lemma *iff-negative*: $\neg$ *P* $\vee$ $\neg$ *Q* $\vee$ *P* = *Q*
⟨*proof*⟩

Theorems for translation to combinators

lemma *abs-S* [*no-atp*]: $\lambda x.$ (*f x*) (*g x*) $\equiv$ *COMBS f g*
⟨*proof*⟩

lemma *abs-I* [*no-atp*]: $\lambda x.$ *x* $\equiv$ *COMBI*
⟨*proof*⟩

lemma *abs-K* [*no-atp*]: $\lambda x.$ *y* $\equiv$ *COMBK y*
⟨*proof*⟩

lemma *abs-B* [*no-atp*]: $\lambda x.\ a\ (g\ x) \equiv COMBB\ a\ g$
⟨*proof*⟩

lemma *abs-C* [*no-atp*]: $\lambda x.\ (f\ x)\ b \equiv COMBC\ f\ b$
⟨*proof*⟩

41.1 Setup of external ATPs

⟨*ML*⟩

41.2 The MESON prover

⟨*ML*⟩

41.3 The Metis prover

⟨*ML*⟩

end

42 Recdef: TFL: recursive function definitions

theory *Recdef*
imports *FunDef Plain*
uses
(*Tools/TFL/casesplit.ML*)
(*Tools/TFL/utils.ML*)
(*Tools/TFL/usyntax.ML*)
(*Tools/TFL/dcterm.ML*)
(*Tools/TFL/thms.ML*)
(*Tools/TFL/rules.ML*)
(*Tools/TFL/thry.ML*)
(*Tools/TFL/tfl.ML*)
(*Tools/TFL/post.ML*)
(*Tools/recdef.ML*)
begin

inductive
wfrec-rel :: $('a * 'a)\ set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
for R :: $('a * 'a)\ set$
and F :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where
wfrecI: $ALL\ z.\ (z, x) : R \longrightarrow wfrec\text{-}rel\ R\ F\ z\ (g\ z) \Longrightarrow$
wfrec-rel $R\ F\ x\ (F\ g\ x)$

definition
cut :: $('a \Rightarrow 'b) \Rightarrow ('a * 'a)set \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b$ **where**

cut f r x == (%y. if (y,x):r then f y else undefined)

definition
adm-wf :: *('a * 'a) set => (('a => 'b) => 'a => 'b) => bool* **where**
adm-wf R F == ALL f g x.
(ALL z. (z, x) : R --> f z = g z) --> F f x = F g x

definition
wfrec :: *('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b* **where**
[*code del*]: *wfrec R F == %x. THE y. wfrec-rel R (%f x. F (cut f R x) x) x y*

42.1 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. wfrec-rel R F x y*
⟨*proof*⟩

lemma *adm-lemma*: *adm-wf R (%f x. F (cut f R x) x)*
⟨*proof*⟩

lemma *wfrec*: *wf(r) ==> wfrec r H a = H (cut (wfrec r H) r a) a*
⟨*proof*⟩

* This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-wfrec*: *[| f==wfrec r H; wf(r) |] ==> f(a) = H (cut f r a) a*
⟨*proof*⟩

lemma *tfl-wf-induct*: *ALL R. wf R -->*
(ALL P. (ALL x. (ALL y. (y,x):R --> P y) --> P x) --> (ALL x. P x))
⟨*proof*⟩

lemma *tfl-cut-apply*: *ALL f R. (x,a):R --> (cut f R a)(x) = f(x)*
⟨*proof*⟩

lemma *tfl-wfrec*:
ALL M R f. (f=wfrec R M) --> wf R --> (ALL x. f x = M (cut f R x) x)
⟨*proof*⟩

lemma *tfl-eq-True*: $(x = True) \dashrightarrow x$
⟨*proof*⟩

lemma *tfl-rev-eq-mp*: $(x = y) \dashrightarrow y \dashrightarrow x$
⟨*proof*⟩

lemma *tfl-simp-thm*: $(x \dashrightarrow y) \dashrightarrow (x = x') \dashrightarrow (x' \dashrightarrow y)$
⟨*proof*⟩

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
⟨*proof*⟩

lemma *tfl-imp-trans*: $(A \dashrightarrow B) \Longrightarrow (B \dashrightarrow C) \Longrightarrow (A \dashrightarrow C)$
⟨*proof*⟩

lemma *tfl-disj-assoc*: $(a \vee b) \vee c == a \vee (b \vee c)$
⟨*proof*⟩

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \dashrightarrow R \Longrightarrow Q \dashrightarrow R \Longrightarrow R$
⟨*proof*⟩

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \dashrightarrow Q \Longrightarrow Q$
⟨*proof*⟩

⟨*ML*⟩

Wellfoundedness of *same-fst*

definition
same-fst :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow ('b * 'b)set) \Rightarrow (('a*'b)*('a*'b))set$
where
same-fst P R == $\{((x',y'),(x,y))\ .\ x'=x\ \&\ P\ x\ \&\ (y',y) : R\ x\}$
— For *rec-def* declarations where the first n parameters stay unchanged in the recursive call.

lemma *same-fstI* [*intro*!]:
$[|\ P\ x;\ (y',y) : R\ x\ |] \Longrightarrow ((x,y'),(x,y)) : same\text{-}fst\ P\ R$
⟨*proof*⟩

lemma *wf-same-fst*:
assumes *prem*: $(!!x.\ P\ x \Longrightarrow wf(R\ x))$
shows $wf(same\text{-}fst\ P\ R)$
⟨*proof*⟩

Rule setup

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def

less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong let-cong image-cong INT-cong UN-cong bex-cong ball-cong imp-cong

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-pred-nat
wf-same-fst
wf-empty

end

43 List: The datatype of finite lists

theory *List*
imports *Plain Presburger Sledgehammer Recdef*
uses (*Tools/list-code.ML*)
begin

datatype *'a list* =
  Nil ([])
| *Cons 'a 'a list* (**infixr** # *65*)

syntax
— list Enumeration
-list :: *args* => *'a list* ([(-)])

translations
[*x*, *xs*] == *x*#[*xs*]
[*x*] == *x*#[]

43.1 Basic list processing functions

primrec
hd :: *'a list* ⇒ *'a* **where**
hd (*x* # *xs*) = *x*

primrec
tl :: *'a list* ⇒ *'a list* **where**
tl [] = []
| *tl* (*x* # *xs*) = *xs*

primrec

last :: *'a list* ⇒ *'a* **where**
last (*x* # *xs*) = (*if xs* = [] *then x else last xs*)

primrec
butlast :: *'a list* ⇒ *'a list* **where**
butlast []= []
| *butlast* (*x* # *xs*) = (*if xs* = [] *then* [] *else x* # *butlast xs*)

primrec
set :: *'a list* ⇒ *'a set* **where**
set [] = {}
| *set* (*x* # *xs*) = *insert x* (*set xs*)

primrec
map :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list* **where**
map f [] = []
| *map f* (*x* # *xs*) = *f x* # *map f xs*

primrec
append :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixr** @ *65*) **where**
append-Nil:[] @ *ys* = *ys*
| *append-Cons*: (*x*#*xs*) @ *ys* = *x* # *xs* @ *ys*

primrec
rev :: *'a list* ⇒ *'a list* **where**
rev [] = []
| *rev* (*x* # *xs*) = *rev xs* @ [*x*]

primrec
filter:: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list* **where**
filter P [] = []
| *filter P* (*x* # *xs*) = (*if P x then x* # *filter P xs else filter P xs*)

syntax
— Special syntax for filter
-*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list* ((*1*[-<−-./ -]))

translations
[*x*<−*xs* . *P*]== *CONST filter* (%*x*. *P*) *xs*

syntax (*xsymbols*)
-*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list*((*1*[-←- ./ -]))
syntax (*HTML* **output**)
-*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list*((*1*[-←- ./ -]))

primrec
foldl :: (*'b* ⇒ *'a* ⇒ *'b*) ⇒ *'b* ⇒ *'a list* ⇒ *'b* **where**
foldl-Nil: *foldl f a* [] = *a*
| *foldl-Cons*: *foldl f a* (*x* # *xs*) = *foldl f* (*f a x*) *xs*

primrec
foldr :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b$ **where**
foldr f [] *a* = *a*
| *foldr f* (*x* # *xs*) *a* = *f x* (*foldr f xs a*)

primrec
concat:: $'a\ list\ list \Rightarrow 'a\ list$ **where**
concat [] = []
| *concat* (*x* # *xs*) = *x* @ *concat xs*

primrec (**in** *monoid-add*)
listsum :: $'a\ list \Rightarrow 'a$ **where**
listsum [] = *0*
| *listsum* (*x* # *xs*) = *x* + *listsum xs*

primrec
drop:: *nat* $\Rightarrow$ $'a\ list \Rightarrow 'a\ list$ **where**
drop-Nil: *drop n* [] = []
| *drop-Cons*: *drop n* (*x* # *xs*) = (*case n of 0* $\Rightarrow$ *x* # *xs* | *Suc m* $\Rightarrow$ *drop m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
take:: *nat* $\Rightarrow$ $'a\ list \Rightarrow 'a\ list$ **where**
take-Nil:*take n* [] = []
| *take-Cons*: *take n* (*x* # *xs*) = (*case n of 0* $\Rightarrow$ [] | *Suc m* $\Rightarrow$ *x* # *take m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
nth :: $'a\ list \Rightarrow nat \Rightarrow 'a$ (**infixl** ! *100*) **where**
nth-Cons: (*x* # *xs*) ! *n* = (*case n of 0* $\Rightarrow$ *x* | *Suc k* $\Rightarrow$ *xs* ! *k*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
list-update :: $'a\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ list$ **where**
list-update [] *i v* = []
| *list-update* (*x* # *xs*) *i v* = (*case i of 0* $\Rightarrow$ *v* # *xs* | *Suc j* $\Rightarrow$ *x* # *list-update xs j v*)

nonterminals *lupdbinds lupdbind*

syntax
-lupdbind:: [$'a$, $'a$] => *lupdbind* ((*2*- :=/ -))
:: *lupdbind* => *lupdbinds* (-)
-lupdbinds :: [*lupdbind*, *lupdbinds*] => *lupdbinds* (-,/ -)
-LUpdate :: [$'a$, *lupdbinds*] => $'a$ (-/[(-)] [*900*,*0*] *900*)

translations
-LUpdate xs (-lupdbinds b bs) == -LUpdate (-LUpdate xs b) bs
xs[i:=x] == CONST list-update xs i x

primrec
takeWhile :: *('a ⇒ bool) ⇒ 'a list ⇒ 'a list* **where**
takeWhile P [] = []
| *takeWhile P (x # xs) = (if P x then x # takeWhile P xs else [])*

primrec
dropWhile :: *('a ⇒ bool) ⇒ 'a list ⇒ 'a list* **where**
dropWhile P [] = []
| *dropWhile P (x # xs) = (if P x then dropWhile P xs else x # xs)*

primrec
zip :: *'a list ⇒ 'b list ⇒ ('a × 'b) list* **where**
zip xs [] = []
| *zip-Cons*: *zip xs (y # ys) = (case xs of [] => [] | z # zs => (z, y) # zip zs ys)*
— Warning: simpset does not contain this definition, but separate theorems for *xs = []* and *xs = z # zs*

primrec
upt :: *nat ⇒ nat ⇒ nat list* (*(1[-..</-']*)) **where**
upt-0: *[i..<0] = []*
| *upt-Suc*: *[i..<(Suc j)] = (if i <= j then [i..<j] @ [j] else [])*

primrec
distinct :: *'a list ⇒ bool* **where**
distinct [] ⟷ True
| *distinct (x # xs) ⟷ x ∉ set xs ∧ distinct xs*

primrec
remdups :: *'a list ⇒ 'a list* **where**
remdups [] = []
| *remdups (x # xs) = (if x ∈ set xs then remdups xs else x # remdups xs)*

definition
insert :: *'a ⇒ 'a list ⇒ 'a list* **where**
insert x xs = (if x ∈ set xs then xs else x # xs)

hide-const (**open**) *insert*
hide-fact (**open**) *insert-def*

primrec
remove1 :: *'a ⇒ 'a list ⇒ 'a list* **where**
remove1 x [] = []
| *remove1 x (y # xs) = (if x = y then xs else y # remove1 x xs)*

primrec
removeAll :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
removeAll x [] = []
| *removeAll x* (*y* # *xs*) = (*if x* = *y then removeAll x xs else y* # *removeAll x xs*)

primrec
replicate :: *nat* ⇒ *'a* ⇒ *'a list* **where**
replicate-0: *replicate 0 x* = []
| *replicate-Suc*: *replicate* (*Suc n*) *x* = *x* # *replicate n x*

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

abbreviation
length :: *'a list* ⇒ *nat* **where**
length ≡ *size*

definition
rotate1 :: *'a list* ⇒ *'a list* **where**
rotate1 xs = (*case xs of* [] ⇒ [] | *x*#*xs* ⇒ *xs* @ [*x*])

definition
rotate :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
rotate n = *rotate1* ^^ *n*

definition
list-all2 :: (*'a* => *'b* => *bool*) => *'a list* => *'b list* => *bool* **where**
[*code del*]: *list-all2 P xs ys* =
(*length xs* = *length ys* ∧ (∀(*x*, *y*) ∈ *set* (*zip xs ys*). *P x y*))

definition
sublist :: *'a list* => *nat set* => *'a list* **where**
sublist xs A = *map fst* (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*0*..<*size xs*]))

primrec
splice :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
splice [] *ys* = *ys*
| *splice* (*x* # *xs*) *ys* = (*if ys* = [] *then x* # *xs else x* # *hd ys* # *splice xs* (*tl ys*))
— Warning: simpset does not contain the second eqn but a derived one.

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient implementations.

context *linorder*
begin

fun *sorted* :: *'a list* ⇒ *bool* **where**

$[a, b] @ [c, d] = [a, b, c, d]$
length $[a, b, c] = 3$
set $[a, b, c] = \{a, b, c\}$
map f $[a, b, c] = [f\ a, f\ b, f\ c]$
rev $[a, b, c] = [c, b, a]$
hd $[a, b, c, d] = a$
tl $[a, b, c, d] = [b, c, d]$
last $[a, b, c, d] = d$
butlast $[a, b, c, d] = [a, b, c]$
filter $(\lambda n{::}nat.\ n{<}2)\ [0,2,1] = [0,1]$
concat $[[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]$
foldl f x $[a, b, c] = f\ (f\ (f\ x\ a)\ b)\ c$
foldr f $[a, b, c]\ x = f\ a\ (f\ b\ (f\ c\ x))$
zip $[a, b, c]\ [x, y, z] = [(a, x), (b, y), (c, z)]$
zip $[a, b]\ [x, y, z] = [(a, x), (b, y)]$
splice $[a, b, c]\ [x, y, z] = [a, x, b, y, c, z]$
splice $[a, b, c, d]\ [x, y] = [a, x, b, y, c, d]$
take 2 $[a, b, c, d] = [a, b]$
take 6 $[a, b, c, d] = [a, b, c, d]$
drop 2 $[a, b, c, d] = [c, d]$
drop 6 $[a, b, c, d] = []$
takeWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [1, 2]$
dropWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [3, 0]$
distinct $[2, 0, 1]$
remdups $[2, 0, 2, 1, 2] = [0, 1, 2]$
List.insert 2 $[0, 1, 2] = [0, 1, 2]$
List.insert 3 $[0, 1, 2] = [3, 0, 1, 2]$
remove1 2 $[2, 0, 2, 1, 2] = [0, 2, 1, 2]$
removeAll 2 $[2, 0, 2, 1, 2] = [0, 1]$
$[a, b, c, d]\ !\ 2 = c$
$[a, b, c, d][2 := x] = [a, b, x, d]$
sublist $[a, b, c, d, e]\ \{0, 2, 3\} = [a, c, d]$
rotate1 $[a, b, c, d] = [b, c, d, a]$
rotate 3 $[a, b, c, d] = [d, a, b, c]$
replicate 4 a $= [a, a, a, a]$
$[2..{<}5] = [2, 3, 4]$
listsum $[1, 2, 3] = 6$

Figure 1: Characteristic examples

sorted [] ⟷ *True* |
sorted [*x*] ⟷ *True* |
sorted (*x*#*y*#*zs*) ⟷ *x* <= *y* ∧ *sorted* (*y*#*zs*)

primrec *insort-key* :: ('*b* ⇒ '*a*) ⇒ '*b* ⇒ '*b list* ⇒ '*b list* **where**
insort-key f x [] = [*x*] |
insort-key f x (*y*#*ys*) = (*if f x* ≤ *f y then* (*x*#*y*#*ys*) *else y*#(*insort-key f x ys*))

definition *sort-key* :: ('*b* ⇒ '*a*) ⇒ '*b list* ⇒ '*b list* **where**
sort-key f xs = *foldr* (*insort-key f*) *xs* []

abbreviation *sort* ≡ *sort-key* (λ*x*. *x*)
abbreviation *insort* ≡ *insort-key* (λ*x*. *x*)

definition *insort-insert* :: '*a* ⇒ '*a list* ⇒ '*a list* **where**
insort-insert x xs = (*if x* ∈ *set xs then xs else insort x xs*)

end

43.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: [(*x*,*y*). *x* ← *xs*, *y* ← *ys*, *x* ≠ *y*], the list of all pairs of distinct elements from *xs* and *ys*. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle's set comprehension): [*e*. *x* ← *xs*, ...] rather than `[e| x <- xs, ...]`.
The qualifiers after the dot are

generators *p* ← *xs*, where *p* is a pattern and *xs* an expression of list type,
or

guards *b*, where *b* is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of [*e*. *x* ← *xs*] is optmized to *map* (λ*x*. *e*) *xs*.
It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

nonterminals *lc-qual lc-quals*

syntax
-listcompr :: '*a* ⇒ *lc-qual* ⇒ *lc-quals* ⇒ '*a list* ([- . --)
-lc-gen :: '*a* ⇒ '*a list* ⇒ *lc-qual* (- <− -)
-lc-test :: *bool* ⇒ *lc-qual* (-)

-lc-end :: *lc-quals* ([])
-lc-quals :: *lc-qual* ⇒ *lc-quals* ⇒ *lc-quals* (, --)
-lc-abs :: *'a* => *'b list* => *'b list*

syntax (*xsymbols*)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)
syntax (*HTML* **output**)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)

⟨*ML*⟩

term [(*x*,*y*,*z*). *b*]
term [(*x*,*y*,*z*). *x*←*xs*]
term [*e x y*. *x*←*xs*, *y*←*ys*]
term [(*x*,*y*,*z*). *x*<*a*, *x*>*b*]
term [(*x*,*y*,*z*). *x*←*xs*, *x*>*b*]
term [(*x*,*y*,*z*). *x*<*a*, *x*←*xs*]
term [(*x*,*y*). *Cons True x* ← *xs*]
term [(*x*,*y*,*z*). *Cons x* [] ← *xs*]
term [(*x*,*y*,*z*). *x*<*a*, *x*>*b*, *x*=*d*]
term [(*x*,*y*,*z*). *x*<*a*, *x*>*b*, *y*←*ys*]
term [(*x*,*y*,*z*). *x*<*a*, *x*←*xs*,*y*>*b*]
term [(*x*,*y*,*z*). *x*<*a*, *x*←*xs*, *y*←*ys*]
term [(*x*,*y*,*z*). *x*←*xs*, *x*>*b*, *y*<*a*]
term [(*x*,*y*,*z*). *x*←*xs*, *x*>*b*, *y*←*ys*]
term [(*x*,*y*,*z*). *x*←*xs*, *y*←*ys*,*y*>*x*]
term [(*x*,*y*,*z*). *x*←*xs*, *y*←*ys*,*z*←*zs*]

43.1.2 [] and *op #*

lemma *not-Cons-self* [*simp*]:
xs ≠ *x # xs*
⟨*proof*⟩

lemmas *not-Cons-self2* [*simp*] = *not-Cons-self* [*symmetric*]

lemma *neq-Nil-conv*: (*xs* ≠ []) = (∃ *y ys*. *xs* = *y # ys*)
⟨*proof*⟩

lemma *length-induct*:
(⋀*xs*. ∀ *ys*. *length ys* < *length xs* ⟶ *P ys* ⟹ *P xs*) ⟹ *P xs*
⟨*proof*⟩

lemma *list-nonempty-induct* [*consumes 1*, *case-names single cons*]:
assumes *xs* ≠ []
assumes *single*: ⋀*x*. *P* [*x*]

assumes *cons*: $\bigwedge x\ xs.\ xs \neq [] \Longrightarrow P\ xs \Longrightarrow P\ (x\ \#\ xs)$
shows $P\ xs$
⟨*proof*⟩

43.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: $length\ (xs\ @\ ys) = length\ xs + length\ ys$
⟨*proof*⟩

lemma *length-map* [*simp*]: $length\ (map\ f\ xs) = length\ xs$
⟨*proof*⟩

lemma *length-rev* [*simp*]: $length\ (rev\ xs) = length\ xs$
⟨*proof*⟩

lemma *length-tl* [*simp*]: $length\ (tl\ xs) = length\ xs - 1$
⟨*proof*⟩

lemma *length-0-conv* [*iff*]: $(length\ xs = 0) = (xs = [])$
⟨*proof*⟩

lemma *length-greater-0-conv* [*iff*]: $(0 < length\ xs) = (xs \neq [])$
⟨*proof*⟩

lemma *length-pos-if-in-set*: $x : set\ xs \Longrightarrow length\ xs > 0$
⟨*proof*⟩

lemma *length-Suc-conv*:
$(length\ xs = Suc\ n) = (\exists y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
⟨*proof*⟩

lemma *Suc-length-conv*:
$(Suc\ n = length\ xs) = (\exists y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
⟨*proof*⟩

lemma *impossible-Cons*: $length\ xs <= length\ ys ==> xs = x\ \#\ ys = False$
⟨*proof*⟩

lemma *list-induct2* [*consumes 1*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow P\ []\ [] \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (x\#xs)\ (y\#ys))$
$\Longrightarrow P\ xs\ ys$
⟨*proof*⟩

lemma *list-induct3* [*consumes 2*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow P\ []\ []\ [] \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys\ z\ zs.\ length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow P\ xs\ ys\ zs$
$\Longrightarrow P\ (x\#xs)\ (y\#ys)\ (z\#zs))$

$\Longrightarrow P\ xs\ ys\ zs$
⟨*proof*⟩

lemma *list-induct4* [*consumes 3*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow length\ zs = length\ ws \Longrightarrow$
$P\ []\ []\ []\ [] \Longrightarrow (\bigwedge x\ xs\ y\ ys\ z\ zs\ w\ ws.\ length\ xs = length\ ys \Longrightarrow$
$length\ ys = length\ zs \Longrightarrow length\ zs = length\ ws \Longrightarrow P\ xs\ ys\ zs\ ws \Longrightarrow$
$P\ (x\#xs)\ (y\#ys)\ (z\#zs)\ (w\#ws)) \Longrightarrow P\ xs\ ys\ zs\ ws$
⟨*proof*⟩

lemma *list-induct2'*:
$[\![\ P\ []\ []; \bigwedge x\ xs.\ P\ (x\#xs)\ []; \bigwedge y\ ys.\ P\ []\ (y\#ys); \bigwedge x\ xs\ y\ ys.\ P\ xs\ ys \Longrightarrow P\ (x\#xs)\ (y\#ys)\ ]\!] \Longrightarrow P\ xs\ ys$
⟨*proof*⟩

lemma *neq-if-length-neq*: $length\ xs \neq length\ ys \Longrightarrow (xs = ys) == False$
⟨*proof*⟩

⟨*ML*⟩

43.1.4 @ − 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*]: $(xs\ @\ ys = []) = (xs = [] \wedge ys = [])$
⟨*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*, *no-atp*]:
$length\ xs = length\ ys \vee length\ us = length\ vs$
$==> (xs@us = ys@vs) = (xs=ys \wedge us=vs)$
⟨*proof*⟩

lemma *append-eq-append-conv2*: $(xs\ @\ ys = zs\ @\ ts) =$

(*EX us. xs = zs @ us & us @ ys = ts | xs @ us = zs & ys = us@ ts*)
⟨*proof*⟩

lemma *same-append-eq* [*iff*, *induct-simp*]: (*xs @ ys = xs @ zs*) = (*ys = zs*)
⟨*proof*⟩

lemma *append1-eq-conv* [*iff*]: (*xs @ [x] = ys @ [y]*) = (*xs = ys ∧ x = y*)
⟨*proof*⟩

lemma *append-same-eq* [*iff*, *induct-simp*]: (*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*,*no-atp*]: *xs ≠ [] ==> 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 ≠ [] ==> hd (xs @ ys) = hd xs*
⟨*proof*⟩

lemma *tl-append*: *tl (xs @ ys) = (case xs of [] => tl ys | z#zs => zs @ ys)*
⟨*proof*⟩

lemma *tl-append2* [*simp*]: *xs ≠ [] ==> tl (xs @ ys) = tl xs @ ys*
⟨*proof*⟩

lemma *Cons-eq-append-conv*: *x#xs = ys@zs =*
(ys = [] & x#xs = zs | (EX ys'. x#ys' = ys & xs = ys'@zs))
⟨*proof*⟩

lemma *append-eq-Cons-conv*: *(ys@zs = x#xs) =*
(ys = [] & zs = x#xs | (EX ys'. ys = x#ys' & ys'@zs = xs))
⟨*proof*⟩

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: *xs = ys ==> xs = [] @ ys*
⟨*proof*⟩

lemma *Cons-eq-appendI*:
[| x # xs1 = ys; xs = xs1 @ zs |] ==> x # xs = ys @ zs
⟨*proof*⟩

lemma *append-eq-appendI*:
[| *xs* @ *xs1* = *zs*; *ys* = *xs1* @ *us* |] ==> *xs* @ *ys* = *zs* @ *us*
⟨*proof*⟩

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

⟨*ML*⟩

43.1.5 *map*

lemma *map-ext*: (!!*x*. *x* : *set xs* --> *f x* = *g x*) ==> *map f xs* = *map g xs*
⟨*proof*⟩

lemma *map-ident* [*simp*]: *map* (λ*x*. *x*) = (λ*xs*. *xs*)
⟨*proof*⟩

lemma *map-append* [*simp*]: *map f* (*xs* @ *ys*) = *map f xs* @ *map f ys*
⟨*proof*⟩

lemma *map-map* [*simp*]: *map f* (*map g xs*) = *map* (*f* ∘ *g*) *xs*
⟨*proof*⟩

lemma *map-comp-map*[*simp*]: ((*map f*) *o* (*map g*)) = *map*(*f o g*)
⟨*proof*⟩

lemma *rev-map*: *rev* (*map f xs*) = *map f* (*rev xs*)
⟨*proof*⟩

lemma *map-eq-conv*[*simp*]: (*map f xs* = *map g xs*) = (!*x* : *set xs*. *f x* = *g x*)
⟨*proof*⟩

lemma *map-cong* [*fundef-cong*, *recdef-cong*]:
xs = *ys* ==> (!!*x*. *x* : *set ys* ==> *f x* = *g x*) ==> *map f xs* = *map g ys*
— a congruence rule for *map*
⟨*proof*⟩

lemma *map-is-Nil-conv* [*iff*]: (*map f xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *Nil-is-map-conv* [*iff*]: ([] = *map f xs*) = (*xs* = [])
⟨*proof*⟩

lemma *map-eq-Cons-conv*:
(*map f xs* = *y*#*ys*) = (∃ *z zs*. *xs* = *z*#*zs* ∧ *f z* = *y* ∧ *map f zs* = *ys*)
⟨*proof*⟩

lemma *Cons-eq-map-conv*:

$(x\#xs = map\ f\ ys) = (\exists z\ zs.\ ys = z\#zs \wedge x = f\ z \wedge xs = map\ f\ zs)$
⟨*proof*⟩

lemmas *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]
lemmas *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]
declare *map-eq-Cons-D* [*dest*!] *Cons-eq-map-D* [*dest*!]

lemma *ex-map-conv*:
(*EX xs. ys = map f xs*) = (*ALL y : set ys. EX x. y = f x*)
⟨*proof*⟩

lemma *map-eq-imp-length-eq*:
assumes *map f xs = map g ys*
shows *length xs = length ys*
⟨*proof*⟩

lemma *map-inj-on*:
[| *map f xs = map f ys*; *inj-on f* (*set xs Un set ys*) |]
==> *xs = ys*
⟨*proof*⟩

lemma *inj-on-map-eq-map*:
inj-on f (*set xs Un set ys*) ⟹ (*map f xs = map f ys*) = (*xs = ys*)
⟨*proof*⟩

lemma *map-injective*:
map f xs = map f ys ==> *inj f* ==> *xs = ys*
⟨*proof*⟩

lemma *inj-map-eq-map*[*simp*]: *inj f* ⟹ (*map f xs = map f ys*) = (*xs = ys*)
⟨*proof*⟩

lemma *inj-mapI*: *inj f* ==> *inj* (*map f*)
⟨*proof*⟩

lemma *inj-mapD*: *inj* (*map f*) ==> *inj f*
⟨*proof*⟩

lemma *inj-map*[*iff*]: *inj* (*map f*) = *inj f*
⟨*proof*⟩

lemma *inj-on-mapI*: *inj-on f* (⋃(*set* ' *A*)) ⟹ *inj-on* (*map f*) *A*
⟨*proof*⟩

lemma *map-idI*: (⋀*x*. *x* ∈ *set xs* ⟹ *f x = x*) ⟹ *map f xs = xs*
⟨*proof*⟩

lemma *map-fun-upd* [*simp*]: *y* ∉ *set xs* ⟹ *map* (*f*(*y*:=*v*)) *xs* = *map f xs*
⟨*proof*⟩

lemma *map-fst-zip*[*simp*]:
length xs = length ys ⟹ map fst (zip xs ys) = xs
⟨*proof*⟩

lemma *map-snd-zip*[*simp*]:
length xs = length ys ⟹ map snd (zip xs ys) = ys
⟨*proof*⟩

43.1.6 *rev*

lemma *rev-append* [*simp*]: *rev (xs @ ys) = rev ys @ rev xs*
⟨*proof*⟩

lemma *rev-rev-ident* [*simp*]: *rev (rev xs) = xs*
⟨*proof*⟩

lemma *rev-swap*: *(rev xs = ys) = (xs = rev ys)*
⟨*proof*⟩

lemma *rev-is-Nil-conv* [*iff*]: *(rev xs = []) = (xs = [])*
⟨*proof*⟩

lemma *Nil-is-rev-conv* [*iff*]: *([] = rev xs) = (xs = [])*
⟨*proof*⟩

lemma *rev-singleton-conv* [*simp*]: *(rev xs = [x]) = (xs = [x])*
⟨*proof*⟩

lemma *singleton-rev-conv* [*simp*]: *([x] = rev xs) = (xs = [x])*
⟨*proof*⟩

lemma *rev-is-rev-conv* [*iff*]: *(rev xs = rev ys) = (xs = ys)*
⟨*proof*⟩

lemma *inj-on-rev*[*iff*]: *inj-on rev A*
⟨*proof*⟩

lemma *rev-induct* [*case-names Nil snoc*]:
[| P []; !!x xs. P xs ==> P (xs @ [x]) |] ==> P xs
⟨*proof*⟩

lemma *rev-exhaust* [*case-names Nil snoc*]:
(xs = [] ==> P) ==>(!!ys y. xs = ys @ [y] ==> P) ==> P
⟨*proof*⟩

lemmas *rev-cases = rev-exhaust*

lemma *rev-eq-Cons-iff*[*iff*]: *(rev xs = y#ys) = (xs = rev ys @ [y])*

⟨*proof*⟩

43.1.7 *set*

lemma *finite-set* [*iff*]: *finite* (*set xs*)
⟨*proof*⟩

lemma *set-append* [*simp*]: *set* (*xs* @ *ys*) = (*set xs* ∪ *set ys*)
⟨*proof*⟩

lemma *hd-in-set*[*simp*]: *xs* ≠ [] ⟹ *hd xs* : *set xs*
⟨*proof*⟩

lemma *set-subset-Cons*: *set xs* ⊆ *set* (*x* # *xs*)
⟨*proof*⟩

lemma *set-ConsD*: *y* ∈ *set* (*x* # *xs*) ⟹ *y*=*x* ∨ *y* ∈ *set xs*
⟨*proof*⟩

lemma *set-empty* [*iff*]: (*set xs* = {}) = (*xs* = [])
⟨*proof*⟩

lemma *set-empty2*[*iff*]: ({} = *set xs*) = (*xs* = [])
⟨*proof*⟩

lemma *set-rev* [*simp*]: *set* (*rev xs*) = *set xs*
⟨*proof*⟩

lemma *set-map* [*simp*]: *set* (*map f xs*) = *f*‘(*set xs*)
⟨*proof*⟩

lemma *set-filter* [*simp*]: *set* (*filter P xs*) = {*x*. *x* : *set xs* ∧ *P x*}
⟨*proof*⟩

lemma *set-upt* [*simp*]: *set*[*i*..<*j*] = {*i*..<*j*}
⟨*proof*⟩

lemma *split-list*: *x* : *set xs* ⟹ ∃ *ys zs*. *xs* = *ys* @ *x* # *zs*
⟨*proof*⟩

lemma *in-set-conv-decomp*: *x* ∈ *set xs* ⟷ (∃ *ys zs*. *xs* = *ys* @ *x* # *zs*)
⟨*proof*⟩

lemma *split-list-first*: *x* : *set xs* ⟹ ∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set ys*
⟨*proof*⟩

lemma *in-set-conv-decomp-first*:
(*x* : *set xs*) = (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set ys*)

⟨*proof*⟩

lemma *split-list-last*: $x : set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ zs$
⟨*proof*⟩

lemma *in-set-conv-decomp-last*:
$(x : set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ zs)$
⟨*proof*⟩

lemma *split-list-prop*: $\exists\, x \in set\ xs.\ P\ x \Longrightarrow \exists\, ys\ x\ zs.\ xs = ys\ @\ x\ \#\ zs\ \&\ P\ x$
⟨*proof*⟩

lemma *split-list-propE*:
assumes $\exists\, x \in set\ xs.\ P\ x$
obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $P\ x$
⟨*proof*⟩

lemma *split-list-first-prop*:
$\exists\, x \in set\ xs.\ P\ x \Longrightarrow$
$\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, y \in set\ ys.\ \neg\ P\ y)$
⟨*proof*⟩

lemma *split-list-first-propE*:
assumes $\exists\, x \in set\ xs.\ P\ x$
obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $P\ x$ **and** $\forall\, y \in set\ ys.\ \neg\ P\ y$
⟨*proof*⟩

lemma *split-list-first-prop-iff*:
$(\exists\, x \in set\ xs.\ P\ x) \longleftrightarrow$
$(\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, y \in set\ ys.\ \neg\ P\ y))$
⟨*proof*⟩

lemma *split-list-last-prop*:
$\exists\, x \in set\ xs.\ P\ x \Longrightarrow$
$\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, z \in set\ zs.\ \neg\ P\ z)$
⟨*proof*⟩

lemma *split-list-last-propE*:
assumes $\exists\, x \in set\ xs.\ P\ x$
obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $P\ x$ **and** $\forall\, z \in set\ zs.\ \neg\ P\ z$
⟨*proof*⟩

lemma *split-list-last-prop-iff*:
$(\exists\, x \in set\ xs.\ P\ x) \longleftrightarrow$
$(\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, z \in set\ zs.\ \neg\ P\ z))$
⟨*proof*⟩

lemma *finite-list*: *finite A ==> EX xs. set xs = A*
⟨*proof*⟩

lemma *card-length*: *card* (*set xs*) ≤ *length xs*
⟨*proof*⟩

lemma *set-minus-filter-out*:
set xs − {*y*} = *set* (*filter* (λ*x*. ¬ (*x* = *y*)) *xs*)
⟨*proof*⟩

43.1.8 *filter*

lemma *filter-append* [*simp*]: *filter P* (*xs* @ *ys*) = *filter P xs* @ *filter P ys*
⟨*proof*⟩

lemma *rev-filter*: *rev* (*filter P xs*) = *filter P* (*rev xs*)
⟨*proof*⟩

lemma *filter-filter* [*simp*]: *filter P* (*filter Q xs*) = *filter* (λ*x*. *Q x* ∧ *P x*) *xs*
⟨*proof*⟩

lemma *length-filter-le* [*simp*]: *length* (*filter P xs*) ≤ *length xs*
⟨*proof*⟩

lemma *sum-length-filter-compl*:
length(*filter P xs*) + *length*(*filter* (%*x*. ~*P x*) *xs*) = *length xs*
⟨*proof*⟩

lemma *filter-True* [*simp*]: ∀ *x* ∈ *set xs*. *P x* ==> *filter P xs* = *xs*
⟨*proof*⟩

lemma *filter-False* [*simp*]: ∀ *x* ∈ *set xs*. ¬ *P x* ==> *filter P xs* = []
⟨*proof*⟩

lemma *filter-empty-conv*: (*filter P xs* = []) = (∀ *x*∈*set xs*. ¬ *P x*)
⟨*proof*⟩

lemma *filter-id-conv*: (*filter P xs* = *xs*) = (∀ *x*∈*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*) ≤ *set xs*
⟨*proof*⟩

lemma *length-filter-less*:
⟦ *x* : *set xs*; ~ *P x* ⟧ ⟹ *length*(*filter P xs*) < *length xs*
⟨*proof*⟩

lemma *length-filter-conv-card*:
length(*filter p xs*) = *card*{*i*. *i* < *length xs* & *p*(*xs*!*i*)}
⟨*proof*⟩

lemma *Cons-eq-filterD*:
x#*xs* = *filter P ys* ⟹
∃*us vs*. *ys* = *us* @ *x* # *vs* ∧ (∀*u*∈*set us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *filter P vs*
(**is** - ⟹ ∃*us vs*. *?P ys us vs*)
⟨*proof*⟩

lemma *filter-eq-ConsD*:
filter P ys = *x*#*xs* ⟹
∃*us vs*. *ys* = *us* @ *x* # *vs* ∧ (∀*u*∈*set us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *filter P vs*
⟨*proof*⟩

lemma *filter-eq-Cons-iff*:
(*filter P ys* = *x*#*xs*) =
(∃*us vs*. *ys* = *us* @ *x* # *vs* ∧ (∀*u*∈*set us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *filter P vs*)
⟨*proof*⟩

lemma *Cons-eq-filter-iff*:
(*x*#*xs* = *filter P ys*) =
(∃*us vs*. *ys* = *us* @ *x* # *vs* ∧ (∀*u*∈*set us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *filter P vs*)
⟨*proof*⟩

lemma *filter-cong*[*fundef-cong*, *recdef-cong*]:
xs = *ys* ⟹ (⋀*x*. *x* ∈ *set ys* ⟹ *P x* = *Q x*) ⟹ *filter P xs* = *filter Q ys*
⟨*proof*⟩

43.1.9 List partitioning

primrec *partition* :: ('*a* ⇒ *bool*) ⇒'*a list* ⇒ '*a list* × '*a list* **where**
partition P [] = ([], [])
| *partition P* (*x* # *xs*) =
(*let* (*yes*, *no*) = *partition P xs*
in if P x then (*x* # *yes*, *no*) *else* (*yes*, *x* # *no*))

lemma *partition-filter1*:
fst (*partition P xs*) = *filter P xs*
⟨*proof*⟩

lemma *partition-filter2*:
snd (*partition P xs*) = *filter* (*Not o P*) *xs*
⟨*proof*⟩

lemma *partition-P*:
assumes *partition P xs* = (*yes*, *no*)
shows (∀ *p* ∈ *set yes*. *P p*) ∧ (∀ *p* ∈ *set no*. ¬ *P p*)
⟨*proof*⟩

lemma *partition-set*:
assumes *partition P xs* = (*yes*, *no*)
shows *set yes* ∪ *set no* = *set xs*
⟨*proof*⟩

lemma *partition-filter-conv*[*simp*]:
partition f xs = (*filter f xs*,*filter* (*Not o f*) *xs*)
⟨*proof*⟩

declare *partition.simps*[*simp del*]

43.1.10 *concat*

lemma *concat-append* [*simp*]: *concat* (*xs* @ *ys*) = *concat xs* @ *concat ys*
⟨*proof*⟩

lemma *concat-eq-Nil-conv* [*simp*]: (*concat xss* = []) = (∀ *xs* ∈ *set xss*. *xs* = [])
⟨*proof*⟩

lemma *Nil-eq-concat-conv* [*simp*]: ([] = *concat xss*) = (∀ *xs* ∈ *set xss*. *xs* = [])
⟨*proof*⟩

lemma *set-concat* [*simp*]: *set* (*concat xs*) = (*UN x*:*set xs*. *set x*)
⟨*proof*⟩

lemma *concat-map-singleton*[*simp*]: *concat*(*map* (%*x*. [*f x*]) *xs*) = *map f xs*
⟨*proof*⟩

lemma *map-concat*: *map f* (*concat xs*) = *concat* (*map* (*map f*) *xs*)
⟨*proof*⟩

lemma *filter-concat*: *filter p* (*concat xs*) = *concat* (*map* (*filter p*) *xs*)
⟨*proof*⟩

lemma *rev-concat*: *rev* (*concat xs*) = *concat* (*map rev* (*rev xs*))
⟨*proof*⟩

43.1.11 *nth*

lemma *nth-Cons-0* [*simp*, *code*]: (*x* # *xs*)!*0* = *x*
⟨*proof*⟩

lemma *nth-Cons-Suc* [*simp*, *code*]: (*x* # *xs*)!(*Suc n*) = *xs*!*n*
⟨*proof*⟩

declare *nth.simps* [*simp del*]

lemma *nth-append*:
(*xs* @ *ys*)!*n* = (*if n* < *length xs then xs*!*n else ys*!(*n* − *length xs*))
⟨*proof*⟩

lemma *nth-append-length* [*simp*]: (*xs* @ *x* # *ys*) ! *length xs* = *x*
⟨*proof*⟩

lemma *nth-append-length-plus*[*simp*]: (*xs* @ *ys*) ! (*length xs* + *n*) = *ys* ! *n*
⟨*proof*⟩

lemma *nth-map* [*simp*]: *n* < *length xs* ==> (*map f xs*)!*n* = *f*(*xs*!*n*)
⟨*proof*⟩

lemma *hd-conv-nth*: *xs* ≠ [] ⟹ *hd xs* = *xs*!*0*
⟨*proof*⟩

lemma *list-eq-iff-nth-eq*:
(*xs* = *ys*) = (*length xs* = *length ys* ∧ (*ALL i*<*length xs*. *xs*!*i* = *ys*!*i*))
⟨*proof*⟩

lemma *set-conv-nth*: *set xs* = {*xs*!*i* | *i*. *i* < *length xs*}
⟨*proof*⟩

lemma *in-set-conv-nth*: (*x* ∈ *set xs*) = (∃ *i* < *length xs*. *xs*!*i* = *x*)
⟨*proof*⟩

lemma *list-ball-nth*: [| *n* < *length xs*; !*x* : *set xs*. *P x*|] ==> *P*(*xs*!*n*)
⟨*proof*⟩

lemma *nth-mem* [*simp*]: *n* < *length xs* ==> *xs*!*n* : *set xs*
⟨*proof*⟩

lemma *all-nth-imp-all-set*:
[| !*i* < *length xs*. *P*(*xs*!*i*); *x* : *set xs*|] ==> *P x*
⟨*proof*⟩

lemma *all-set-conv-all-nth*:
(∀ *x* ∈ *set xs*. *P x*) = (∀ *i*. *i* < *length xs* −−> *P* (*xs* ! *i*))
⟨*proof*⟩

lemma *rev-nth*:
n < *size xs* ⟹ *rev xs* ! *n* = *xs* ! (*length xs* − *Suc n*)
⟨*proof*⟩

lemma *Skolem-list-nth*:
(*ALL i*<*k*. *EX x*. *P i x*) = (*EX xs*. *size xs* = *k* & (*ALL i*<*k*. *P i* (*xs*!*i*)))

(**is** - = (*EX xs. ?P k xs*))
⟨*proof*⟩

43.1.12 *list-update*

lemma *length-list-update* [*simp*]: *length*(*xs*[*i*:=*x*]) = *length xs*
⟨*proof*⟩

lemma *nth-list-update*:
i < *length xs*==> (*xs*[*i*:=*x*])!*j* = (*if i* = *j then x else xs*!*j*)
⟨*proof*⟩

lemma *nth-list-update-eq* [*simp*]: *i* < *length xs* ==> (*xs*[*i*:=*x*])!*i* = *x*
⟨*proof*⟩

lemma *nth-list-update-neq* [*simp*]: *i* ≠ *j* ==> *xs*[*i*:=*x*]!*j* = *xs*!*j*
⟨*proof*⟩

lemma *list-update-id*[*simp*]: *xs*[*i* := *xs*!*i*] = *xs*
⟨*proof*⟩

lemma *list-update-beyond*[*simp*]: *length xs* ≤ *i* ⟹ *xs*[*i*:=*x*] = *xs*
⟨*proof*⟩

lemma *list-update-nonempty*[*simp*]: *xs*[*k*:=*x*] = [] ⟷ *xs*=[]
⟨*proof*⟩

lemma *list-update-same-conv*:
i < *length xs* ==> (*xs*[*i* := *x*] = *xs*) = (*xs*!*i* = *x*)
⟨*proof*⟩

lemma *list-update-append1*:
i < *size xs* ⟹ (*xs* @ *ys*)[*i*:=*x*] = *xs*[*i*:=*x*] @ *ys*
⟨*proof*⟩

lemma *list-update-append*:
(*xs* @ *ys*) [*n*:= *x*] =
(*if n* < *length xs then xs*[*n*:= *x*] @ *ys else xs* @ (*ys* [*n*−*length xs*:= *x*]))
⟨*proof*⟩

lemma *list-update-length* [*simp*]:
(*xs* @ *x* # *ys*)[*length xs* := *y*] = (*xs* @ *y* # *ys*)
⟨*proof*⟩

lemma *map-update*: *map f* (*xs*[*k*:= *y*]) = (*map f xs*)[*k* := *f y*]
⟨*proof*⟩

lemma *rev-update*:
k < *length xs* ⟹ *rev* (*xs*[*k*:= *y*]) = (*rev xs*)[*length xs* − *k* − *1* := *y*]

⟨*proof*⟩

lemma *update-zip*:
(*zip xs ys*)[*i*:=*xy*] = *zip* (*xs*[*i*:=*fst xy*]) (*ys*[*i*:=*snd xy*])
⟨*proof*⟩

lemma *set-update-subset-insert*: *set*(*xs*[*i*:=*x*]) <= *insert x* (*set xs*)
⟨*proof*⟩

lemma *set-update-subsetI*: [| *set xs* <= *A*; *x*:*A* |] ==> *set*(*xs*[*i* := *x*]) <= *A*
⟨*proof*⟩

lemma *set-update-memI*: *n* < *length xs* ⟹ *x* ∈ *set* (*xs*[*n* := *x*])
⟨*proof*⟩

lemma *list-update-overwrite*[*simp*]:
xs [*i* := *x*, *i* := *y*] = *xs* [*i* := *y*]
⟨*proof*⟩

lemma *list-update-swap*:
i ≠ *i'* ⟹ *xs* [*i* := *x*, *i'* := *x'*] = *xs* [*i'* := *x'*, *i* := *x*]
⟨*proof*⟩

lemma *list-update-code* [*code*]:
[][*i* := *y*] = []
(*x* # *xs*)[*0* := *y*] = *y* # *xs*
(*x* # *xs*)[*Suc i* := *y*] = *x* # *xs*[*i* := *y*]
⟨*proof*⟩

43.1.13 *last* and *butlast*

lemma *last-snoc* [*simp*]: *last* (*xs* @ [*x*]) = *x*
⟨*proof*⟩

lemma *butlast-snoc* [*simp*]: *butlast* (*xs* @ [*x*]) = *xs*
⟨*proof*⟩

lemma *last-ConsL*: *xs* = [] ⟹ *last*(*x*#*xs*) = *x*
⟨*proof*⟩

lemma *last-ConsR*: *xs* ≠ [] ⟹ *last*(*x*#*xs*) = *last xs*
⟨*proof*⟩

lemma *last-append*: *last*(*xs* @ *ys*) = (*if ys* = [] *then last xs else last ys*)
⟨*proof*⟩

lemma *last-appendL*[*simp*]: *ys* = [] ⟹ *last*(*xs* @ *ys*) = *last xs*
⟨*proof*⟩

lemma *last-appendR*[*simp*]: $ys \neq [] \Longrightarrow last(xs \mathbin{@} ys) = last\ ys$
⟨*proof*⟩

lemma *hd-rev*: $xs \neq [] \Longrightarrow hd(rev\ xs) = last\ xs$
⟨*proof*⟩

lemma *last-rev*: $xs \neq [] \Longrightarrow last(rev\ xs) = hd\ xs$
⟨*proof*⟩

lemma *last-in-set*[*simp*]: $as \neq [] \Longrightarrow last\ as \in set\ as$
⟨*proof*⟩

lemma *length-butlast* [*simp*]: $length\ (butlast\ xs) = length\ xs - 1$
⟨*proof*⟩

lemma *butlast-append*:
$butlast\ (xs \mathbin{@} ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs \mathbin{@} butlast\ ys)$
⟨*proof*⟩

lemma *append-butlast-last-id* [*simp*]:
$xs \neq [] ==> butlast\ xs \mathbin{@} [last\ xs] = xs$
⟨*proof*⟩

lemma *in-set-butlastD*: $x : set\ (butlast\ xs) ==> x : set\ xs$
⟨*proof*⟩

lemma *in-set-butlast-appendI*:
$x : set\ (butlast\ xs) \mid x : set\ (butlast\ ys) ==> x : set\ (butlast\ (xs \mathbin{@} ys))$
⟨*proof*⟩

lemma *last-drop*[*simp*]: $n < length\ xs \Longrightarrow last\ (drop\ n\ xs) = last\ xs$
⟨*proof*⟩

lemma *last-conv-nth*: $xs \neq [] \Longrightarrow last\ xs = xs!(length\ xs - 1)$
⟨*proof*⟩

lemma *butlast-conv-take*: $butlast\ xs = take\ (length\ xs - 1)\ xs$
⟨*proof*⟩

lemma *last-list-update*:
$xs \neq [] \Longrightarrow last(xs[k:=x]) = (if\ k = size\ xs - 1\ then\ x\ else\ last\ xs)$
⟨*proof*⟩

lemma *butlast-list-update*:
$butlast(xs[k:=x]) =$
$(if\ k = size\ xs - 1\ then\ butlast\ xs\ else\ (butlast\ xs)[k:=x])$
⟨*proof*⟩

lemma *last-map*:

$xs \neq [] \Longrightarrow last\ (map\ f\ xs) = f\ (last\ xs)$
⟨*proof*⟩

lemma *map-butlast*:
$map\ f\ (butlast\ xs) = butlast\ (map\ f\ xs)$
⟨*proof*⟩

43.1.14 *take* and *drop*

lemma *take-0* [*simp*]: $take\ 0\ xs = []$
⟨*proof*⟩

lemma *drop-0* [*simp*]: $drop\ 0\ xs = xs$
⟨*proof*⟩

lemma *take-Suc-Cons* [*simp*]: $take\ (Suc\ n)\ (x\ \#\ xs) = x\ \#\ take\ n\ xs$
⟨*proof*⟩

lemma *drop-Suc-Cons* [*simp*]: $drop\ (Suc\ n)\ (x\ \#\ xs) = drop\ n\ xs$
⟨*proof*⟩

declare *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

lemma *take-1-Cons* [*simp*]: $take\ 1\ (x\ \#\ xs) = [x]$
⟨*proof*⟩

lemma *drop-1-Cons* [*simp*]: $drop\ 1\ (x\ \#\ xs) = xs$
⟨*proof*⟩

lemma *take-Suc*: $xs \sim= [] ==> take\ (Suc\ n)\ xs = hd\ xs\ \#\ take\ n\ (tl\ xs)$
⟨*proof*⟩

lemma *drop-Suc*: $drop\ (Suc\ n)\ xs = drop\ n\ (tl\ xs)$
⟨*proof*⟩

lemma *take-tl*: $take\ n\ (tl\ xs) = tl\ (take\ (Suc\ n)\ xs)$
⟨*proof*⟩

lemma *drop-tl*: $drop\ n\ (tl\ xs) = tl(drop\ n\ xs)$
⟨*proof*⟩

lemma *tl-take*: $tl\ (take\ n\ xs) = take\ (n - 1)\ (tl\ xs)$
⟨*proof*⟩

lemma *tl-drop*: $tl\ (drop\ n\ xs) = drop\ n\ (tl\ xs)$
⟨*proof*⟩

lemma *nth-via-drop*: $drop\ n\ xs = y\#ys \Longrightarrow xs!n = y$
⟨*proof*⟩

lemma *take-Suc-conv-app-nth*:
$i < length\ xs \Longrightarrow take\ (Suc\ i)\ xs = take\ i\ xs\ @\ [xs!i]$
⟨*proof*⟩

lemma *drop-Suc-conv-tl*:
$i < length\ xs \Longrightarrow (xs!i)\ \#\ (drop\ (Suc\ i)\ xs) = drop\ i\ xs$
⟨*proof*⟩

lemma *length-take* [*simp*]: $length\ (take\ n\ xs) = min\ (length\ xs)\ n$
⟨*proof*⟩

lemma *length-drop* [*simp*]: $length\ (drop\ n\ xs) = (length\ xs - n)$
⟨*proof*⟩

lemma *take-all* [*simp*]: *length xs <= n ==> take n xs = xs*
⟨*proof*⟩

lemma *drop-all* [*simp*]: *length xs <= n ==> drop n xs = []*
⟨*proof*⟩

lemma *take-append* [*simp*]:
$take\ n\ (xs\ @\ ys) = (take\ n\ xs\ @\ take\ (n - length\ xs)\ ys)$
⟨*proof*⟩

lemma *drop-append* [*simp*]:
$drop\ n\ (xs\ @\ ys) = drop\ n\ xs\ @\ drop\ (n - length\ xs)\ ys$
⟨*proof*⟩

lemma *take-take* [*simp*]: $take\ n\ (take\ m\ xs) = take\ (min\ n\ m)\ xs$
⟨*proof*⟩

lemma *drop-drop* [*simp*]: $drop\ n\ (drop\ m\ xs) = drop\ (n + m)\ xs$
⟨*proof*⟩

lemma *take-drop*: $take\ n\ (drop\ m\ xs) = drop\ m\ (take\ (n + m)\ xs)$
⟨*proof*⟩

lemma *drop-take*: $drop\ n\ (take\ m\ xs) = take\ (m-n)\ (drop\ n\ xs)$
⟨*proof*⟩

lemma *append-take-drop-id* [*simp*]: *take n xs @ drop n xs = xs*
⟨*proof*⟩

lemma *take-eq-Nil*[*simp*]: $(take\ n\ xs = []) = (n = 0 \vee xs = [])$
⟨*proof*⟩

lemma *drop-eq-Nil*[*simp*]: $(drop\ n\ xs = []) = (length\ xs <= n)$
⟨*proof*⟩

lemma *take-map*: *take n (map f xs) = map f (take n xs)*
⟨*proof*⟩

lemma *drop-map*: *drop n (map f xs) = map f (drop n xs)*
⟨*proof*⟩

lemma *rev-take*: *rev (take i xs) = drop (length xs − i) (rev xs)*
⟨*proof*⟩

lemma *rev-drop*: *rev (drop i xs) = take (length xs − i) (rev xs)*
⟨*proof*⟩

lemma *nth-take* [*simp*]: *i < n ==> (take n xs)!i = xs!i*
⟨*proof*⟩

lemma *nth-drop* [*simp*]:
n + i <= length xs ==> (drop n xs)!i = xs!(n + i)
⟨*proof*⟩

lemma *butlast-take*:
n <= length xs ==> butlast (take n xs) = take (n − 1) xs
⟨*proof*⟩

lemma *butlast-drop*: *butlast (drop n xs) = drop n (butlast xs)*
⟨*proof*⟩

lemma *take-butlast*: *n < length xs ==> take n (butlast xs) = take n xs*
⟨*proof*⟩

lemma *drop-butlast*: *drop n (butlast xs) = butlast (drop n xs)*
⟨*proof*⟩

lemma *hd-drop-conv-nth*: *⟦ xs ≠ []; n < length xs ⟧ ⟹ hd(drop n xs) = xs!n*
⟨*proof*⟩

lemma *set-take-subset-set-take*:
m <= n ⟹ set(take m xs) <= set(take n xs)
⟨*proof*⟩

lemma *set-take-subset*: *set(take n xs) ⊆ set xs*
⟨*proof*⟩

lemma *set-drop-subset*: *set(drop n xs) ⊆ set xs*
⟨*proof*⟩

lemma *set-drop-subset-set-drop*:
m >= n ⟹ set(drop m xs) <= set(drop n 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*:
$(xs\ @\ ys = zs) = (xs = take\ (length\ xs)\ zs \wedge ys = drop\ (length\ xs)\ zs)$
⟨*proof*⟩

lemma *take-add*:
$i+j \leq length(xs) \Longrightarrow take\ (i+j)\ xs = take\ i\ xs\ @\ take\ j\ (drop\ i\ xs)$
⟨*proof*⟩

lemma *append-eq-append-conv-if*:
$(xs_1\ @\ xs_2 = ys_1\ @\ ys_2) =$
$(if\ size\ xs_1 \leq size\ ys_1$
$\ then\ xs_1 = take\ (size\ xs_1)\ ys_1 \wedge xs_2 = drop\ (size\ xs_1)\ ys_1\ @\ ys_2$
$\ else\ take\ (size\ ys_1)\ xs_1 = ys_1 \wedge drop\ (size\ ys_1)\ xs_1\ @\ xs_2 = ys_2)$
⟨*proof*⟩

lemma *take-hd-drop*:
$n < length\ xs \Longrightarrow take\ n\ xs\ @\ [hd\ (drop\ n\ xs)] = take\ (Suc\ n)\ xs$
⟨*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*⟩

lemma *nth-drop'*:
$i < length\ xs \Longrightarrow xs\ !\ i\ \#\ drop\ (Suc\ i)\ xs = drop\ i\ xs$
⟨*proof*⟩

43.1.15 *takeWhile* and *dropWhile*

lemma *length-takeWhile-le*: $length\ (takeWhile\ P\ xs) \leq length\ xs$
⟨*proof*⟩

lemma *takeWhile-dropWhile-id* [*simp*]: $takeWhile\ P\ xs\ @\ dropWhile\ P\ xs = xs$
⟨*proof*⟩

lemma *takeWhile-append1* [*simp*]:
[| x:*set xs*; ~$P(x)$|] ==> $takeWhile\ P\ (xs\ @\ ys) = takeWhile\ P\ xs$
⟨*proof*⟩

lemma *takeWhile-append2* [*simp*]:
(!!*x*. *x* : *set xs* ==> *P x*) ==> *takeWhile P* (*xs* @ *ys*) = *xs* @ *takeWhile P ys*
⟨*proof*⟩

lemma *takeWhile-tail*: ¬ *P x* ==> *takeWhile P* (*xs* @ (*x*#*l*)) = *takeWhile P xs*
⟨*proof*⟩

lemma *takeWhile-nth*: *j* < *length* (*takeWhile P xs*) ⟹ *takeWhile P xs* ! *j* = *xs*
! *j*
⟨*proof*⟩

lemma *dropWhile-nth*: *j* < *length* (*dropWhile P xs*) ⟹ *dropWhile P xs* ! *j* = *xs*
! (*j* + *length* (*takeWhile P xs*))
⟨*proof*⟩

lemma *length-dropWhile-le*: *length* (*dropWhile P xs*) ≤ *length xs*
⟨*proof*⟩

lemma *dropWhile-append1* [*simp*]:
[| *x* : *set xs*; ~*P*(*x*)|] ==> *dropWhile P* (*xs* @ *ys*) = (*dropWhile P xs*)@*ys*
⟨*proof*⟩

lemma *dropWhile-append2* [*simp*]:
(!!*x*. *x*:*set xs* ==> *P*(*x*)) ==> *dropWhile P* (*xs* @ *ys*) = *dropWhile P ys*
⟨*proof*⟩

lemma *set-takeWhileD*: *x* : *set* (*takeWhile P xs*) ==> *x* : *set xs* ∧ *P x*
⟨*proof*⟩

lemma *takeWhile-eq-all-conv*[*simp*]:
(*takeWhile P xs* = *xs*) = (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemma *dropWhile-eq-Nil-conv*[*simp*]:
(*dropWhile P xs* = []) = (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemma *dropWhile-eq-Cons-conv*:
(*dropWhile P xs* = *y*#*ys*) = (*xs* = *takeWhile P xs* @ *y* # *ys* & ¬ *P y*)
⟨*proof*⟩

lemma *distinct-takeWhile*[*simp*]: *distinct xs* ==> *distinct* (*takeWhile P xs*)
⟨*proof*⟩

lemma *distinct-dropWhile*[*simp*]: *distinct xs* ==> *distinct* (*dropWhile P xs*)
⟨*proof*⟩

lemma *takeWhile-map*: *takeWhile P* (*map f xs*) = *map f* (*takeWhile* (*P* ∘ *f*) *xs*)

⟨*proof*⟩

lemma *dropWhile-map*: *dropWhile P* (*map f xs*) = *map f* (*dropWhile* (*P* ∘ *f*) *xs*)
⟨*proof*⟩

lemma *takeWhile-eq-take*: *takeWhile P xs* = *take* (*length* (*takeWhile P xs*)) *xs*
⟨*proof*⟩

lemma *dropWhile-eq-drop*: *dropWhile P xs* = *drop* (*length* (*takeWhile P xs*)) *xs*
⟨*proof*⟩

lemma *hd-dropWhile*:
dropWhile P xs ≠ [] ⟹ ¬ *P* (*hd* (*dropWhile P xs*))
⟨*proof*⟩

lemma *takeWhile-eq-filter*:
assumes ⋀ *x*. *x* ∈ *set* (*dropWhile P xs*) ⟹ ¬ *P x*
shows *takeWhile P xs* = *filter P xs*
⟨*proof*⟩

lemma *takeWhile-eq-take-P-nth*:
⟦ ⋀ *i*. ⟦ *i* < *n* ; *i* < *length xs* ⟧ ⟹ *P* (*xs* ! *i*) ; *n* < *length xs* ⟹ ¬ *P* (*xs* ! *n*)
⟧ ⟹
takeWhile P xs = *take n xs*
⟨*proof*⟩

lemma *nth-length-takeWhile*:
length (*takeWhile P xs*) < *length xs* ⟹ ¬ *P* (*xs* ! *length* (*takeWhile P xs*))
⟨*proof*⟩

lemma *length-takeWhile-less-P-nth*:
assumes *all*: ⋀ *i*. *i* < *j* ⟹ *P* (*xs* ! *i*) **and** *j* ≤ *length xs*
shows *j* ≤ *length* (*takeWhile P xs*)
⟨*proof*⟩

The following two lemmmas could be generalized to an arbitrary property.

lemma *takeWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
takeWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *rev* (*tl* (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*))
⟨*proof*⟩

lemma *dropWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
dropWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *x* # *rev* (*takeWhile* (λ*y*. *y* ≠ *x*) *xs*)
⟨*proof*⟩

lemma *takeWhile-not-last*:
⟦ *xs* ≠ []; *distinct xs*⟧ ⟹ *takeWhile* (λ*y*. *y* ≠ *last xs*) *xs* = *butlast xs*
⟨*proof*⟩

lemma *takeWhile-cong* [*fundef-cong*, *recdef-cong*]:

[| $l = k$; !!x. x : *set* l ==> $P\ x = Q\ x$ |]
==> *takeWhile* $P\ l$ = *takeWhile* $Q\ k$
⟨*proof*⟩

lemma *dropWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| $l = k$; !!x. x : *set* l ==> $P\ x = Q\ x$ |]
==> *dropWhile* $P\ l$ = *dropWhile* $Q\ k$
⟨*proof*⟩

43.1.16 *zip*

lemma *zip-Nil* [*simp*]: *zip* [] *ys* = []
⟨*proof*⟩

lemma *zip-Cons-Cons* [*simp*]: *zip* (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip xs ys*
⟨*proof*⟩

declare *zip-Cons* [*simp del*]

lemma [*code*]:
zip [] *ys* = []
zip xs [] = []
zip (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip xs ys*
⟨*proof*⟩

lemma *zip-Cons1*:
zip (*x*#*xs*) *ys* = (*case ys of* [] ⇒ [] | *y*#*ys* ⇒ (*x*,*y*)#*zip xs ys*)
⟨*proof*⟩

lemma *length-zip* [*simp*]:
length (*zip xs ys*) = *min* (*length xs*) (*length ys*)
⟨*proof*⟩

lemma *zip-obtain-same-length*:
assumes ⋀*zs ws n*. *length zs* = *length ws* ⟹ *n* = *min* (*length xs*) (*length ys*)
⟹ *zs* = *take n xs* ⟹ *ws* = *take n ys* ⟹ *P* (*zip zs ws*)
shows *P* (*zip xs ys*)
⟨*proof*⟩

lemma *zip-append1*:
zip (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
⟨*proof*⟩

lemma *zip-append2*:
zip xs (*ys* @ *zs*) =
zip (*take* (*length ys*) *xs*) *ys* @ *zip* (*drop* (*length ys*) *xs*) *zs*
⟨*proof*⟩

lemma *zip-append* [*simp*]:
[| *length xs* = *length us*; *length ys* = *length vs* |] ==>
zip (*xs*@*ys*) (*us*@*vs*) = *zip xs us* @ *zip ys vs*
⟨*proof*⟩

lemma *zip-rev*:
length xs = *length ys* ==> *zip* (*rev xs*) (*rev ys*) = *rev* (*zip xs ys*)
⟨*proof*⟩

lemma *zip-map-map*:
zip (*map f xs*) (*map g ys*) = *map* (λ (*x*, *y*). (*f x*, *g y*)) (*zip xs ys*)
⟨*proof*⟩

lemma *zip-map1*:
zip (*map f xs*) *ys* = *map* (λ(*x*, *y*). (*f x*, *y*)) (*zip xs ys*)
⟨*proof*⟩

lemma *zip-map2*:
zip xs (*map f ys*) = *map* (λ(*x*, *y*). (*x*, *f y*)) (*zip xs ys*)
⟨*proof*⟩

lemma *map-zip-map*:
map f (*zip* (*map g xs*) *ys*) = *map* (%(*x*,*y*). *f*(*g x*, *y*)) (*zip xs ys*)
⟨*proof*⟩

lemma *map-zip-map2*:
map f (*zip xs* (*map g ys*)) = *map* (%(*x*,*y*). *f*(*x*, *g y*)) (*zip xs ys*)
⟨*proof*⟩

Courtesy of Andreas Lochbihler:

lemma *zip-same-conv-map*: *zip xs xs* = *map* (λ*x*. (*x*, *x*)) *xs*
⟨*proof*⟩

lemma *nth-zip* [*simp*]:
[| *i* < *length xs*; *i* < *length ys*|] ==> (*zip xs ys*)!*i* = (*xs*!*i*, *ys*!*i*)
⟨*proof*⟩

lemma *set-zip*:
set (*zip xs ys*) = {(*xs*!*i*, *ys*!*i*) | *i*. *i* < *min* (*length xs*) (*length ys*)}
⟨*proof*⟩

lemma *zip-same*: ((*a*,*b*) ∈ *set* (*zip xs xs*)) = (*a* ∈ *set xs* ∧ *a* = *b*)
⟨*proof*⟩

lemma *zip-update*:
zip (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*]) = (*zip xs ys*)[*i*:=(*x*,*y*)]
⟨*proof*⟩

lemma *zip-replicate* [*simp*]:

zip (*replicate i x*) (*replicate j y*) = *replicate* (*min i j*) (*x*,*y*)
⟨*proof*⟩

lemma *take-zip*:
take n (*zip xs ys*) = *zip* (*take n xs*) (*take n ys*)
⟨*proof*⟩

lemma *drop-zip*:
drop n (*zip xs ys*) = *zip* (*drop n xs*) (*drop n ys*)
⟨*proof*⟩

lemma *zip-takeWhile-fst*: *zip* (*takeWhile P xs*) *ys* = *takeWhile* (*P* ∘ *fst*) (*zip xs ys*)
⟨*proof*⟩

lemma *zip-takeWhile-snd*: *zip xs* (*takeWhile P ys*) = *takeWhile* (*P* ∘ *snd*) (*zip xs ys*)
⟨*proof*⟩

lemma *set-zip-leftD*:
(*x*,*y*)∈ *set* (*zip xs ys*) ⟹ *x* ∈ *set xs*
⟨*proof*⟩

lemma *set-zip-rightD*:
(*x*,*y*)∈ *set* (*zip xs ys*) ⟹ *y* ∈ *set ys*
⟨*proof*⟩

lemma *in-set-zipE*:
(*x*,*y*) : *set*(*zip xs ys*) ⟹ (⟦ *x* : *set xs*; *y* : *set ys* ⟧ ⟹ *R*) ⟹ *R*
⟨*proof*⟩

lemma *zip-map-fst-snd*:
zip (*map fst zs*) (*map snd zs*) = *zs*
⟨*proof*⟩

lemma *zip-eq-conv*:
length xs = *length ys* ⟹ *zip xs ys* = *zs* ⟷ *map fst zs* = *xs* ∧ *map snd zs* = *ys*
⟨*proof*⟩

lemma *distinct-zipI1*:
distinct xs ⟹ *distinct* (*zip xs ys*)
⟨*proof*⟩

lemma *distinct-zipI2*:
distinct xs ⟹ *distinct* (*zip xs ys*)
⟨*proof*⟩

43.1.17 *list-all2*

lemma *list-all2-lengthD* [*intro?*]:
list-all2 P xs ys ==> length xs = length ys
⟨*proof*⟩

lemma *list-all2-Nil* [*iff*, *code*]: *list-all2 P [] ys = (ys = [])*
⟨*proof*⟩

lemma *list-all2-Nil2* [*iff*, *code*]: *list-all2 P xs [] = (xs = [])*
⟨*proof*⟩

lemma *list-all2-Cons* [*iff*, *code*]:
list-all2 P (x # xs) (y # ys) = (P x y ∧ list-all2 P xs ys)
⟨*proof*⟩

lemma *list-all2-Cons1*:
list-all2 P (x # xs) ys = (∃ z zs. ys = z # zs ∧ P x z ∧ list-all2 P xs zs)
⟨*proof*⟩

lemma *list-all2-Cons2*:
list-all2 P xs (y # ys) = (∃ z zs. xs = z # zs ∧ P z y ∧ list-all2 P zs ys)
⟨*proof*⟩

lemma *list-all2-rev* [*iff*]:
list-all2 P (rev xs) (rev ys) = list-all2 P xs ys
⟨*proof*⟩

lemma *list-all2-rev1*:
list-all2 P (rev xs) ys = list-all2 P xs (rev ys)
⟨*proof*⟩

lemma *list-all2-append1*:
list-all2 P (xs @ ys) zs =
(EX us vs. zs = us @ vs ∧ length us = length xs ∧ length vs = length ys ∧
list-all2 P xs us ∧ list-all2 P ys vs)
⟨*proof*⟩

lemma *list-all2-append2*:
list-all2 P xs (ys @ zs) =
(EX us vs. xs = us @ vs ∧ length us = length ys ∧ length vs = length zs ∧
list-all2 P us ys ∧ list-all2 P vs zs)
⟨*proof*⟩

lemma *list-all2-append*:
length xs = length ys ⟹
list-all2 P (xs@us) (ys@vs) = (list-all2 P xs ys ∧ list-all2 P us vs)
⟨*proof*⟩

lemma *list-all2-appendI* [*intro?*, *trans*]:

⟦ *list-all2 P a b*; *list-all2 P c d* ⟧ ⟹ *list-all2 P* (*a*@*c*) (*b*@*d*)
⟨*proof*⟩

lemma *list-all2-conv-all-nth*:
list-all2 P xs ys =
(*length xs* = *length ys* ∧ (∀ *i* < *length xs*. *P* (*xs*!*i*) (*ys*!*i*)))
⟨*proof*⟩

lemma *list-all2-trans*:
assumes *tr*: !!*a b c*. *P1 a b* ==> *P2 b c* ==> *P3 a c*
shows !!*bs cs*. *list-all2 P1 as bs* ==> *list-all2 P2 bs cs* ==> *list-all2 P3 as cs*
(**is** !!*bs cs*. *PROP ?Q as bs cs*)
⟨*proof*⟩

lemma *list-all2-all-nthI* [*intro?*]:
length a = *length b* ⟹ (⋀*n*. *n* < *length a* ⟹ *P* (*a*!*n*) (*b*!*n*)) ⟹ *list-all2 P a b*
⟨*proof*⟩

lemma *list-all2I*:
∀ *x* ∈ *set* (*zip a b*). *split P x* ⟹ *length a* = *length b* ⟹ *list-all2 P a b*
⟨*proof*⟩

lemma *list-all2-nthD*:
⟦ *list-all2 P xs ys*; *p* < *size xs* ⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
⟨*proof*⟩

lemma *list-all2-nthD2*:
⟦*list-all2 P xs ys*; *p* < *size ys*⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
⟨*proof*⟩

lemma *list-all2-map1*:
list-all2 P (*map f as*) *bs* = *list-all2* (λ*x y*. *P* (*f x*) *y*) *as bs*
⟨*proof*⟩

lemma *list-all2-map2*:
list-all2 P as (*map f bs*) = *list-all2* (λ*x y*. *P x* (*f y*)) *as bs*
⟨*proof*⟩

lemma *list-all2-refl* [*intro?*]:
(⋀*x*. *P x x*) ⟹ *list-all2 P xs xs*
⟨*proof*⟩

lemma *list-all2-update-cong*:
⟦ *i*<*size xs*; *list-all2 P xs ys*; *P x y* ⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
⟨*proof*⟩

lemma *list-all2-update-cong2*:
⟦*list-all2 P xs ys*; *P x y*; *i* < *length ys*⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
⟨*proof*⟩

lemma *list-all2-takeI* [*simp*,*intro?*]:
list-all2 P xs ys $\Longrightarrow$ *list-all2 P* (*take n xs*) (*take n ys*)
⟨*proof*⟩

lemma *list-all2-dropI* [*simp*,*intro?*]:
list-all2 P as bs $\Longrightarrow$ *list-all2 P* (*drop n as*) (*drop n bs*)
⟨*proof*⟩

lemma *list-all2-mono* [*intro?*]:
list-all2 P xs ys $\Longrightarrow$ ($\bigwedge$*xs ys*. *P xs ys* $\Longrightarrow$ *Q xs ys*) $\Longrightarrow$ *list-all2 Q xs ys*
⟨*proof*⟩

lemma *list-all2-eq*:
xs = *ys* $\longleftrightarrow$ *list-all2* (*op* =) *xs ys*
⟨*proof*⟩

43.1.18 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:
foldl f a (*xs* @ *ys*) = *foldl f* (*foldl f a xs*) *ys*
⟨*proof*⟩

lemma *foldr-append*[*simp*]: *foldr f* (*xs* @ *ys*) *a* = *foldr f xs* (*foldr f ys a*)
⟨*proof*⟩

lemma *foldr-map*: *foldr g* (*map f xs*) *a* = *foldr* (*g o f*) *xs a*
⟨*proof*⟩

For efficient code generation: avoid intermediate list.

lemma *foldl-map*[*code-unfold*]:
foldl g a (*map f xs*) = *foldl* (%*a x*. *g a* (*f x*)) *a xs*
⟨*proof*⟩

lemma *foldl-apply*:
assumes $\bigwedge$*x*. *x* ∈ *set xs* $\Longrightarrow$ *f x* ∘ *h* = *h* ∘ *g x*
shows *foldl* (λ*s x*. *f x s*) (*h s*) *xs* = *h* (*foldl* (λ*s x*. *g x s*) *s xs*)
⟨*proof*⟩

lemma *foldl-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f a x* = *g a x* |]
==> *foldl f a l* = *foldl g b k*
⟨*proof*⟩

lemma *foldr-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f x a* = *g x a* |]
==> *foldr f l a* = *foldr g k b*
⟨*proof*⟩

lemma *foldl-fun-comm*:
assumes $\bigwedge x\ y\ s.\ f\ (f\ s\ x)\ y = f\ (f\ s\ y)\ x$
shows $f\ (foldl\ f\ s\ xs)\ x = foldl\ f\ (f\ s\ x)\ xs$
⟨*proof*⟩

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows $foldl\ op{+}\ (x{+}y)\ zs = x + (foldl\ op{+}\ y\ zs)$
⟨*proof*⟩

lemma (**in** *monoid-add*) *foldl-absorb0*:
shows $x + (foldl\ op{+}\ 0\ zs) = foldl\ op{+}\ x\ zs$
⟨*proof*⟩

lemma *foldl-rev*:
assumes $\bigwedge x\ y\ s.\ f\ (f\ s\ x)\ y = f\ (f\ s\ y)\ x$
shows $foldl\ f\ s\ (rev\ xs) = foldl\ f\ s\ xs$
⟨*proof*⟩

The "First Duality Theorem" in Bird & Wadler:

lemma *foldl-foldr1-lemma*:
$foldl\ op\ +\ a\ xs = a + foldr\ op\ +\ xs\ (0{::}'a{::}monoid\text{-}add)$
⟨*proof*⟩

corollary *foldl-foldr1*:
$foldl\ op\ +\ 0\ xs = foldr\ op\ +\ xs\ (0{::}'a{::}monoid\text{-}add)$
⟨*proof*⟩

The "Third Duality Theorem" in Bird & Wadler:

lemma *foldr-foldl*: $foldr\ f\ xs\ a = foldl\ (\%x\ y.\ f\ y\ x)\ a\ (rev\ xs)$
⟨*proof*⟩

lemma *foldl-foldr*: $foldl\ f\ a\ xs = foldr\ (\%x\ y.\ f\ y\ x)\ (rev\ xs)\ a$
⟨*proof*⟩

lemma (**in** *ab-semigroup-add*) *foldr-conv-foldl*: $foldr\ op\ +\ xs\ a = foldl\ op\ +\ a\ xs$
⟨*proof*⟩

Note: $n \leq foldl\ (op\ +)\ n\ ns$ looks simpler, but is more difficult to use because it requires an additional transitivity step.

lemma *start-le-sum*: $(m{::}nat) <= n ==> m <= foldl\ (op\ +)\ n\ ns$
⟨*proof*⟩

lemma *elem-le-sum*: $(n{::}nat) : set\ ns ==> n <= foldl\ (op\ +)\ 0\ ns$
⟨*proof*⟩

lemma *sum-eq-0-conv* [*iff*]:
$(foldl\ (op\ +)\ (m{::}nat)\ ns = 0) = (m = 0 \wedge (\forall n \in set\ ns.\ n = 0))$
⟨*proof*⟩

lemma *foldr-invariant*:
⟦*Q x* ; ∀ *x*∈ *set xs*. *P x*; ∀ *x y*. *P x* ∧ *Q y* ⟶ *Q* (*f x y*) ⟧ ⟹ *Q* (*foldr f xs x*)
⟨*proof*⟩

lemma *foldl-invariant*:
⟦*Q x* ; ∀ *x*∈ *set xs*. *P x*; ∀ *x y*. *P x* ∧ *Q y* ⟶ *Q* (*f y x*) ⟧ ⟹ *Q* (*foldl f x xs*)
⟨*proof*⟩

lemma *foldl-weak-invariant*:
assumes *P s*
and ⋀*s x*. *x* ∈ *set xs* ⟹ *P s* ⟹ *P* (*f s x*)
shows *P* (*foldl f s xs*)
⟨*proof*⟩

foldl and *concat*

lemma *foldl-conv-concat*:
foldl (*op* @) *xs xss* = *xs* @ *concat xss*
⟨*proof*⟩

lemma *concat-conv-foldl*: *concat xss* = *foldl* (*op* @) [] *xss*
⟨*proof*⟩

fold and *foldl*

lemma (**in** *fun-left-comm*) *fold-set-remdups*:
fold f y (*set xs*) = *foldl* (λ*y x*. *f x y*) *y* (*remdups xs*)
⟨*proof*⟩

lemma (**in** *fun-left-comm-idem*) *fold-set*:
fold f y (*set xs*) = *foldl* (λ*y x*. *f x y*) *y xs*
⟨*proof*⟩

lemma (**in** *ab-semigroup-idem-mult*) *fold1-set*:
assumes *xs* ≠ []
shows *fold1 times* (*set xs*) = *foldl times* (*hd xs*) (*tl xs*)
⟨*proof*⟩

lemma (**in** *lattice*) *Inf-fin-set-fold* [*code-unfold*]:
Inf-fin (*set* (*x* # *xs*)) = *foldl inf x xs*
⟨*proof*⟩

lemma (**in** *lattice*) *Sup-fin-set-fold* [*code-unfold*]:
Sup-fin (*set* (*x* # *xs*)) = *foldl sup x xs*
⟨*proof*⟩

lemma (**in** *linorder*) *Min-fin-set-fold* [*code-unfold*]:
Min (*set* (*x* # *xs*)) = *foldl min x xs*
⟨*proof*⟩

lemma (**in** *linorder*) *Max-fin-set-fold* [*code-unfold*]:

Max (set (x # xs)) = foldl max x xs
⟨*proof*⟩

lemma (**in** *complete-lattice*) *Inf-set-fold* [*code-unfold*]:
Inf (set xs) = foldl inf top xs
⟨*proof*⟩

lemma (**in** *complete-lattice*) *Sup-set-fold* [*code-unfold*]:
Sup (set xs) = foldl sup bot xs
⟨*proof*⟩

lemma (**in** *complete-lattice*) *INFI-set-fold*:
INFI (set xs) f = foldl (λy x. inf (f x) y) top xs
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUPR-set-fold*:
SUPR (set xs) f = foldl (λy x. sup (f x) y) bot xs
⟨*proof*⟩

43.1.19 List summation: *listsum* and $\sum$

lemma *listsum-append* [*simp*]: *listsum (xs @ ys) = listsum xs + listsum ys*
⟨*proof*⟩

lemma *listsum-rev* [*simp*]:
fixes *xs* :: *'a*::*comm-monoid-add list*
shows *listsum (rev xs) = listsum xs*
⟨*proof*⟩

lemma *listsum-map-remove1*:
fixes *f* :: *'a ⇒ ('b*::*comm-monoid-add)*
shows *x* : *set xs ⟹ listsum(map f xs) = f x + listsum(map f (remove1 x xs))*
⟨*proof*⟩

lemma *list-size-conv-listsum*:
list-size f xs = listsum (map f xs) + size xs
⟨*proof*⟩

lemma *listsum-foldr*: *listsum xs = foldr (op +) xs 0*
⟨*proof*⟩

lemma *length-concat*: *length (concat xss) = listsum (map length xss)*
⟨*proof*⟩

lemma *listsum-map-filter*:
fixes *f* :: *'a ⇒ 'b* :: *comm-monoid-add*
assumes ⋀ *x*. ⟦ *x* ∈ *set xs* ; ¬ *P x* ⟧ ⟹ *f x = 0*
shows *listsum (map f (filter P xs)) = listsum (map f xs)*
⟨*proof*⟩

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

lemma *listsum*[*code-unfold*]: *listsum xs* = *foldl* (*op* +) *0 xs*
⟨*proof*⟩

lemma *distinct-listsum-conv-Setsum*:
distinct xs ⟹ *listsum xs* = *Setsum*(*set xs*)
⟨*proof*⟩

lemma *listsum-eq-0-nat-iff-nat*[*simp*]:
listsum ns = (*0*::*nat*) ⟷ (∀ *n* ∈ *set ns*. *n* = *0*)
⟨*proof*⟩

lemma *elem-le-listsum-nat*: *k*<*size ns* ⟹ *ns*!*k* <= *listsum*(*ns*::*nat list*)
⟨*proof*⟩

lemma *listsum-update-nat*: *k*<*size ns* ⟹
listsum (*ns*[*k* := (*n*::*nat*)]) = *listsum ns* + *n* − *ns*!*k*
⟨*proof*⟩

Some syntactic sugar for summing a function over a list:

syntax
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3SUM -<−-. -*) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*∑ *-←-. -*) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*∑ *-←-. -*) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM x<−*xs. b* == *CONST listsum* (*CONST map* (%*x. b*) *xs*)
∑ *x*←*xs. b* == *CONST listsum* (*CONST map* (%*x. b*) *xs*)

lemma *listsum-triv*: (∑ *x*←*xs. r*) = *of-nat* (*length xs*) ∗ *r*
⟨*proof*⟩

lemma *listsum-0* [*simp*]: (∑ *x*←*xs. 0*) = *0*
⟨*proof*⟩

For non-Abelian groups *xs* needs to be reversed on one side:

lemma *uminus-listsum-map*:
fixes *f* :: *'a* ⇒ *'b*::*ab-group-add*
shows − *listsum* (*map f xs*) = (*listsum* (*map* (*uminus o f*) *xs*))
⟨*proof*⟩

lemma *listsum-addf*:
fixes *f g* :: *'a* ⇒ *'b*::*comm-monoid-add*
shows (∑ *x*←*xs. f x* + *g x*) = *listsum* (*map f xs*) + *listsum* (*map g xs*)
⟨*proof*⟩

lemma *listsum-subtractf*:

fixes *f g* :: *'a* ⇒ *'b*::*ab-group-add*
shows $(\sum x{\leftarrow}xs.\ f\ x - g\ x) = listsum\ (map\ f\ xs) - listsum\ (map\ g\ xs)$
⟨*proof*⟩

lemma *listsum-const-mult*:
fixes *f* :: *'a* ⇒ *'b*::*semiring-0*
shows $(\sum x{\leftarrow}xs.\ c * f\ x) = c * (\sum x{\leftarrow}xs.\ f\ x)$
⟨*proof*⟩

lemma *listsum-mult-const*:
fixes *f* :: *'a* ⇒ *'b*::*semiring-0*
shows $(\sum x{\leftarrow}xs.\ f\ x * c) = (\sum x{\leftarrow}xs.\ f\ x) * c$
⟨*proof*⟩

lemma *listsum-abs*:
fixes *xs* :: *'a*::*ordered-ab-group-add-abs list*
shows $|listsum\ xs| \leq listsum\ (map\ abs\ xs)$
⟨*proof*⟩

lemma *listsum-mono*:
fixes *f g* :: *'a* ⇒ *'b*::{*comm-monoid-add*, *ordered-ab-semigroup-add*}
shows $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\sum x{\leftarrow}xs.\ f\ x) \leq (\sum x{\leftarrow}xs.\ g\ x)$
⟨*proof*⟩

43.1.20 *upt*

lemma *upt-rec*[*code*]: [*i*..<*j*] = (*if i*<*j then i*#[*Suc i*..<*j*] *else* [])
— simp does not terminate!
⟨*proof*⟩

lemmas *upt-rec-number-of*[*simp*] = *upt-rec*[*of number-of m number-of n*, *standard*]

lemma *upt-conv-Nil* [*simp*]: *j* <= *i* ==> [*i*..<*j*] = []
⟨*proof*⟩

lemma *upt-eq-Nil-conv*[*simp*]: ([*i*..<*j*] = []) = (*j* = *0* ∨ *j* <= *i*)
⟨*proof*⟩

lemma *upt-eq-Cons-conv*:
([*i*..<*j*] = *x*#*xs*) = (*i* < *j* & *i* = *x* & [*i*+*1*..<*j*] = *xs*)
⟨*proof*⟩

lemma *upt-Suc-append*: *i* <= *j* ==> [*i*..<(*Suc j*)] = [*i*..<*j*]@[*j*]
— Only needed if *upt-Suc* is deleted from the simpset.
⟨*proof*⟩

lemma *upt-conv-Cons*: *i* < *j* ==> [*i*..<*j*] = *i* # [*Suc i*..<*j*]
⟨*proof*⟩

lemma *upt-add-eq-append*: $i<=j$ ==> $[i..<j+k] = [i..<j]@[j..<j+k]$
— LOOPS as a simprule, since $j <= j$.
⟨*proof*⟩

lemma *length-upt* [*simp*]: *length* $[i..<j] = j - i$
⟨*proof*⟩

lemma *nth-upt* [*simp*]: $i + k < j$ ==> $[i..<j] ! k = i + k$
⟨*proof*⟩

lemma *hd-upt*[*simp*]: $i < j \Longrightarrow hd[i..<j] = i$
⟨*proof*⟩

lemma *last-upt*[*simp*]: $i < j \Longrightarrow last[i..<j] = j - 1$
⟨*proof*⟩

lemma *take-upt* [*simp*]: $i+m <= n$ ==> *take* m $[i..<n] = [i..<i+m]$
⟨*proof*⟩

lemma *drop-upt*[*simp*]: *drop* m $[i..<j] = [i+m..<j]$
⟨*proof*⟩

lemma *map-Suc-upt*: *map Suc* $[m..<n] = [Suc\ m..<Suc\ n]$
⟨*proof*⟩

lemma *nth-map-upt*: $i < n-m$ ==> (*map f* $[m..<n]$) ! $i = f(m+i)$
⟨*proof*⟩

lemma *nth-take-lemma*:
$k <= length\ xs$ ==> $k <= length\ ys$ ==>
(!!i. $i < k$ --> $xs!i = ys!i$) ==> *take k xs* = *take k ys*
⟨*proof*⟩

lemma *nth-equalityI*:
[| *length xs* = *length ys*; *ALL* $i <$ *length xs*. $xs!i = ys!i$ |] ==> $xs = ys$
⟨*proof*⟩

lemma *map-nth*:
map (λi. *xs* ! i) $[0..<length\ xs] = xs$
⟨*proof*⟩

lemma *list-all2-antisym*:
⟦ ($\bigwedge x\ y$. ⟦$P\ x\ y$; $Q\ y\ x$⟧ ⟹ $x = y$); *list-all2 P xs ys*; *list-all2 Q ys xs* ⟧
⟹ $xs = ys$
⟨*proof*⟩

lemma *take-equalityI*: ($\forall i$. *take* i *xs* = *take* i *ys*) ==> $xs = ys$

— The famous take-lemma.
⟨*proof*⟩

lemma *take-Cons'*:
take n (x # xs) = (if n = 0 then [] else x # take (n − 1) xs)
⟨*proof*⟩

lemma *drop-Cons'*:
drop n (x # xs) = (if n = 0 then x # xs else drop (n − 1) xs)
⟨*proof*⟩

lemma *nth-Cons'*: *(x # xs)!n = (if n = 0 then x else xs!(n − 1))*
⟨*proof*⟩

lemmas *take-Cons-number-of = take-Cons'*[*of number-of v*,*standard*]
lemmas *drop-Cons-number-of = drop-Cons'*[*of number-of v*,*standard*]
lemmas *nth-Cons-number-of = nth-Cons'*[*of - - number-of v*,*standard*]

declare *take-Cons-number-of* [*simp*]
drop-Cons-number-of [*simp*]
nth-Cons-number-of [*simp*]

43.1.21 *upto*: interval-list on *int*

function *upto* :: *int* ⇒ *int* ⇒ *int list* ((*1*[-../-])) **where**
upto i j = (if i ≤ j then i # [i+1..j] else [])
⟨*proof*⟩
termination
⟨*proof*⟩

declare *upto.simps*[*code*, *simp del*]

lemmas *upto-rec-number-of*[*simp*] =
upto.simps[*of number-of m number-of n*, *standard*]

lemma *upto-empty*[*simp*]: *j < i ⟹ [i..j] = []*
⟨*proof*⟩

lemma *set-upto*[*simp*]: *set*[*i..j*] = {*i..j*}
⟨*proof*⟩

43.1.22 *distinct* and *remdups*

lemma *distinct-append* [*simp*]:
distinct (xs @ ys) = (distinct xs ∧ distinct ys ∧ set xs ∩ set ys = {})
⟨*proof*⟩

lemma *distinct-rev*[*simp*]: *distinct*(*rev xs*) = *distinct xs*
⟨*proof*⟩

lemma *set-remdups* [*simp*]: *set* (*remdups xs*) = *set xs*
⟨*proof*⟩

lemma *distinct-remdups* [*iff*]: *distinct* (*remdups xs*)
⟨*proof*⟩

lemma *distinct-remdups-id*: *distinct xs* ==> *remdups xs* = *xs*
⟨*proof*⟩

lemma *remdups-id-iff-distinct* [*simp*]: *remdups xs* = *xs* ⟷ *distinct xs*
⟨*proof*⟩

lemma *finite-distinct-list*: *finite A* ⟹ *EX xs*. *set xs* = *A* & *distinct xs*
⟨*proof*⟩

lemma *remdups-eq-nil-iff* [*simp*]: (*remdups x* = []) = (*x* = [])
⟨*proof*⟩

lemma *remdups-eq-nil-right-iff* [*simp*]: ([] = *remdups x*) = (*x* = [])
⟨*proof*⟩

lemma *length-remdups-leq*[*iff*]: *length*(*remdups xs*) <= *length xs*
⟨*proof*⟩

lemma *length-remdups-eq*[*iff*]:
(*length* (*remdups xs*) = *length xs*) = (*remdups xs* = *xs*)
⟨*proof*⟩

lemma *remdups-filter*: *remdups*(*filter P xs*) = *filter P* (*remdups xs*)
⟨*proof*⟩

lemma *distinct-map*:
distinct(*map f xs*) = (*distinct xs* & *inj-on f* (*set xs*))
⟨*proof*⟩

lemma *distinct-filter* [*simp*]: *distinct xs* ==> *distinct* (*filter P xs*)
⟨*proof*⟩

lemma *distinct-upt*[*simp*]: *distinct*[*i*..<*j*]
⟨*proof*⟩

lemma *distinct-upto*[*simp*]: *distinct*[*i*..*j*]
⟨*proof*⟩

lemma *distinct-take*[*simp*]: *distinct xs* ⟹ *distinct* (*take i xs*)
⟨*proof*⟩

lemma *distinct-drop*[*simp*]: *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*⟩

lemma *distinct-concat*:
assumes *distinct xs*
and $\bigwedge ys.\ ys \in set\ xs \Longrightarrow distinct\ ys$
and $\bigwedge ys\ zs.\ [\![ys \in set\ xs\ ;\ zs \in set\ xs\ ;\ ys \neq zs\ ]\!] \Longrightarrow set\ ys \cap set\ zs = \{\}$
shows *distinct* (*concat xs*)
⟨*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* ≠ *xs*!*j*)
⟨*proof*⟩

lemma *nth-eq-iff-index-eq*:
$[\![$ *distinct xs*; *i* < *length xs*; *j* < *length xs* $]\!] \Longrightarrow$ (*xs*!*i* = *xs*!*j*) = (*i* = *j*)
⟨*proof*⟩

lemma *distinct-card*: *distinct xs* ==> *card* (*set xs*) = *size xs*
⟨*proof*⟩

lemma *card-distinct*: *card* (*set xs*) = *size xs* ==> *distinct xs*
⟨*proof*⟩

lemma *not-distinct-decomp*: ~ *distinct ws* ==> *EX xs ys zs y*. *ws* = *xs*@[*y*]@*ys*@[*y*]@*zs*
⟨*proof*⟩

lemma *length-remdups-concat*:
length(*remdups*(*concat xss*)) = *card*($\bigcup xs \in set\ xss.\ set\ xs$)
⟨*proof*⟩

lemma *length-remdups-card-conv*: *length*(*remdups xs*) = *card*(*set xs*)
⟨*proof*⟩

lemma *remdups-remdups*:
remdups (*remdups xs*) = *remdups xs*
⟨*proof*⟩

lemma *distinct-butlast*:
assumes *xs* ≠ [] **and** *distinct xs*
shows *distinct* (*butlast xs*)
⟨*proof*⟩

43.1.23 *insert*

lemma *in-set-insert* [*simp*]:
x ∈ *set xs* ⟹ *List.insert x xs* = *xs*
⟨*proof*⟩

lemma *not-in-set-insert* [*simp*]:
x ∉ *set xs* ⟹ *List.insert x xs* = *x* # *xs*
⟨*proof*⟩

lemma *insert-Nil* [*simp*]:
List.insert x [] = [*x*]
⟨*proof*⟩

lemma *set-insert* [*simp*]:
set (*List.insert x xs*) = *insert x* (*set xs*)
⟨*proof*⟩

lemma *distinct-insert* [*simp*]:
distinct xs ⟹ *distinct* (*List.insert x xs*)
⟨*proof*⟩

lemma *insert-remdups*:
List.insert x (*remdups xs*) = *remdups* (*List.insert x xs*)
⟨*proof*⟩

lemma *distinct-induct* [*consumes 1*, *case-names Nil insert*]:
assumes *distinct xs*
assumes *P* []
assumes *insert*: ⋀*x xs*. *distinct xs* ⟹ *x* ∉ *set xs*
⟹ *P xs* ⟹ *P* (*List.insert x xs*)
shows *P xs*
⟨*proof*⟩

43.1.24 *remove1*

lemma *remove1-append*:
remove1 x (*xs* @ *ys*) =
(*if x* ∈ *set xs then remove1 x xs* @ *ys else xs* @ *remove1 x ys*)
⟨*proof*⟩

lemma *remove1-commute*: *remove1 x* (*remove1 y zs*) = *remove1 y* (*remove1 x zs*)
⟨*proof*⟩

lemma *in-set-remove1*[*simp*]:
a ≠ *b* ⟹ *a* : *set*(*remove1 b xs*) = (*a* : *set xs*)
⟨*proof*⟩

lemma *set-remove1-subset*: *set*(*remove1 x xs*) <= *set xs*
⟨*proof*⟩

lemma *set-remove1-eq* [*simp*]: *distinct xs ==> set*(*remove1 x xs*) = *set xs* − {*x*}
⟨*proof*⟩

lemma *length-remove1*:
length(*remove1 x xs*) = (*if x* : *set xs then length xs* − *1 else length xs*)
⟨*proof*⟩

lemma *remove1-filter-not*[*simp*]:
¬ *P x* ⟹ *remove1 x* (*filter P xs*) = *filter P xs*
⟨*proof*⟩

lemma *notin-set-remove1*[*simp*]: *x* ~: *set xs ==> x* ~: *set*(*remove1 y xs*)
⟨*proof*⟩

lemma *distinct-remove1*[*simp*]: *distinct xs ==> distinct*(*remove1 x xs*)
⟨*proof*⟩

lemma *remove1-remdups*:
distinct xs ⟹ *remove1 x* (*remdups xs*) = *remdups* (*remove1 x xs*)
⟨*proof*⟩

lemma *remove1-idem*:
assumes *x* ∉ *set xs*
shows *remove1 x xs* = *xs*
⟨*proof*⟩

43.1.25 *removeAll*

lemma *removeAll-filter-not-eq*:
removeAll x = *filter* (λ*y*. *x* ≠ *y*)
⟨*proof*⟩

lemma *removeAll-append*[*simp*]:
removeAll x (*xs* @ *ys*) = *removeAll x xs* @ *removeAll x ys*
⟨*proof*⟩

lemma *set-removeAll*[*simp*]: *set*(*removeAll x xs*) = *set xs* − {*x*}
⟨*proof*⟩

lemma *removeAll-id*[*simp*]: *x* ∉ *set xs* ⟹ *removeAll x xs* = *xs*
⟨*proof*⟩

lemma *removeAll-filter-not*[*simp*]:
¬ *P x* ⟹ *removeAll x* (*filter P xs*) = *filter P xs*
⟨*proof*⟩

lemma *distinct-removeAll*:
distinct xs ⟹ *distinct* (*removeAll x xs*)
⟨*proof*⟩

lemma *distinct-remove1-removeAll*:
distinct xs ==> *remove1 x xs* = *removeAll x xs*
⟨*proof*⟩

lemma *map-removeAll-inj-on*: *inj-on f* (*insert x* (*set xs*)) ⟹
map f (*removeAll x xs*) = *removeAll* (*f x*) (*map f xs*)
⟨*proof*⟩

lemma *map-removeAll-inj*: *inj f* ⟹
map f (*removeAll x xs*) = *removeAll* (*f x*) (*map f xs*)
⟨*proof*⟩

43.1.26 *replicate*

lemma *length-replicate* [*simp*]: *length* (*replicate n x*) = *n*
⟨*proof*⟩

lemma *Ex-list-of-length*: ∃ *xs*. *length xs* = *n*
⟨*proof*⟩

lemma *map-replicate* [*simp*]: *map f* (*replicate n x*) = *replicate n* (*f x*)
⟨*proof*⟩

lemma *map-replicate-const*:
map (λ *x*. *k*) *lst* = *replicate* (*length lst*) *k*
⟨*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*⟩

Courtesy of Andreas Lochbihler:

lemma *filter-replicate*:
filter P (*replicate n x*) = (*if P x then replicate n x else* [])

⟨*proof*⟩

lemma *hd-replicate* [*simp*]: $n \neq 0 \Longrightarrow hd\ (replicate\ n\ x) = x$
⟨*proof*⟩

lemma *tl-replicate* [*simp*]: $n \neq 0 \Longrightarrow tl\ (replicate\ n\ x) = replicate\ (n - 1)\ x$
⟨*proof*⟩

lemma *last-replicate* [*simp*]: $n \neq 0 \Longrightarrow last\ (replicate\ n\ x) = x$
⟨*proof*⟩

lemma *nth-replicate*[*simp*]: $i < n \Longrightarrow (replicate\ n\ x)!i = x$
⟨*proof*⟩

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[*simp*]: $take\ i\ (replicate\ k\ x) = replicate\ (min\ i\ k)\ x$
⟨*proof*⟩

lemma *drop-replicate*[*simp*]: $drop\ i\ (replicate\ k\ x) = replicate\ (k-i)\ x$
⟨*proof*⟩

lemma *set-replicate-Suc*: $set\ (replicate\ (Suc\ n)\ x) = \{x\}$
⟨*proof*⟩

lemma *set-replicate* [*simp*]: $n \neq 0 \Longrightarrow set\ (replicate\ n\ x) = \{x\}$
⟨*proof*⟩

lemma *set-replicate-conv-if*: $set\ (replicate\ n\ x) = (if\ n = 0\ then\ \{\}\ else\ \{x\})$
⟨*proof*⟩

lemma *in-set-replicateD*: $x : set\ (replicate\ n\ y) \Longrightarrow x = y$
⟨*proof*⟩

lemma *replicate-append-same*:
$replicate\ i\ x\ @\ [x] = x\ \#\ replicate\ i\ x$
⟨*proof*⟩

lemma *map-replicate-trivial*:
$map\ (\lambda i.\ x)\ [0..<i] = replicate\ i\ x$
⟨*proof*⟩

lemma *concat-replicate-trivial*[*simp*]:
$concat\ (replicate\ i\ []) = []$
⟨*proof*⟩

lemma *replicate-empty*[*simp*]: $(replicate\ n\ x = []) \longleftrightarrow n=0$
⟨*proof*⟩

lemma *empty-replicate*[*simp*]: ([] = *replicate n x*) ⟷ *n=0*
⟨*proof*⟩

lemma *replicate-eq-replicate*[*simp*]:
(*replicate m x* = *replicate n y*) ⟷ (*m=n* & (*m≠0* ⟶ *x=y*))
⟨*proof*⟩

43.1.27 *rotate1* and *rotate*

lemma *rotate-simps*[*simp*]: *rotate1* [] = [] ∧ *rotate1* (*x#xs*) = *xs* @ [*x*]
⟨*proof*⟩

lemma *rotate0*[*simp*]: *rotate 0* = *id*
⟨*proof*⟩

lemma *rotate-Suc*[*simp*]: *rotate* (*Suc n*) *xs* = *rotate1*(*rotate n xs*)
⟨*proof*⟩

lemma *rotate-add*:
rotate (*m+n*) = *rotate m o rotate n*
⟨*proof*⟩

lemma *rotate-rotate*: *rotate m* (*rotate n xs*) = *rotate* (*m+n*) *xs*
⟨*proof*⟩

lemma *rotate1-rotate-swap*: *rotate1* (*rotate n xs*) = *rotate n* (*rotate1 xs*)
⟨*proof*⟩

lemma *rotate1-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate1 xs* = *xs*
⟨*proof*⟩

lemma *rotate-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate n xs* = *xs*
⟨*proof*⟩

lemma *rotate1-hd-tl*: *xs* ≠ [] ⟹ *rotate1 xs* = *tl xs* @ [*hd xs*]
⟨*proof*⟩

lemma *rotate-drop-take*:
rotate n xs = *drop* (*n mod length xs*) *xs* @ *take* (*n mod length xs*) *xs*
⟨*proof*⟩

lemma *rotate-conv-mod*: *rotate n xs* = *rotate* (*n mod length xs*) *xs*
⟨*proof*⟩

lemma *rotate-id*[*simp*]: *n mod length xs* = *0* ⟹ *rotate n xs* = *xs*
⟨*proof*⟩

lemma *length-rotate1*[*simp*]: *length*(*rotate1 xs*) = *length xs*
⟨*proof*⟩

lemma *length-rotate*[*simp*]: *length*(*rotate n xs*) = *length xs*
⟨*proof*⟩

lemma *distinct1-rotate*[*simp*]: *distinct*(*rotate1 xs*) = *distinct xs*
⟨*proof*⟩

lemma *distinct-rotate*[*simp*]: *distinct*(*rotate n xs*) = *distinct xs*
⟨*proof*⟩

lemma *rotate-map*: *rotate n* (*map f xs*) = *map f* (*rotate n xs*)
⟨*proof*⟩

lemma *set-rotate1*[*simp*]: *set*(*rotate1 xs*) = *set xs*
⟨*proof*⟩

lemma *set-rotate*[*simp*]: *set*(*rotate n xs*) = *set xs*
⟨*proof*⟩

lemma *rotate1-is-Nil-conv*[*simp*]: (*rotate1 xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *rotate-is-Nil-conv*[*simp*]: (*rotate n xs* = []) = (*xs* = [])
⟨*proof*⟩

lemma *rotate-rev*:
rotate n (*rev xs*) = *rev*(*rotate* (*length xs* − (*n mod length xs*)) *xs*)
⟨*proof*⟩

lemma *hd-rotate-conv-nth*: *xs* ≠ [] ⟹ *hd*(*rotate n xs*) = *xs*!(*n mod length xs*)
⟨*proof*⟩

43.1.28 *sublist* — a generalization of *nth* to sets

lemma *sublist-empty* [*simp*]: *sublist xs* {} = []
⟨*proof*⟩

lemma *sublist-nil* [*simp*]: *sublist* [] *A* = []
⟨*proof*⟩

lemma *length-sublist*:
length(*sublist xs I*) = *card*{*i*. *i* < *length xs* ∧ *i* : *I*}
⟨*proof*⟩

lemma *sublist-shift-lemma-Suc*:
map fst (*filter* (%*p*. *P*(*Suc*(*snd p*))) (*zip xs is*)) =
map fst (*filter* (%*p*. *P*(*snd p*)) (*zip xs* (*map Suc is*)))
⟨*proof*⟩

lemma *sublist-shift-lemma*:
map fst [p<−zip xs [i..<i + length xs] . snd p : A] =
map fst [p<−zip xs [0..<length xs] . snd p + i : A]
⟨*proof*⟩

lemma *sublist-append*:
sublist (l @ l′) A = sublist l A @ sublist l′ {j. j + length l : A}
⟨*proof*⟩

lemma *sublist-Cons*:
sublist (x # l) A = (if 0:A then [x] else []) @ sublist l {j. Suc j : A}
⟨*proof*⟩

lemma *set-sublist*: *set(sublist xs I) = {xs!i|i. i<size xs ∧ i ∈ I}*
⟨*proof*⟩

lemma *set-sublist-subset*: *set(sublist xs I) ⊆ set xs*
⟨*proof*⟩

lemma *notin-set-sublistI*[*simp*]: *x ∉ set xs ⟹ x ∉ set(sublist xs I)*
⟨*proof*⟩

lemma *in-set-sublistD*: *x ∈ set(sublist xs I) ⟹ x ∈ set xs*
⟨*proof*⟩

lemma *sublist-singleton* [*simp*]: *sublist [x] A = (if 0 : A then [x] else [])*
⟨*proof*⟩

lemma *distinct-sublistI*[*simp*]: *distinct xs ⟹ distinct(sublist xs I)*
⟨*proof*⟩

lemma *sublist-upt-eq-take* [*simp*]: *sublist l {..<n} = take n l*
⟨*proof*⟩

lemma *filter-in-sublist*:
distinct xs ⟹ filter (%x. x ∈ set(sublist xs s)) xs = sublist xs s
⟨*proof*⟩

43.1.29 *splice*

lemma *splice-Nil2* [*simp*, *code*]:
splice xs [] = xs
⟨*proof*⟩

lemma *splice-Cons-Cons* [*simp*, *code*]:
splice (x#xs) (y#ys) = x # y # splice xs ys
⟨*proof*⟩

declare *splice.simps*(*2*) [*simp del*, *code del*]

lemma *length-splice*[*simp*]: *length*(*splice xs ys*) = *length xs* + *length ys*
⟨*proof*⟩

43.1.30 Transpose

function *transpose* **where**
transpose [] = [] |
transpose ([] # *xss*) = *transpose xss* |
transpose ((*x*#*xs*) # *xss*) =
(*x* # [*h*. (*h*#*t*) ← *xss*]) # *transpose* (*xs* # [*t*. (*h*#*t*) ← *xss*])
⟨*proof*⟩

lemma *transpose-aux-filter-head*:
concat (*map* (*list-case* [] (λ*h t*. [*h*])) *xss*) =
map (λ*xs*. *hd xs*) [*ys*←*xss* . *ys* ≠ []]
⟨*proof*⟩

lemma *transpose-aux-filter-tail*:
concat (*map* (*list-case* [] (λ*h t*. [*t*])) *xss*) =
map (λ*xs*. *tl xs*) [*ys*←*xss* . *ys* ≠ []]
⟨*proof*⟩

lemma *transpose-aux-max*:
max (*Suc* (*length xs*)) (*foldr* (λ*xs*. *max* (*length xs*)) *xss* *0*) =
Suc (*max* (*length xs*) (*foldr* (λ*x*. *max* (*length x* − *Suc* *0*)) [*ys*←*xss* . *ys*≠[]] *0*))
(**is** *max* - *?foldB* = *Suc* (*max* - *?foldA*))
⟨*proof*⟩

termination *transpose*
⟨*proof*⟩

lemma *transpose-empty*: (*transpose xs* = []) ⟷ (∀ *x* ∈ *set xs*. *x* = [])
⟨*proof*⟩

lemma *length-transpose*:
fixes *xs* :: *'a list list*
shows *length* (*transpose xs*) = *foldr* (λ*xs*. *max* (*length xs*)) *xs* *0*
⟨*proof*⟩

lemma *nth-transpose*:
fixes *xs* :: *'a list list*
assumes *i* < *length* (*transpose xs*)
shows *transpose xs* ! *i* = *map* (λ*xs*. *xs* ! *i*) [*ys* ← *xs*. *i* < *length ys*]
⟨*proof*⟩

lemma *transpose-map-map*:

transpose (*map* (*map f*) *xs*) = *map* (*map f*) (*transpose xs*)
⟨*proof*⟩

43.1.31 (In)finiteness

lemma *finite-maxlen*:
finite (*M*::'*a list set*) ==> *EX n*. *ALL s*:*M*. *size s* < *n*
⟨*proof*⟩

lemma *finite-lists-length-eq*:
assumes *finite A*
shows *finite* {*xs*. *set xs* ⊆ *A* ∧ *length xs* = *n*} (**is** *finite* (*?S n*))
⟨*proof*⟩

lemma *finite-lists-length-le*:
assumes *finite A* **shows** *finite* {*xs*. *set xs* ⊆ *A* ∧ *length xs* ≤ *n*}
(**is** *finite ?S*)
⟨*proof*⟩

lemma *infinite-UNIV-listI*: ~ *finite*(*UNIV*::'*a list set*)
⟨*proof*⟩

43.2 Sorting

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context *linorder*
begin

lemma *length-insert*[*simp*] : *length* (*insort-key f x xs*) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *insort-left-comm*:
insort x (*insort y xs*) = *insort y* (*insort x xs*)
⟨*proof*⟩

lemma *fun-left-comm-insort*:
fun-left-comm insort
⟨*proof*⟩

lemma *sort-key-simps* [*simp*]:
sort-key f [] = []
sort-key f (*x*#*xs*) = *insort-key f x* (*sort-key f xs*)
⟨*proof*⟩

lemma *sort-foldl-insort*:
sort xs = foldl (λys x. insort x ys) [] xs
⟨*proof*⟩

lemma *length-sort*[*simp*]: *length (sort-key f xs) = length xs*
⟨*proof*⟩

lemma *sorted-Cons*: *sorted (x#xs) = (sorted xs & (ALL y:set xs. x <= y))*
⟨*proof*⟩

lemma *sorted-append*:
sorted (xs@ys) = (sorted xs & sorted ys & (∀x ∈ set xs. ∀y ∈ set ys. x≤y))
⟨*proof*⟩

lemma *sorted-nth-mono*:
sorted xs ⟹ i ≤ j ⟹ j < length xs ⟹ xs!i ≤ xs!j
⟨*proof*⟩

lemma *sorted-rev-nth-mono*:
sorted (rev xs) ⟹ i ≤ j ⟹ j < length xs ⟹ xs!j ≤ xs!i
⟨*proof*⟩

lemma *sorted-nth-monoI*:
(⋀ i j. ⟦ i ≤ j ; j < length xs ⟧ ⟹ xs ! i ≤ xs ! j) ⟹ sorted xs
⟨*proof*⟩

lemma *sorted-equals-nth-mono*:
sorted xs = (∀j < length xs. ∀i ≤ j. xs ! i ≤ xs ! j)
⟨*proof*⟩

lemma *set-insort*: *set(insort-key f x xs) = insert x (set xs)*
⟨*proof*⟩

lemma *set-sort*[*simp*]: *set(sort-key f xs) = set xs*
⟨*proof*⟩

lemma *distinct-insort*: *distinct (insort-key f x xs) = (x ∉ set xs ∧ distinct xs)*
⟨*proof*⟩

lemma *distinct-sort*[*simp*]: *distinct (sort-key f xs) = distinct xs*
⟨*proof*⟩

lemma *sorted-insort-key*: *sorted (map f (insort-key f x xs)) = sorted (map f xs)*
⟨*proof*⟩

lemma *sorted-insort*: *sorted (insort x xs) = sorted xs*
⟨*proof*⟩

theorem *sorted-sort-key*[*simp*]: *sorted (map f (sort-key f xs))*

⟨*proof*⟩

theorem *sorted-sort*[*simp*]: *sorted* (*sort xs*)
⟨*proof*⟩

lemma *sorted-butlast*:
assumes *xs* ≠ [] **and** *sorted xs*
shows *sorted* (*butlast xs*)
⟨*proof*⟩

lemma *insort-not-Nil* [*simp*]:
insort-key f a xs ≠ []
⟨*proof*⟩

lemma *insort-is-Cons*: ∀ *x*∈*set xs*. *f a* ≤ *f x* ⟹ *insort-key f a xs* = *a* # *xs*
⟨*proof*⟩

lemma *sorted-remove1*: *sorted xs* ⟹ *sorted* (*remove1 a xs*)
⟨*proof*⟩

lemma *insort-key-remove1*: ⟦ *a* ∈ *set xs*; *sorted* (*map f xs*) ; *inj-on f* (*set xs*) ⟧
⟹ *insort-key f a* (*remove1 a xs*) = *xs*
⟨*proof*⟩

lemma *insort-remove1*: ⟦ *a* ∈ *set xs*; *sorted xs* ⟧ ⟹ *insort a* (*remove1 a xs*) = *xs*
⟨*proof*⟩

lemma *sorted-remdups*[*simp*]:
sorted l ⟹ *sorted* (*remdups l*)
⟨*proof*⟩

lemma *sorted-distinct-set-unique*:
assumes *sorted xs distinct xs sorted ys distinct ys set xs* = *set ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *map-sorted-distinct-set-unique*:
assumes *inj-on f* (*set xs* ∪ *set ys*)
assumes *sorted* (*map f xs*) *distinct* (*map f xs*)
sorted (*map f ys*) *distinct* (*map f ys*)
assumes *set xs* = *set ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *finite-sorted-distinct-unique*:
shows *finite A* ⟹ *EX*! *xs*. *set xs* = *A* & *sorted xs* & *distinct xs*
⟨*proof*⟩

lemma *sorted-take*:

sorted xs $\Longrightarrow$ *sorted* (*take n xs*)
⟨*proof*⟩

lemma *sorted-drop*:
sorted xs $\Longrightarrow$ *sorted* (*drop n xs*)
⟨*proof*⟩

lemma *sorted-dropWhile*: *sorted xs* $\Longrightarrow$ *sorted* (*dropWhile P xs*)
⟨*proof*⟩

lemma *sorted-takeWhile*: *sorted xs* $\Longrightarrow$ *sorted* (*takeWhile P xs*)
⟨*proof*⟩

lemma *sorted-filter*:
sorted (*map f xs*) $\Longrightarrow$ *sorted* (*map f* (*filter P xs*))
⟨*proof*⟩

lemma *foldr-max-sorted*:
assumes *sorted* (*rev xs*)
shows *foldr max xs y* = (*if xs* = [] *then y else max* (*xs* ! *0*) *y*)
⟨*proof*⟩

lemma *filter-equals-takeWhile-sorted-rev*:
assumes *sorted*: *sorted* (*rev* (*map f xs*))
shows [*x* ← *xs*. *t* < *f x*] = *takeWhile* (λ *x*. *t* < *f x*) *xs*
(**is** *filter ?P xs* = *?tW*)
⟨*proof*⟩

lemma *set-insort-insert*:
set (*insort-insert x xs*) = *insert x* (*set xs*)
⟨*proof*⟩

lemma *distinct-insort-insert*:
assumes *distinct xs*
shows *distinct* (*insort-insert x xs*)
⟨*proof*⟩

lemma *sorted-insort-insert*:
assumes *sorted xs*
shows *sorted* (*insort-insert x xs*)
⟨*proof*⟩

lemma *filter-insort-key-triv*:
¬ *P x* $\Longrightarrow$ *filter P* (*insort-key f x xs*) = *filter P xs*
⟨*proof*⟩

lemma *filter-insort-key*:
sorted (*map f xs*) $\Longrightarrow$ *P x* $\Longrightarrow$ *filter P* (*insort-key f x xs*) = *insort-key f x* (*filter P xs*)

⟨*proof*⟩

lemma *filter-sort-key*:
filter P (*sort-key f xs*) = *sort-key f* (*filter P xs*)
⟨*proof*⟩

lemma *sorted-same* [*simp*]:
sorted [*x*←*xs*. *x* = *f xs*]
⟨*proof*⟩

lemma *remove1-insort* [*simp*]:
remove1 x (*insort x xs*) = *xs*
⟨*proof*⟩

end

lemma *sorted-upt*[*simp*]: *sorted*[*i*..<*j*]
⟨*proof*⟩

lemma *sorted-upto*[*simp*]: *sorted*[*i*..*j*]
⟨*proof*⟩

43.2.1 *transpose* on sorted lists

lemma *sorted-transpose*[*simp*]:
shows *sorted* (*rev* (*map length* (*transpose xs*)))
⟨*proof*⟩

lemma *transpose-max-length*:
foldr (λ*xs*. *max* (*length xs*)) (*transpose xs*) *0* = *length* [*x* ← *xs*. *x* ≠ []]
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *length-transpose-sorted*:
fixes *xs* :: *'a list list*
assumes *sorted*: *sorted* (*rev* (*map length xs*))
shows *length* (*transpose xs*) = (*if xs* = [] *then 0 else length* (*xs* ! *0*))
⟨*proof*⟩

lemma *nth-nth-transpose-sorted*[*simp*]:
fixes *xs* :: *'a list list*
assumes *sorted*: *sorted* (*rev* (*map length xs*))
and *i*: *i* < *length* (*transpose xs*)
and *j*: *j* < *length* [*ys* ← *xs*. *i* < *length ys*]
shows *transpose xs* ! *i* ! *j* = *xs* ! *j* ! *i*
⟨*proof*⟩

lemma *transpose-column-length*:
fixes *xs* :: *'a list list*

assumes *sorted*: *sorted* (*rev* (*map length xs*)) **and** $i < length\ xs$
shows *length* (*filter* ($\lambda ys.\ i < length\ ys$) (*transpose xs*)) = *length* (*xs* ! *i*)
⟨*proof*⟩

lemma *transpose-column*:
fixes *xs* :: *'a list list*
assumes *sorted*: *sorted* (*rev* (*map length xs*)) **and** $i < length\ xs$
shows *map* ($\lambda ys.\ ys$! *i*) (*filter* ($\lambda ys.\ i < length\ ys$) (*transpose xs*))
= *xs* ! *i* (**is** *?R* = -)
⟨*proof*⟩

lemma *transpose-transpose*:
fixes *xs* :: *'a list list*
assumes *sorted*: *sorted* (*rev* (*map length xs*))
shows *transpose* (*transpose xs*) = *takeWhile* ($\lambda x.\ x \neq []$) *xs* (**is** *?L* = *?R*)
⟨*proof*⟩

theorem *transpose-rectangle*:
assumes $xs = [] \Longrightarrow n = 0$
assumes *rect*: $\bigwedge i.\ i < length\ xs \Longrightarrow length\ (xs\ !\ i) = n$
shows *transpose xs* = *map* ($\lambda\ i.$ *map* ($\lambda\ j.$ *xs* ! *j* ! *i*) [*0*..<*length xs*]) [*0*..<*n*]
(**is** *?trans* = *?map*)
⟨*proof*⟩

43.2.2 *sorted-list-of-set*

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

context *linorder*
begin

definition *sorted-list-of-set* :: *'a set* ⇒ *'a list* **where**
sorted-list-of-set = *Finite-Set.fold insort* []

lemma *sorted-list-of-set-empty* [*simp*]:
sorted-list-of-set {} = []
⟨*proof*⟩

lemma *sorted-list-of-set-insert* [*simp*]:
assumes *finite A*
shows *sorted-list-of-set* (*insert x A*) = *insort x* (*sorted-list-of-set* ($A - \{x\}$))
⟨*proof*⟩

lemma *sorted-list-of-set* [*simp*]:
finite A $\Longrightarrow$ *set* (*sorted-list-of-set A*) = $A \wedge$ *sorted* (*sorted-list-of-set A*)
$\wedge$ *distinct* (*sorted-list-of-set A*)
⟨*proof*⟩

lemma *sorted-list-of-set-sort-remdups*:
sorted-list-of-set (*set xs*) = *sort* (*remdups xs*)
⟨*proof*⟩

lemma *sorted-list-of-set-remove*:
assumes *finite A*
shows *sorted-list-of-set* (*A* − {*x*}) = *remove1 x* (*sorted-list-of-set A*)
⟨*proof*⟩

end

lemma *sorted-list-of-set-range* [*simp*]:
sorted-list-of-set {*m*..<*n*} = [*m*..<*n*]
⟨*proof*⟩

43.2.3 *lists*: the list-forming operator over sets

inductive-set
lists :: *'a set* => *'a list set*
for *A* :: *'a set*
where
Nil [*intro*!]: []: *lists A*
| *Cons* [*intro*!,*no-atp*]: [| *a*: *A*; *l*: *lists A*|] ==> *a*#*l* : *lists A*

inductive-cases *listsE* [*elim*!,*no-atp*]: *x*#*l* : *lists A*
inductive-cases *listspE* [*elim*!,*no-atp*]: *listsp A* (*x* # *l*)

lemma *listsp-mono* [*mono*]: *A* ≤ *B* ==> *listsp A* ≤ *listsp B*
⟨*proof*⟩

lemmas *lists-mono* = *listsp-mono* [*to-set pred-subset-eq*]

lemma *listsp-infI*:
assumes *l*: *listsp A l* **shows** *listsp B l* ==> *listsp* (*inf A B*) *l* ⟨*proof*⟩

lemmas *lists-IntI* = *listsp-infI* [*to-set*]

lemma *listsp-inf-eq* [*simp*]: *listsp* (*inf A B*) = *inf* (*listsp A*) (*listsp B*)
⟨*proof*⟩

lemmas *listsp-conj-eq* [*simp*] = *listsp-inf-eq* [*simplified inf-fun-eq inf-bool-eq*]

lemmas *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set pred-equals-eq*]

lemma *append-in-listsp-conv* [*iff*]:
(*listsp A* (*xs* @ *ys*)) = (*listsp A xs* ∧ *listsp A ys*)
⟨*proof*⟩

lemmas *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

lemma *in-listsp-conv-set*: $(listsp\ A\ xs) = (\forall x \in set\ xs.\ A\ x)$
— eliminate *listsp* in favour of *set*
⟨*proof*⟩

lemmas *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest*!,*no-atp*]: $listsp\ A\ xs \Longrightarrow \forall x{\in}set\ xs.\ A\ x$
⟨*proof*⟩

lemmas *in-listsD* [*dest*!,*no-atp*] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro*!,*no-atp*]: $\forall x{\in}set\ xs.\ A\ x \Longrightarrow listsp\ A\ xs$
⟨*proof*⟩

lemmas *in-listsI* [*intro*!,*no-atp*] = *in-listspI* [*to-set*]

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
⟨*proof*⟩

43.2.4 Inductive definition for membership

inductive *ListMem* :: $'a \Rightarrow 'a\ list \Rightarrow bool$
where
  elem: $ListMem\ x\ (x\ \#\ xs)$
| *insert*: $ListMem\ x\ xs \Longrightarrow ListMem\ x\ (y\ \#\ xs)$

lemma *ListMem-iff*: $(ListMem\ x\ xs) = (x \in set\ xs)$
⟨*proof*⟩

43.2.5 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from A and tail drawn from *Xs*.

definition
  set-Cons :: $'a\ set \Rightarrow 'a\ list\ set \Rightarrow 'a\ list\ set$ **where**
  [*code del*]: $set\text{-}Cons\ A\ XS = \{z.\ \exists x\ xs.\ z = x\ \#\ xs \wedge x \in A \wedge xs \in XS\}$

lemma *set-Cons-sing-Nil* [*simp*]: $set\text{-}Cons\ A\ \{[]\} = (\%x.\ [x])\text{`}A$
⟨*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.

primrec
  listset :: $'a\ set\ list \Rightarrow 'a\ list\ set$ **where**
    $listset\ [] = \{[]\}$
| $listset\ (A\ \#\ As) = set\text{-}Cons\ A\ (listset\ As)$

43.3 Relations on Lists

43.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

primrec — The lexicographic ordering for lists of the specified length
lexn :: $('a \times 'a)$ *set* $\Rightarrow$ *nat* $\Rightarrow$ $('a\ list \times 'a\ list)$ *set* **where**
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*}

definition
lex :: $('a \times 'a)$ *set* $\Rightarrow$ $('a\ list \times 'a\ list)$ *set* **where**
[*code del*]: *lex r* = ($\bigcup$ *n*. *lexn r n*) — Holds only between lists of the same length

definition
lenlex :: $('a \times 'a)$ *set* => $('a\ list \times 'a\ list)$ *set* **where**
[*code del*]: *lenlex r* = *inv-image* (*less-than* <∗*lex*∗> *lex r*) (λ*xs*. (*length xs*, *xs*))
— Compares lists by their length and then lexicographically

lemma *wf-lexn*: *wf r* ==> *wf* (*lexn r n*)
⟨*proof*⟩

lemma *lexn-length*:
(*xs*, *ys*) : *lexn r n* ==> *length xs* = *n* ∧ *length ys* = *n*
⟨*proof*⟩

lemma *wf-lex* [*intro*!]: *wf r* ==> *wf* (*lex r*)
⟨*proof*⟩

lemma *lexn-conv*:
lexn r n =
{(*xs*,*ys*). *length xs* = *n* ∧ *length ys* = *n* ∧
(∃ *xys x y xs′ ys′*. *xs*= *xys* @ *x*#*xs′* ∧ *ys*= *xys* @ *y* # *ys′* ∧ (*x*, *y*):*r*)}
⟨*proof*⟩

lemma *lex-conv*:
lex r =
{(*xs*,*ys*). *length xs* = *length ys* ∧
(∃ *xys x y xs′ ys′*. *xs* = *xys* @ *x* # *xs′* ∧ *ys* = *xys* @ *y* # *ys′* ∧ (*x*, *y*):*r*)}
⟨*proof*⟩

lemma *wf-lenlex* [*intro*!]: *wf r* ==> *wf* (*lenlex r*)
⟨*proof*⟩

lemma *lenlex-conv*:
lenlex r = {(*xs*,*ys*). *length xs* < *length ys* |

$$\textit{length } xs = \textit{length } ys \wedge (xs,\ ys) : \textit{lex } r\}$$

⟨*proof*⟩

lemma *Nil-notin-lex* [*iff*]: $([],\ ys) \notin \textit{lex } r$
⟨*proof*⟩

lemma *Nil2-notin-lex* [*iff*]: $(xs,\ []) \notin \textit{lex } r$
⟨*proof*⟩

lemma *Cons-in-lex* [*simp*]:
$((x \# xs,\ y \# ys) : \textit{lex } r) =$
$((x,\ y) : r \wedge \textit{length } xs = \textit{length } ys \mid x = y \wedge (xs,\ ys) : \textit{lex } r)$
⟨*proof*⟩

43.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" ¡ "ab" ¡ "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

definition
lexord :: $('a \times 'a)\ set \Rightarrow ('a\ list \times 'a\ list)\ set$ **where**
[*code del*]: $\textit{lexord } r = \{(x,y).\ \exists\ a\ v.\ y = x\ @\ a \# v \vee$
$(\exists\ u\ a\ b\ v\ w.\ (a,b) \in r \wedge x = u\ @\ (a \# v) \wedge y = u\ @\ (b \# w))\}$

lemma *lexord-Nil-left*[*simp*]: $([],y) \in \textit{lexord } r = (\exists\ a\ x.\ y = a \# x)$
⟨*proof*⟩

lemma *lexord-Nil-right*[*simp*]: $(x,[]) \notin \textit{lexord } r$
⟨*proof*⟩

lemma *lexord-cons-cons*[*simp*]:
$((a \# x,\ b \# y) \in \textit{lexord } r) = ((a,b) \in r \mid (a = b\ \&\ (x,y) \in \textit{lexord } r))$
⟨*proof*⟩

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-append-rightI*: $\exists\ b\ z.\ y = b \# z \Longrightarrow (x,\ x\ @\ y) \in \textit{lexord } r$
⟨*proof*⟩

lemma *lexord-append-left-rightI*:
$(a,b) \in r \Longrightarrow (u\ @\ a \# x,\ u\ @\ b \# y) \in \textit{lexord } r$
⟨*proof*⟩

lemma *lexord-append-leftI*: $(u,v) \in \textit{lexord } r \Longrightarrow (x\ @\ u,\ x\ @\ v) \in \textit{lexord } r$
⟨*proof*⟩

lemma *lexord-append-leftD*:
$[\![\ (x\ @\ u,\ x\ @\ v) \in \textit{lexord } r;\ (!\ a.\ (a,a) \notin r)\ ]\!] \Longrightarrow (u,v) \in \textit{lexord } r$
⟨*proof*⟩

lemma *lexord-take-index-conv*:
$((x,y) : lexord\ r) =$
$((length\ x < length\ y \wedge take\ (length\ x)\ y = x) \vee$
$(\exists i.\ i < min(length\ x)(length\ y)\ \&\ take\ i\ x = take\ i\ y\ \&\ (x!i,y!i) \in r))$
⟨*proof*⟩
lemma *lexord-lex*: $(x,y) \in lex\ r = ((x,y) \in lexord\ r \wedge length\ x = length\ y)$
⟨*proof*⟩

lemma *lexord-irreflexive*: $(!\ x.\ (x,x) \notin r) \Longrightarrow (y,y) \notin lexord\ r$
⟨*proof*⟩

lemma *lexord-trans*:
$[\![\ (x,\ y) \in lexord\ r;\ (y,\ z) \in lexord\ r;\ trans\ r\ ]\!] \Longrightarrow (x,\ z) \in lexord\ r$
⟨*proof*⟩

lemma *lexord-transI*: $trans\ r \Longrightarrow trans\ (lexord\ r)$
⟨*proof*⟩

lemma *lexord-linear*: $(!\ a\ b.\ (a,b) \in r \mid a = b \mid (b,a) \in r) \Longrightarrow (x,y) : lexord\ r \mid x = y \mid (y,x) : lexord\ r$
⟨*proof*⟩

43.4 Lexicographic combination of measure functions

These are useful for termination proofs

definition
$measures\ fs = inv\text{-}image\ (lex\ less\text{-}than)\ (\%a.\ map\ (\%f.\ f\ a)\ fs)$

lemma *wf-measures*[*recdef-wf*, *simp*]: $wf\ (measures\ fs)$
⟨*proof*⟩

lemma *in-measures*[*simp*]:
$(x,\ y) \in measures\ [] = False$
$(x,\ y) \in measures\ (f\ \#\ fs)$
$= (f\ x < f\ y \vee (f\ x = f\ y \wedge (x,\ y) \in measures\ fs))$
⟨*proof*⟩

lemma *measures-less*: $f\ x < f\ y \Longrightarrow (x,\ y) \in measures\ (f\#fs)$
⟨*proof*⟩

lemma *measures-lesseq*: $f\ x <= f\ y \Longrightarrow (x,\ y) \in measures\ fs \Longrightarrow (x,\ y) \in measures\ (f\#fs)$
⟨*proof*⟩

43.4.1 Lifting a Relation on List Elements to the Lists

inductive-set
$listrel :: ('a * 'a)set => ('a\ list * 'a\ list)set$
for $r :: ('a * 'a)set$

where
  Nil: ([],[]) ∈ *listrel r*
| *Cons*: [| (*x*,*y*) ∈ *r*; (*xs*,*ys*) ∈ *listrel r* |] ==> (*x*#*xs*, *y*#*ys*) ∈ *listrel r*

inductive-cases *listrel-Nil1* [*elim*!]: ([],*xs*) ∈ *listrel r*
inductive-cases *listrel-Nil2* [*elim*!]: (*xs*,[]) ∈ *listrel r*
inductive-cases *listrel-Cons1* [*elim*!]: (*y*#*ys*,*xs*) ∈ *listrel r*
inductive-cases *listrel-Cons2* [*elim*!]: (*xs*,*y*#*ys*) ∈ *listrel r*

lemma *listrel-mono*: *r* ⊆ *s* ⟹ *listrel r* ⊆ *listrel s*
⟨*proof*⟩

lemma *listrel-subset*: *r* ⊆ *A* × *A* ⟹ *listrel r* ⊆ *lists A* × *lists A*
⟨*proof*⟩

lemma *listrel-refl-on*: *refl-on A r* ⟹ *refl-on* (*lists A*) (*listrel r*)
⟨*proof*⟩

lemma *listrel-sym*: *sym r* ⟹ *sym* (*listrel r*)
⟨*proof*⟩

lemma *listrel-trans*: *trans r* ⟹ *trans* (*listrel r*)
⟨*proof*⟩

theorem *equiv-listrel*: *equiv A r* ⟹ *equiv* (*lists A*) (*listrel r*)
⟨*proof*⟩

lemma *listrel-Nil* [*simp*]: *listrel r* `` {[]} = {[]}
⟨*proof*⟩

lemma *listrel-Cons*:
  listrel r `` {*x*#*xs*} = *set-Cons* (*r*``{*x*}) (*listrel r* `` {*xs*})
⟨*proof*⟩

43.5 Size function

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (*list-size f*)
⟨*proof*⟩

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (*option-size f*)
⟨*proof*⟩

lemma *list-size-estimation*[*termination-simp*]:
  x ∈ *set xs* ⟹ *y* < *f x* ⟹ *y* < *list-size f xs*
⟨*proof*⟩

lemma *list-size-estimation′*[*termination-simp*]:
  x ∈ *set xs* ⟹ *y* ≤ *f x* ⟹ *y* ≤ *list-size f xs*

⟨*proof*⟩

lemma *list-size-map*[*simp*]: *list-size f* (*map g xs*) = *list-size* (*f o g*) *xs*
⟨*proof*⟩

lemma *list-size-pointwise*[*termination-simp*]:
(⋀*x*. *x* ∈ *set xs* ⟹ *f x* < *g x*) ⟹ *list-size f xs* ≤ *list-size g xs*
⟨*proof*⟩

43.6 Transfer

definition
embed-list :: *nat list* ⇒ *int list*
where
embed-list l = *map int l*

definition
nat-list :: *int list* ⇒ *bool*
where
nat-list l = *nat-set* (*set l*)

definition
return-list :: *int list* ⇒ *nat list*
where
return-list l = *map nat l*

lemma *transfer-nat-int-list-return-embed*: *nat-list l* ⟶
embed-list (*return-list l*) = *l*
⟨*proof*⟩

lemma *transfer-nat-int-list-functions*:
l @ *m* = *return-list* (*embed-list l* @ *embed-list m*)
[] = *return-list* []
⟨*proof*⟩

43.7 Code generator

43.7.1 Setup

⟨*ML*⟩

code-type *list*
(*SML* - *list*)
(*OCaml* - *list*)
(*Haskell* ![(-)])
(*Scala List*[(-)])

code-const *Nil*
(*SML* [])
(*OCaml* [])

```
  (Haskell [])
  (Scala Nil)

code-instance list :: eq
  (Haskell -)

code-const eq-class.eq :: 'a::eq list ⇒ 'a list ⇒ bool
  (Haskell infixl 4 ==)

code-reserved SML
  list

code-reserved OCaml
  list

types-code
  list (- list)
attach (term-of) ⟪
fun term-of-list f T = HOLogic.mk-list T o map f;
⟫
attach (test) ⟪
fun gen-list' aG aT i j = frequency
  [(i, fn () =>
      let
        val (x, t) = aG j;
        val (xs, ts) = gen-list' aG aT (i-1) j
      in (x :: xs, fn () => HOLogic.cons-const aT $ t () $ ts ()) end),
   (1, fn () => ([], fn () => HOLogic.nil-const aT))] ()
and gen-list aG aT i = gen-list' aG aT i i;
⟫

consts-code Cons ((- ::/ -))

⟨ML⟩
```

43.7.2 Generation of efficient code

```
definition member :: 'a list ⇒ 'a ⇒ bool where
  mem-iff [code-post]: member xs x ⟷ x ∈ set xs

primrec
  null:: 'a list ⇒ bool
where
  null [] = True
  | null (x#xs) = False

primrec
  list-inter :: 'a list ⇒ 'a list ⇒ 'a list
where
```

```
  list-inter [] bs = []
  | list-inter (a#as) bs =
    (if a ∈ set bs then a # list-inter as bs else list-inter as bs)

primrec
  list-all :: ('a ⇒ bool) ⇒ ('a list ⇒ bool)
where
  list-all P [] = True
  | list-all P (x#xs) = (P x ∧ list-all P xs)

primrec
  list-ex :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
where
  list-ex P [] = False
  | list-ex P (x#xs) = (P x ∨ list-ex P xs)

primrec
  filtermap :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list
where
  filtermap f [] = []
  | filtermap f (x#xs) =
    (case f x of None ⇒ filtermap f xs
     | Some y ⇒ y # filtermap f xs)

primrec
  map-filter :: ('a ⇒ 'b) ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ 'b list
where
  map-filter f P [] = []
  | map-filter f P (x#xs) =
    (if P x then f x # map-filter f P xs else map-filter f P xs)

primrec
  length-unique :: 'a list ⇒ nat
where
  length-unique [] = 0
  | length-unique (x#xs) =
     (if x ∈ set xs then length-unique xs else Suc (length-unique xs))

primrec
  concat-map :: ('a => 'b list) => 'a list => 'b list
where
  concat-map f [] = []
  | concat-map f (x#xs) = f x @ concat-map f xs
```

Only use *member* for generating executable code. Otherwise use $x \in set\ xs$ instead — it is much easier to reason about. The same is true for *list-all* and *list-ex*: write $\forall x{\in}set\ xs$ and $\exists x{\in}set\ xs$ instead because the HOL quantifiers are aleady known to the automatic provers. In fact, the declarations in the code subsection make sure that $\in$, $\forall x{\in}set\ xs$ and $\exists x{\in}set\ xs$ are implemented

efficiently.

Efficient emptyness check is implemented by *null*.

The functions *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

lemma *rev-foldl-cons* [*code*]:
rev xs = *foldl* (λ*xs x*. *x* # *xs*) [] *xs*
⟨*proof*⟩

lemmas *in-set-code* [*code-unfold*] = *mem-iff* [*symmetric*]

lemma *member-simps* [*simp*, *code*]:
member (*x* # *xs*) *y* ⟷ *x* = *y* ∨ *member xs y*
member [] *y* ⟷ *False*
⟨*proof*⟩

lemma *member-set*:
member = *set*
⟨*proof*⟩

abbreviation (*input*) *mem* :: *'a* ⇒ *'a list* ⇒ *bool* (**infixl** *mem* 55) **where**
x mem xs ≡ *member xs x* — for backward compatibility

lemma *empty-null*:
xs = [] ⟷ *null xs*
⟨*proof*⟩

lemma [*code-unfold*]:
eq-class.eq xs [] ⟷ *null xs*
⟨*proof*⟩

lemmas *null-empty* [*code-post*] =
empty-null [*symmetric*]

lemma *list-inter-conv*:
set (*list-inter xs ys*) = *set xs* ∩ *set ys*
⟨*proof*⟩

lemma *list-all-iff* [*code-post*]:
list-all P xs ⟷ (∀ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemmas *list-ball-code* [*code-unfold*] = *list-all-iff* [*symmetric*]

lemma *list-all-append* [*simp*]:
list-all P (*xs* @ *ys*) ⟷ (*list-all P xs* ∧ *list-all P ys*)
⟨*proof*⟩

lemma *list-all-rev* [*simp*]:

list-all P (rev xs) ⟷ *list-all P xs*
⟨*proof*⟩

lemma *list-all-length*:
list-all P xs ⟷ (∀ *n* < *length xs*. *P* (*xs* ! *n*))
⟨*proof*⟩

lemma *list-ex-iff* [*code-post*]:
list-ex P xs ⟷ (∃ *x* ∈ *set xs*. *P x*)
⟨*proof*⟩

lemmas *list-bex-code* [*code-unfold*] =
list-ex-iff [*symmetric*]

lemma *list-ex-length*:
list-ex P xs ⟷ (∃ *n* < *length xs*. *P* (*xs* ! *n*))
⟨*proof*⟩

lemma *filtermap-conv*:
filtermap f xs = *map* (λ*x*. *the* (*f x*)) (*filter* (λ*x*. *f x* ≠ *None*) *xs*)
⟨*proof*⟩

lemma *map-filter-conv* [*simp*]:
map-filter f P xs = *map f* (*filter P xs*)
⟨*proof*⟩

lemma *length-remdups-length-unique* [*code-unfold*]:
length (*remdups xs*) = *length-unique xs*
⟨*proof*⟩

lemma *concat-map-code*[*code-unfold*]:
concat(*map f xs*) = *concat-map f xs*
⟨*proof*⟩

declare *INFI-def* [*code-unfold*]
declare *SUPR-def* [*code-unfold*]

declare *set-map* [*symmetric*, *code-unfold*]

hide-const (**open**) *length-unique*

Code for bounded quantification and summation over nats.

lemma *atMost-upto* [*code-unfold*]:
{..*n*} = *set* [*0*..<*Suc n*]
⟨*proof*⟩

lemma *atLeast-upt* [*code-unfold*]:
{..<*n*} = *set* [*0*..<*n*]
⟨*proof*⟩

lemma *greaterThanLessThan-upt* [*code-unfold*]:
$\{n<..<m\} = set\ [Suc\ n..<m]$
⟨*proof*⟩

lemmas *atLeastLessThan-upt* [*code-unfold*] = *set-upt* [*symmetric*]

lemma *greaterThanAtMost-upt* [*code-unfold*]:
$\{n<..m\} = set\ [Suc\ n..<Suc\ m]$
⟨*proof*⟩

lemma *atLeastAtMost-upt* [*code-unfold*]:
$\{n..m\} = set\ [n..<Suc\ m]$
⟨*proof*⟩

lemma *all-nat-less-eq* [*code-unfold*]:
$(\forall\, m{<}n{::}nat.\ P\ m) \longleftrightarrow (\forall\, m \in \{0..<n\}.\ P\ m)$
⟨*proof*⟩

lemma *ex-nat-less-eq* [*code-unfold*]:
$(\exists\, m{<}n{::}nat.\ P\ m) \longleftrightarrow (\exists\, m \in \{0..<n\}.\ P\ m)$
⟨*proof*⟩

lemma *all-nat-less* [*code-unfold*]:
$(\forall\, m{\leq}n{::}nat.\ P\ m) \longleftrightarrow (\forall\, m \in \{0..n\}.\ P\ m)$
⟨*proof*⟩

lemma *ex-nat-less* [*code-unfold*]:
$(\exists\, m{\leq}n{::}nat.\ P\ m) \longleftrightarrow (\exists\, m \in \{0..n\}.\ P\ m)$
⟨*proof*⟩

lemma *setsum-set-distinct-conv-listsum*:
$distinct\ xs \Longrightarrow setsum\ f\ (set\ xs) = listsum\ (map\ f\ xs)$
⟨*proof*⟩

lemma *setsum-set-upt-conv-listsum* [*code-unfold*]:
$setsum\ f\ (set\ [m..<n]) = listsum\ (map\ f\ [m..<n])$
⟨*proof*⟩

General equivalence between *listsum* and *setsum*

lemma *listsum-setsum-nth*:
$listsum\ xs = (\sum\ i = 0\ ..<\ length\ xs.\ xs\ !\ i)$
⟨*proof*⟩

Code for summation over ints.

lemma *greaterThanLessThan-upto* [*code-unfold*]:
$\{i<..<j{::}int\} = set\ [i+1..j\ -\ 1]$
⟨*proof*⟩

lemma *atLeastLessThan-upto* [*code-unfold*]:
$\{i..<j::int\} = set\ [i..j - 1]$
⟨*proof*⟩

lemma *greaterThanAtMost-upto* [*code-unfold*]:
$\{i<..j::int\} = set\ [i+1..j]$
⟨*proof*⟩

lemmas *atLeastAtMost-upto* [*code-unfold*] = *set-upto*[*symmetric*]

lemma *setsum-set-upto-conv-listsum* [*code-unfold*]:
$setsum\ f\ (set\ [i..j::int]) = listsum\ (map\ f\ [i..j])$
⟨*proof*⟩

Optimized code for $\forall\, i{\in}\{a..b::int\}$ and $\forall\, n{:}\{a..<b::nat\}$ and similiarly for $\exists$.

function *all-from-to-nat* :: $(nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**
$all\text{-}from\text{-}to\text{-}nat\ P\ i\ j =$
$(if\ i < j\ then\ if\ P\ i\ then\ all\text{-}from\text{-}to\text{-}nat\ P\ (i+1)\ j\ else\ False$
$\ \ else\ True)$
⟨*proof*⟩
termination
⟨*proof*⟩

declare *all-from-to-nat.simps*[*simp del*]

lemma *all-from-to-nat-iff-ball*:
$all\text{-}from\text{-}to\text{-}nat\ P\ i\ j = (ALL\ n : \{i\ ..<\ j\}.\ P\ n)$
⟨*proof*⟩

lemma *list-all-iff-all-from-to-nat*[*code-unfold*]:
$list\text{-}all\ P\ [i..<j] = all\text{-}from\text{-}to\text{-}nat\ P\ i\ j$
⟨*proof*⟩

lemma *list-ex-iff-not-all-from-to-not-nat*[*code-unfold*]:
$list\text{-}ex\ P\ [i..<j] = (\sim all\text{-}from\text{-}to\text{-}nat\ (\%x.\ \sim P\ x)\ i\ j)$
⟨*proof*⟩

function *all-from-to-int* :: $(int \Rightarrow bool) \Rightarrow int \Rightarrow int \Rightarrow bool$ **where**
$all\text{-}from\text{-}to\text{-}int\ P\ i\ j =$
$(if\ i <= j\ then\ if\ P\ i\ then\ all\text{-}from\text{-}to\text{-}int\ P\ (i+1)\ j\ else\ False$
$\ \ else\ True)$
⟨*proof*⟩
termination
⟨*proof*⟩

declare *all-from-to-int.simps*[*simp del*]

lemma *all-from-to-int-iff-ball*:
all-from-to-int P i j = (ALL n : {i .. j}. P n)
⟨*proof*⟩

lemma *list-all-iff-all-from-to-int*[*code-unfold*]:
list-all P [i..j] = all-from-to-int P i j
⟨*proof*⟩

lemma *list-ex-iff-not-all-from-to-not-int*[*code-unfold*]:
list-ex P [i..j] = (~ all-from-to-int (%x. ~P x) i j)
⟨*proof*⟩

end

44 Random: A HOL random engine

theory *Random*
imports *Code-Numeral List*
begin

notation *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o→* *60*)

44.1 Auxiliary functions

fun *log* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral* **where**
log b i = (if b ≤ 1 ∨ i < b then 1 else 1 + log b (i div b))

definition *inc-shift* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral* **where**
inc-shift v k = (if v = k then 1 else k + 1)

definition *minus-shift* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral*
where
minus-shift r k l = (if k < l then r + k − l else k − l)

44.2 Random seeds

types *seed = code-numeral × code-numeral*

primrec *next* :: *seed* ⇒ *code-numeral* × *seed* **where**
next (v, w) = (let
k = v div 53668;
*v′ = minus-shift 2147483563 ((v mod 53668) * 40014) (k * 12211)*;
l = w div 52774;
*w′ = minus-shift 2147483399 ((w mod 52774) * 40692) (l * 3791)*;
z = minus-shift 2147483562 v′ (w′ + 1) + 1
in (z, (v′, w′)))

definition *split-seed* :: *seed* ⇒ *seed* × *seed* **where**
split-seed s = (*let*
(*v*, *w*) = *s*;
(*v'*, *w'*) = *snd* (*next s*);
v'' = *inc-shift 2147483562 v*;
w'' = *inc-shift 2147483398 w*
in ((*v''*, *w'*), (*v'*, *w''*)))

44.3 Base selectors

fun *iterate* :: *code-numeral* ⇒ ('*b* ⇒ '*a* ⇒ '*b* × '*a*) ⇒ '*b* ⇒ '*a* ⇒ '*b* × '*a* **where**
iterate k f x = (*if k* = *0 then Pair x else f x o*→ *iterate* (*k* − *1*) *f*)

definition *range* :: *code-numeral* ⇒ *seed* ⇒ *code-numeral* × *seed* **where**
range k = *iterate* (*log 2147483561 k*)
(λ*l*. *next o*→ (λ*v*. *Pair* (*v* + *l* ∗ *2147483561*))) *1*
o→ (λ*v*. *Pair* (*v mod k*))

lemma *range*:
k > *0* ⟹ *fst* (*range k s*) < *k*
⟨*proof*⟩

definition *select* :: '*a list* ⇒ *seed* ⇒ '*a* × *seed* **where**
select xs = *range* (*Code-Numeral.of-nat* (*length xs*))
o→ (λ*k*. *Pair* (*nth xs* (*Code-Numeral.nat-of k*)))

lemma *select*:
assumes *xs* ≠ []
shows *fst* (*select xs s*) ∈ *set xs*
⟨*proof*⟩

primrec *pick* :: (*code-numeral* × '*a*) *list* ⇒ *code-numeral* ⇒ '*a* **where**
pick (*x* # *xs*) *i* = (*if i* < *fst x then snd x else pick xs* (*i* − *fst x*))

lemma *pick-member*:
i < *listsum* (*map fst xs*) ⟹ *pick xs i* ∈ *set* (*map snd xs*)
⟨*proof*⟩

lemma *pick-drop-zero*:
pick (*filter* (λ(*k*, -). *k* > *0*) *xs*) = *pick xs*
⟨*proof*⟩

lemma *pick-same*:
l < *length xs* ⟹ *Random.pick* (*map* (*Pair 1*) *xs*) (*Code-Numeral.of-nat l*) =
nth xs l
⟨*proof*⟩

definition *select-weight* :: (*code-numeral* × '*a*) *list* ⇒ *seed* ⇒ '*a* × *seed* **where**

select-weight xs = range (listsum (map fst xs))
o→ (λk. Pair (pick xs k))

lemma *select-weight-member*:
assumes *0 < listsum (map fst xs)*
shows *fst (select-weight xs s) ∈ set (map snd xs)*
⟨*proof*⟩

lemma *select-weight-cons-zero*:
select-weight ((0, x) # xs) = select-weight xs
⟨*proof*⟩

lemma *select-weigth-drop-zero*:
select-weight (filter (λ(k, -). k > 0) xs) = select-weight xs
⟨*proof*⟩

lemma *select-weigth-select*:
assumes *xs ≠ []*
shows *select-weight (map (Pair 1) xs) = select xs*
⟨*proof*⟩

44.4 *ML* interface

code-reflect *Random-Engine*
functions *range select select-weight*

⟨*ML*⟩

hide-type (**open**) *seed*
hide-const (**open**) *inc-shift minus-shift log next split-seed iterate range select pick select-weight*
hide-fact (**open**) *range-def*

no-notation *fcomp* (**infixl** *o>* *60*)
no-notation *scomp* (**infixl** *o→* *60*)

end

45 String: Character and string types

theory *String*
imports *List*
uses
(*Tools/string-syntax.ML*)
(*Tools/string-code.ML*)
begin

45.1 Characters

datatype *nibble* =
  Nibble0 | *Nibble1* | *Nibble2* | *Nibble3* | *Nibble4* | *Nibble5* | *Nibble6* | *Nibble7*
| *Nibble8* | *Nibble9* | *NibbleA* | *NibbleB* | *NibbleC* | *NibbleD* | *NibbleE* | *NibbleF*

lemma *UNIV-nibble*:
  UNIV = {*Nibble0*, *Nibble1*, *Nibble2*, *Nibble3*, *Nibble4*, *Nibble5*, *Nibble6*, *Nibble7*,
    Nibble8, *Nibble9*, *NibbleA*, *NibbleB*, *NibbleC*, *NibbleD*, *NibbleE*, *NibbleF*} (**is** - = *?A*)
⟨*proof*⟩

instance *nibble* :: *finite*
  ⟨*proof*⟩

datatype *char* = *Char nibble nibble*
  — Note: canonical order of character encoding coincides with standard term ordering

lemma *UNIV-char*:
  UNIV = *image* (*split Char*) (*UNIV* × *UNIV*)
⟨*proof*⟩

instance *char* :: *finite*
  ⟨*proof*⟩

lemma *size-char* [*code*, *simp*]:
  size (*c*::*char*) = *0* ⟨*proof*⟩

lemma *char-size* [*code*, *simp*]:
  char-size (*c*::*char*) = *0* ⟨*proof*⟩

primrec *nibble-pair-of-char* :: *char* ⇒ *nibble* × *nibble* **where**
  nibble-pair-of-char (*Char n m*) = (*n*, *m*)

⟨*ML*⟩

lemma *char-case-nibble-pair* [*code*, *code-unfold*]:
  char-case f = *split f o nibble-pair-of-char*
  ⟨*proof*⟩

lemma *char-rec-nibble-pair* [*code*, *code-unfold*]:
  char-rec f = *split f o nibble-pair-of-char*
  ⟨*proof*⟩

syntax
  -*Char* :: *xstr* => *char* (*CHR* -)

45.2 Strings

types *string* = *char list*

syntax
-String :: *xstr* => *string* (-)

⟨*ML*⟩

definition *chars* :: *string* **where**
chars = [*Char Nibble0 Nibble0*, *Char Nibble0 Nibble1*, *Char Nibble0 Nibble2*,
Char Nibble0 Nibble3, *Char Nibble0 Nibble4*, *Char Nibble0 Nibble5*,
Char Nibble0 Nibble6, *Char Nibble0 Nibble7*, *Char Nibble0 Nibble8*,
Char Nibble0 Nibble9, *Char Nibble0 NibbleA*, *Char Nibble0 NibbleB*,
Char Nibble0 NibbleC, *Char Nibble0 NibbleD*, *Char Nibble0 NibbleE*,
Char Nibble0 NibbleF, *Char Nibble1 Nibble0*, *Char Nibble1 Nibble1*,
Char Nibble1 Nibble2, *Char Nibble1 Nibble3*, *Char Nibble1 Nibble4*,
Char Nibble1 Nibble5, *Char Nibble1 Nibble6*, *Char Nibble1 Nibble7*,
Char Nibble1 Nibble8, *Char Nibble1 Nibble9*, *Char Nibble1 NibbleA*,
Char Nibble1 NibbleB, *Char Nibble1 NibbleC*, *Char Nibble1 NibbleD*,
Char Nibble1 NibbleE, *Char Nibble1 NibbleF*, *CHR '' ''*, *CHR ''!''*,
Char Nibble2 Nibble2, *CHR ''#''*, *CHR ''$''*, *CHR ''%''*, *CHR ''&''*,
Char Nibble2 Nibble7, *CHR ''(''*, *CHR '')''*, *CHR ''*''*, *CHR ''+''*, *CHR '',''*,
CHR ''−'', *CHR ''.''*, *CHR ''/''*, *CHR ''0''*, *CHR ''1''*, *CHR ''2''*, *CHR ''3''*,
CHR ''4'', *CHR ''5''*, *CHR ''6''*, *CHR ''7''*, *CHR ''8''*, *CHR ''9''*, *CHR '':''*,
CHR '';'', *CHR ''<''*, *CHR ''=''*, *CHR ''>''*, *CHR ''?''*, *CHR ''@''*, *CHR ''A''*,
CHR ''B'', *CHR ''C''*, *CHR ''D''*, *CHR ''E''*, *CHR ''F''*, *CHR ''G''*, *CHR ''H''*,
CHR ''I'', *CHR ''J''*, *CHR ''K''*, *CHR ''L''*, *CHR ''M''*, *CHR ''N''*, *CHR ''O''*,
CHR ''P'', *CHR ''Q''*, *CHR ''R''*, *CHR ''S''*, *CHR ''T''*, *CHR ''U''*, *CHR ''V''*,
CHR ''W'', *CHR ''X''*, *CHR ''Y''*, *CHR ''Z''*, *CHR ''[''*, *Char Nibble5 NibbleC*,
CHR '']'', *CHR ''^''*, *CHR ''-''*, *Char Nibble6 Nibble0*, *CHR ''a''*, *CHR ''b''*,
CHR ''c'', *CHR ''d''*, *CHR ''e''*, *CHR ''f''*, *CHR ''g''*, *CHR ''h''*, *CHR ''i''*,
CHR ''j'', *CHR ''k''*, *CHR ''l''*, *CHR ''m''*, *CHR ''n''*, *CHR ''o''*, *CHR ''p''*,
CHR ''q'', *CHR ''r''*, *CHR ''s''*, *CHR ''t''*, *CHR ''u''*, *CHR ''v''*, *CHR ''w''*,
CHR ''x'', *CHR ''y''*, *CHR ''z''*, *CHR ''{''*, *CHR ''|''*, *CHR ''}''*, *CHR ''~''*,
Char Nibble7 NibbleF, *Char Nibble8 Nibble0*, *Char Nibble8 Nibble1*,
Char Nibble8 Nibble2, *Char Nibble8 Nibble3*, *Char Nibble8 Nibble4*,
Char Nibble8 Nibble5, *Char Nibble8 Nibble6*, *Char Nibble8 Nibble7*,
Char Nibble8 Nibble8, *Char Nibble8 Nibble9*, *Char Nibble8 NibbleA*,
Char Nibble8 NibbleB, *Char Nibble8 NibbleC*, *Char Nibble8 NibbleD*,
Char Nibble8 NibbleE, *Char Nibble8 NibbleF*, *Char Nibble9 Nibble0*,
Char Nibble9 Nibble1, *Char Nibble9 Nibble2*, *Char Nibble9 Nibble3*,
Char Nibble9 Nibble4, *Char Nibble9 Nibble5*, *Char Nibble9 Nibble6*,
Char Nibble9 Nibble7, *Char Nibble9 Nibble8*, *Char Nibble9 Nibble9*,
Char Nibble9 NibbleA, *Char Nibble9 NibbleB*, *Char Nibble9 NibbleC*,
Char Nibble9 NibbleD, *Char Nibble9 NibbleE*, *Char Nibble9 NibbleF*,
Char NibbleA Nibble0, *Char NibbleA Nibble1*, *Char NibbleA Nibble2*,
Char NibbleA Nibble3, *Char NibbleA Nibble4*, *Char NibbleA Nibble5*,
Char NibbleA Nibble6, *Char NibbleA Nibble7*, *Char NibbleA Nibble8*,
Char NibbleA Nibble9, *Char NibbleA NibbleA*, *Char NibbleA NibbleB*,

Char NibbleA NibbleC, *Char NibbleA NibbleD*, *Char NibbleA NibbleE*, *Char NibbleA NibbleF*, *Char NibbleB Nibble0*, *Char NibbleB Nibble1*, *Char NibbleB Nibble2*, *Char NibbleB Nibble3*, *Char NibbleB Nibble4*, *Char NibbleB Nibble5*, *Char NibbleB Nibble6*, *Char NibbleB Nibble7*, *Char NibbleB Nibble8*, *Char NibbleB Nibble9*, *Char NibbleB NibbleA*, *Char NibbleB NibbleB*, *Char NibbleB NibbleC*, *Char NibbleB NibbleD*, *Char NibbleB NibbleE*, *Char NibbleB NibbleF*, *Char NibbleC Nibble0*, *Char NibbleC Nibble1*, *Char NibbleC Nibble2*, *Char NibbleC Nibble3*, *Char NibbleC Nibble4*, *Char NibbleC Nibble5*, *Char NibbleC Nibble6*, *Char NibbleC Nibble7*, *Char NibbleC Nibble8*, *Char NibbleC Nibble9*, *Char NibbleC NibbleA*, *Char NibbleC NibbleB*, *Char NibbleC NibbleC*, *Char NibbleC NibbleD*, *Char NibbleC NibbleE*, *Char NibbleC NibbleF*, *Char NibbleD Nibble0*, *Char NibbleD Nibble1*, *Char NibbleD Nibble2*, *Char NibbleD Nibble3*, *Char NibbleD Nibble4*, *Char NibbleD Nibble5*, *Char NibbleD Nibble6*, *Char NibbleD Nibble7*, *Char NibbleD Nibble8*, *Char NibbleD Nibble9*, *Char NibbleD NibbleA*, *Char NibbleD NibbleB*, *Char NibbleD NibbleC*, *Char NibbleD NibbleD*, *Char NibbleD NibbleE*, *Char NibbleD NibbleF*, *Char NibbleE Nibble0*, *Char NibbleE Nibble1*, *Char NibbleE Nibble2*, *Char NibbleE Nibble3*, *Char NibbleE Nibble4*, *Char NibbleE Nibble5*, *Char NibbleE Nibble6*, *Char NibbleE Nibble7*, *Char NibbleE Nibble8*, *Char NibbleE Nibble9*, *Char NibbleE NibbleA*, *Char NibbleE NibbleB*, *Char NibbleE NibbleC*, *Char NibbleE NibbleD*, *Char NibbleE NibbleE*, *Char NibbleE NibbleF*, *Char NibbleF Nibble0*, *Char NibbleF Nibble1*, *Char NibbleF Nibble2*, *Char NibbleF Nibble3*, *Char NibbleF Nibble4*, *Char NibbleF Nibble5*, *Char NibbleF Nibble6*, *Char NibbleF Nibble7*, *Char NibbleF Nibble8*, *Char NibbleF Nibble9*, *Char NibbleF NibbleA*, *Char NibbleF NibbleB*, *Char NibbleF NibbleC*, *Char NibbleF NibbleD*, *Char NibbleF NibbleE*, *Char NibbleF NibbleF*]

45.3 Strings as dedicated datatype

datatype *literal* = *STR string*

declare *literal.cases* [*code del*] *literal.recs* [*code del*]

lemma [*code*]: *size* (*s*::*literal*) = *0*
⟨*proof*⟩

lemma [*code*]: *literal-size* (*s*::*literal*) = *0*
⟨*proof*⟩

45.4 Code generator

⟨*ML*⟩

code-reserved *SML string*
code-reserved *OCaml string*
code-reserved *Scala string*

code-type *literal*

(*SML string*)
(*OCaml string*)
(*Haskell String*)
(*Scala String*)

⟨*ML*⟩

code-instance *literal* :: *eq*
(*Haskell* −)

code-const *eq-class.eq* :: *literal* ⇒ *literal* ⇒ *bool*
(*SML* !((- : *string*) = -))
(*OCaml* !((- : *string*) = -))
(*Haskell* **infixl** *4* ==)
(*Scala* **infixl** *5* ==)

types-code
char (*string*)
attach (*term-of*) ⟪
val term-of-char = HOLogic.mk-char o ord;
⟫
attach (*test*) ⟪
fun gen-char i =
let val j = random-range (*ord a*) (*Int.min* (*ord a + i*, *ord z*))
in (*chr j*, *fn* () => *HOLogic.mk-char j*) *end*;
⟫

⟨*ML*⟩

hide-type (**open**) *literal*

end

46 Typerep: Reflecting Pure types into HOL

theory *Typerep*
imports *Plain String*
begin

datatype *typerep* = *Typerep String.literal typerep list*

class *typerep* =
fixes *typerep* :: *'a itself* ⇒ *typerep*
begin

definition *typerep-of* :: *'a* ⇒ *typerep* **where**
[*simp*]: *typerep-of x* = *typerep TYPE*(*'a*)

end

syntax
-TYPEREP :: *type* => *logic* ((*1TYPEREP*/(*1'*(*-'*))))

⟨*ML*⟩

lemma [*code*]:
eq-class.eq (*Typerep tyco1 tys1*) (*Typerep tyco2 tys2*) ⟷ *eq-class.eq tyco1 tyco2*
∧ *list-all2 eq-class.eq tys1 tys2*
⟨*proof*⟩

code-type *typerep*
(*Eval Term.typ*)

code-const *Typerep*
(*Eval Term.Type*/ (-, -))

code-reserved *Eval Term*

hide-const (**open**) *typerep Typerep*

end

47 Code-Evaluation: Term evaluation using the generic code generator

theory *Code-Evaluation*
imports *Plain Typerep Code-Numeral*
begin

47.1 Term representation

47.1.1 Terms and class *term-of*

datatype *term* = *dummy-term*

definition *Const* :: *String.literal* ⇒ *typerep* ⇒ *term* **where**
Const - - = *dummy-term*

definition *App* :: *term* ⇒ *term* ⇒ *term* **where**
App - - = *dummy-term*

code-datatype *Const App*

class *term-of* = *typerep* +
fixes *term-of* :: *'a* ⇒ *term*

lemma *term-of-anything*: *term-of x* ≡ *t*

⟨*proof*⟩

definition *valapp* :: ('*a* ⇒ '*b*) × (*unit* ⇒ *term*)
⇒ '*a* × (*unit* ⇒ *term*) ⇒ '*b* × (*unit* ⇒ *term*) **where**
valapp f x = (*fst f* (*fst x*), λ*u*. *App* (*snd f* ()) (*snd x* ()))

lemma *valapp-code* [*code*, *code-unfold*]:
valapp (*f*, *tf*) (*x*, *tx*) = (*f x*, λ*u*. *App* (*tf* ()) (*tx* ()))
⟨*proof*⟩

47.1.2 *term-of* instances

instantiation *fun* :: (*typerep*, *typerep*) *term-of*
begin

definition
term-of (*f* :: '*a* ⇒ '*b*) = *Const* (*STR* ''*dummy-pattern*'') (*Typerep.Typerep* (*STR* ''*fun*'')
[*Typerep.typerep TYPE*('*a*), *Typerep.typerep TYPE*('*b*)])

instance ⟨*proof*⟩

end

⟨*ML*⟩

47.1.3 Code generator setup

lemmas [*code del*] = *term.recs term.cases term.size*
lemma [*code*, *code del*]: *eq-class.eq* (*t1*::*term*) *t2* ⟷ *eq-class.eq t1 t2* ⟨*proof*⟩

lemma [*code*, *code del*]: (*term-of* :: *typerep* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code*, *code del*]: (*term-of* :: *term* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code*, *code del*]: (*term-of* :: *String.literal* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code*, *code del*]:
(*Code-Evaluation.term-of* :: '*a*::{*type*, *term-of*} *Predicate.pred* ⇒ *Code-Evaluation.term*)
= *Code-Evaluation.term-of* ⟨*proof*⟩
lemma [*code*, *code del*]:
(*Code-Evaluation.term-of* :: '*a*::{*type*, *term-of*} *Predicate.seq* ⇒ *Code-Evaluation.term*)
= *Code-Evaluation.term-of* ⟨*proof*⟩

lemma *term-of-char* [*unfolded typerep-fun-def typerep-char-def typerep-nibble-def*, *code*]: *Code-Evaluation.term-of c* =
(*let* (*n*, *m*) = *nibble-pair-of-char c*
in Code-Evaluation.App (*Code-Evaluation.App* (*Code-Evaluation.Const* (*STR* ''*String.char.Char*'') (*TYPEREP*(*nibble* ⇒ *nibble* ⇒ *char*)))
(*Code-Evaluation.term-of n*)) (*Code-Evaluation.term-of m*))
⟨*proof*⟩

code-type *term*

(*Eval Term.term*)

code-const *Const* **and** *App*
(*Eval Term.Const*/ ((-), (-)) **and** *Term.*$/ ((-), (-)))

code-const *term-of* :: *String.literal* ⇒ *term*
(*Eval HOLogic.mk'-literal*)

code-reserved *Eval HOLogic*

47.1.4 Syntax

definition *termify* :: *'a* ⇒ *term* **where**
[*code del*]: *termify x* = *dummy-term*

abbreviation *valtermify* :: *'a* ⇒ *'a* × (*unit* ⇒ *term*) **where**
valtermify x ≡ (*x*, λ*u*. *termify x*)

⟨*ML*⟩

locale *term-syntax*
begin

notation *App* (**infixl** <·> *70*)
and *valapp* (**infixl** {·} *70*)

end

interpretation *term-syntax* ⟨*proof*⟩

no-notation *App* (**infixl** <·> *70*)
and *valapp* (**infixl** {·} *70*)

47.2 Numeric types

definition *term-of-num* :: *'a*::{*semiring-div*} ⇒ *'a*::{*semiring-div*} ⇒ *term* **where**
term-of-num two = (λ-. *dummy-term*)

lemma (**in** *term-syntax*) *term-of-num-code* [*code*]:
term-of-num two k = (*if k* = *0 then termify Int.Pls*
else (*if k mod two* = *0*
then termify Int.Bit0 <·> *term-of-num two* (*k div two*)
else termify Int.Bit1 <·> *term-of-num two* (*k div two*)))
⟨*proof*⟩

lemma (**in** *term-syntax*) *term-of-nat-code* [*code*]:
term-of (*n*::*nat*) = *termify* (*number-of* :: *int* ⇒ *nat*) <·> *term-of-num* (*2*::*nat*)
n
⟨*proof*⟩

lemma (**in** *term-syntax*) *term-of-int-code* [*code*]:
term-of (*k*::*int*) = (*if k = 0 then termify* (*0* :: *int*)
else if k > 0 then termify (*number-of* :: *int* ⇒ *int*) <·> *term-of-num* (*2*::*int*) *k*
else termify (*uminus* :: *int* ⇒ *int*) <·> (*termify* (*number-of* :: *int* ⇒ *int*) <·> *term-of-num* (*2*::*int*) (− *k*)))
⟨*proof*⟩

lemma (**in** *term-syntax*) *term-of-code-numeral-code* [*code*]:
term-of (*k*::*code-numeral*) = *termify* (*number-of* :: *int* ⇒ *code-numeral*) <·> *term-of-num* (*2*::*code-numeral*) *k*
⟨*proof*⟩

47.3 Obfuscate

⟨*ML*⟩

hide-const *dummy-term App valapp*
hide-const (**open**) *Const termify valtermify term-of term-of-num*

47.4 Tracing of generated and evaluated code

definition *tracing* :: *String.literal* => *'a* => *'a*
where
[*code del*]: *tracing s x* = *x*

⟨*ML*⟩

code-const *tracing* :: *String.literal* => *'a* => *'a*
(*Eval Code'-Evaluation.tracing*)

hide-const (**open**) *tracing*
code-reserved *Eval Code-Evaluation*

47.5 Evaluation setup

⟨*ML*⟩

end

48 Quickcheck: A simple counterexample generator

theory *Quickcheck*
imports *Random Code-Evaluation*
uses (*Tools/quickcheck-generators.ML*)
begin

notation *fcomp* (**infixl** *o>* *60*)

notation *scomp* (**infixl** *o→ 60*)

48.1 The *random* class

class *random* = *typerep* +
fixes *random* :: *code-numeral* ⇒ *Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*

48.2 Fundamental and numeric types

instantiation *bool* :: *random*
begin

definition
random i = *Random.range 2 o→*
(λ*k*. *Pair* (*if k* = *0 then Code-Evaluation.valtermify False else Code-Evaluation.valtermify True*))

instance ⟨*proof*⟩

end

instantiation *itself* :: (*typerep*) *random*
begin

definition *random-itself* :: *code-numeral* ⇒ *Random.seed* ⇒ (*'a itself* × (*unit* ⇒ *term*)) × *Random.seed* **where**
random-itself - = *Pair* (*Code-Evaluation.valtermify TYPE*(*'a*))

instance ⟨*proof*⟩

end

instantiation *char* :: *random*
begin

definition
random - = *Random.select chars o→* (λ*c*. *Pair* (*c*, λ*u*. *Code-Evaluation.term-of c*))

instance ⟨*proof*⟩

end

instantiation *String.literal* :: *random*
begin

definition
random - = *Pair* (*STR ''''*, λ*u*. *Code-Evaluation.term-of* (*STR ''''*))

instance ⟨*proof*⟩

end

instantiation *nat* :: *random*
begin

definition *random-nat* :: *code-numeral* ⇒ *Random.seed* ⇒ (*nat* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed* **where**
random-nat i = *Random.range* (*i* + *1*) *o*→ (λ*k*. *Pair* (
let n = *Code-Numeral.nat-of k*
in (*n*, λ-. *Code-Evaluation.term-of n*)))

instance ⟨*proof*⟩

end

instantiation *int* :: *random*
begin

definition
random i = *Random.range* (*2* ∗ *i* + *1*) *o*→ (λ*k*. *Pair* (
let j = (*if k* ≥ *i then Code-Numeral.int-of* (*k* − *i*) *else* − *Code-Numeral.int-of* (*i* − *k*))
in (*j*, λ-. *Code-Evaluation.term-of j*)))

instance ⟨*proof*⟩

end

48.3 Complex generators

Towards *'a* ⇒ *'b*

axiomatization *random-fun-aux* :: *typerep* ⇒ *typerep* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *term*)
⇒ (*Random.seed* ⇒ (*'b* × (*unit* ⇒ *term*)) × *Random.seed*) ⇒ (*Random.seed* ⇒ *Random.seed* × *Random.seed*)
⇒ *Random.seed* ⇒ ((*'a* ⇒ *'b*) × (*unit* ⇒ *term*)) × *Random.seed*

definition *random-fun-lift* :: (*Random.seed* ⇒ (*'b* × (*unit* ⇒ *term*)) × *Random.seed*)
⇒ *Random.seed* ⇒ ((*'a*::*term-of* ⇒ *'b*::*typerep*) × (*unit* ⇒ *term*)) × *Random.seed*
where
random-fun-lift f = *random-fun-aux TYPEREP*(*'a*) *TYPEREP*(*'b*) (*op* =) *Code-Evaluation.term-of f Random.split-seed*

instantiation *fun* :: ({*eq*, *term-of*}, *random*) *random*
begin

definition *random-fun* :: *code-numeral* ⇒ *Random.seed* ⇒ ((*'a* ⇒ *'b*) × (*unit* ⇒ *term*)) × *Random.seed* **where**
random i = *random-fun-lift* (*random i*)

instance ⟨*proof*⟩

end

Towards type copies and datatypes

definition *collapse* :: (*'a* ⇒ (*'a* ⇒ *'b* × *'a*) × *'a*) ⇒ *'a* ⇒ *'b* × *'a* **where**
collapse f = (*f o*→ *id*)

definition *beyond* :: *code-numeral* ⇒ *code-numeral* ⇒ *code-numeral* **where**
beyond k l = (*if l* > *k then l else 0*)

lemma *beyond-zero*:
beyond k 0 = *0*
⟨*proof*⟩

lemma *random-aux-rec*:
fixes *random-aux* :: *code-numeral* ⇒ *'a*
assumes *random-aux 0* = *rhs 0*
and ⋀*k*. *random-aux* (*Suc-code-numeral k*) = *rhs* (*Suc-code-numeral k*)
shows *random-aux k* = *rhs k*
⟨*proof*⟩

⟨*ML*⟩

48.4 Code setup

code-const *random-fun-aux* (*Quickcheck Quickcheck'-Generators.random'-fun*)
— With enough criminal energy this can be abused to derive *False*; for this reason we use a distinguished target *Quickcheck* not spoiling the regular trusted code generation

code-reserved *Quickcheck Quickcheck-Generators*

no-notation *fcomp* (**infixl** *o*> *60*)
no-notation *scomp* (**infixl** *o*→ *60*)

48.5 The Random-Predicate Monad

fun *iter'* ::
'a itself => *code-numeral* => *code-numeral* => *code-numeral* ∗ *code-numeral* => (*'a*::*random*) *Predicate.pred*
where
iter' T nrandom sz seed = (*if nrandom* = *0 then bot-class.bot else*
let ((*x*, -), *seed'*) = *random sz seed*
in Predicate.Seq (%*u*. *Predicate.Insert x* (*iter' T* (*nrandom* − *1*) *sz seed'*)))

definition *iter* :: *code-numeral* => *code-numeral* => *code-numeral* * *code-numeral* => (*'a*::*random*) *Predicate.pred*
where
iter nrandom sz seed = *iter'* (*TYPE*(*'a*)) *nrandom sz seed*

lemma [*code*]:
iter nrandom sz seed = (*if nrandom* = *0 then bot-class.bot else*
let ((*x*, -), *seed'*) = *random sz seed*
in Predicate.Seq (%*u*. *Predicate.Insert x* (*iter* (*nrandom* − *1*) *sz seed'*)))
⟨*proof*⟩

types *'a randompred* = *Random.seed* ⇒ (*'a Predicate.pred* × *Random.seed*)

definition *empty* :: *'a randompred*
where *empty* = *Pair* (*bot-class.bot*)

definition *single* :: *'a* => *'a randompred*
where *single x* = *Pair* (*Predicate.single x*)

definition *bind* :: *'a randompred* ⇒ (*'a* ⇒ *'b randompred*) ⇒ *'b randompred*
where
bind R f = (λ*s*. *let*
(*P*, *s'*) = *R s*;
(*s1*, *s2*) = *Random.split-seed s'*
in (*Predicate.bind P* (%*a*. *fst* (*f a s1*)), *s2*))

definition *union* :: *'a randompred* ⇒ *'a randompred* ⇒ *'a randompred*
where
union R1 R2 = (λ*s*. *let*
(*P1*, *s'*) = *R1 s*; (*P2*, *s''*) = *R2 s'*
in (*semilattice-sup-class.sup P1 P2*, *s''*))

definition *if-randompred* :: *bool* ⇒ *unit randompred*
where
if-randompred b = (*if b then single* () *else empty*)

definition *iterate-upto* :: (*code-numeral* => *'a*) => *code-numeral* => *code-numeral* => *'a randompred*
where
iterate-upto f n m = *Pair* (*Code-Numeral.iterate-upto f n m*)

definition *not-randompred* :: *unit randompred* ⇒ *unit randompred*
where
not-randompred P = (λ*s*. *let*
(*P'*, *s'*) = *P s*
in if Predicate.eval P' () *then* (*Orderings.bot*, *s'*) *else* (*Predicate.single* (), *s'*))

definition *Random* :: (*Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*) ⇒

'a randompred
where *Random g = scomp g (Pair o (Predicate.single o fst))*

definition *map* :: *('a ⇒ 'b) ⇒ ('a randompred ⇒ 'b randompred)*
where *map f P = bind P (single o f)*

hide-fact (**open**) *iter'.simps iter-def empty-def single-def bind-def union-def if-randompred-def iterate-upto-def not-randompred-def Random-def map-def*
hide-type (**open**) *randompred*
hide-const (**open**) *random collapse beyond random-fun-aux random-fun-lift iter' iter empty single bind union if-randompred iterate-upto not-randompred Random map*

end

49 Lazy-Sequence: Lazy sequences

theory *Lazy-Sequence*
imports *List Code-Numeral*
begin

datatype *'a lazy-sequence = Empty | Insert 'a 'a lazy-sequence*

definition *Lazy-Sequence* :: *(unit => ('a * 'a lazy-sequence) option) => 'a lazy-sequence*
where
Lazy-Sequence f = (case f () of None => Empty | Some (x, xq) => Insert x xq)

code-datatype *Lazy-Sequence*

primrec *yield* :: *'a lazy-sequence => ('a * 'a lazy-sequence) option*
where
yield Empty = None
| *yield (Insert x xq) = Some (x, xq)*

lemma [*simp*]: *yield xq = Some (x, xq') ==> size xq' < size xq*
⟨*proof*⟩

lemma *yield-Seq* [*code*]:
yield (Lazy-Sequence f) = f ()
⟨*proof*⟩

lemma *Seq-yield*:
Lazy-Sequence (%u. yield f) = f
⟨*proof*⟩

lemma *lazy-sequence-size-code* [*code*]:
lazy-sequence-size s xq = (case yield xq of None => 0 | Some (x, xq') => s x + lazy-sequence-size s xq' + 1)

⟨*proof*⟩

lemma *size-code* [*code*]:
size xq = (*case yield xq of None* => *0* | *Some* (*x*, *xq'*) => *size xq'* + *1*)
⟨*proof*⟩

lemma [*code*]: *eq-class.eq xq yq* = (*case* (*yield xq*, *yield yq*) *of*
(*None*, *None*) => *True* | (*Some* (*x*, *xq'*), *Some* (*y*, *yq'*)) => (*HOL.eq x y*) ∧
(*eq-class.eq xq yq*) | - => *False*)
⟨*proof*⟩

lemma *seq-case* [*code*]:
lazy-sequence-case f g xq = (*case* (*yield xq*) *of None* => *f* | *Some* (*x*, *xq'*) => *g*
x xq')
⟨*proof*⟩

lemma [*code*]: *lazy-sequence-rec f g xq* = (*case* (*yield xq*) *of None* => *f* | *Some*
(*x*, *xq'*) => *g x xq'* (*lazy-sequence-rec f g xq'*))
⟨*proof*⟩

definition *empty* :: *'a lazy-sequence*
where
[*code*]: *empty* = *Lazy-Sequence* (%*u*. *None*)

definition *single* :: *'a* => *'a lazy-sequence*
where
[*code*]: *single x* = *Lazy-Sequence* (%*u*. *Some* (*x*, *empty*))

primrec *append* :: *'a lazy-sequence* => *'a lazy-sequence* => *'a lazy-sequence*
where
append Empty yq = *yq*
| *append* (*Insert x xq*) *yq* = *Insert x* (*append xq yq*)

lemma [*code*]:
append xq yq = *Lazy-Sequence* (%*u*. *case yield xq of*
None => *yield yq*
| *Some* (*x*, *xq'*) => *Some* (*x*, *append xq' yq*))
⟨*proof*⟩

primrec *flat* :: *'a lazy-sequence lazy-sequence* => *'a lazy-sequence*
where
flat Empty = *Empty*
| *flat* (*Insert xq xqq*) = *append xq* (*flat xqq*)

lemma [*code*]:
flat xqq = *Lazy-Sequence* (%*u*. *case yield xqq of*
None => *None*
| *Some* (*xq*, *xqq'*) => *yield* (*append xq* (*flat xqq'*)))
⟨*proof*⟩

primrec *map* :: (*'a => 'b*) *=> 'a lazy-sequence => 'b lazy-sequence*
where
map f Empty = Empty
| *map f (Insert x xq) = Insert (f x) (map f xq)*

lemma [*code*]:
map f xq = Lazy-Sequence (%u. Option.map (%(x, xq'). (f x, map f xq')) (yield xq))
⟨*proof*⟩

definition *bind* :: *'a lazy-sequence => ('a => 'b lazy-sequence) => 'b lazy-sequence*
where
[*code*]: *bind xq f = flat (map f xq)*

definition *if-seq* :: *bool => unit lazy-sequence*
where
if-seq b = (if b then single () else empty)

function *iterate-upto* :: (*code-numeral => 'a*) *=> code-numeral => code-numeral => 'a Lazy-Sequence.lazy-sequence*
where
iterate-upto f n m = Lazy-Sequence.Lazy-Sequence (%u. if n > m then None else Some (f n, iterate-upto f (n + 1) m))
⟨*proof*⟩

termination ⟨*proof*⟩

definition *not-seq* :: *unit lazy-sequence => unit lazy-sequence*
where
not-seq xq = (case yield xq of None => single () | Some ((), xq) => empty)

49.1 Code setup

fun *anamorph* :: (*'a ⇒ ('b × 'a) option*) *⇒ code-numeral ⇒ 'a ⇒ 'b list × 'a*
where
anamorph f k x = (if k = 0 then ([], x)
else case f x of None ⇒ ([], x) | Some (v, y) ⇒
let (vs, z) = anamorph f (k − 1) y
in (v # vs, z))

definition *yieldn* :: *code-numeral ⇒ 'a lazy-sequence ⇒ 'a list × 'a lazy-sequence*
where
yieldn = anamorph yield

code-reflect *Lazy-Sequence*
datatypes *lazy-sequence = Lazy-Sequence*
functions *map yield yieldn*

49.2 With Hit Bound Value

assuming in negative context

types *'a hit-bound-lazy-sequence = 'a option lazy-sequence*

definition *hit-bound :: 'a hit-bound-lazy-sequence*
where
[code]: hit-bound = Lazy-Sequence (%u. Some (None, empty))

definition *hb-single :: 'a => 'a hit-bound-lazy-sequence*
where
[code]: hb-single x = Lazy-Sequence (%u. Some (Some x, empty))

primrec *hb-flat :: 'a hit-bound-lazy-sequence hit-bound-lazy-sequence => 'a hit-bound-lazy-sequence*
where
hb-flat Empty = Empty
| hb-flat (Insert xq xqq) = append (case xq of None => hit-bound | Some xq => xq) (hb-flat xqq)

lemma *[code]:*
hb-flat xqq = Lazy-Sequence (%u. case yield xqq of
None => None
| Some (xq, xqq') => yield (append (case xq of None => hit-bound | Some xq => xq) (hb-flat xqq')))
⟨*proof*⟩

primrec *hb-map :: ('a => 'b) => 'a hit-bound-lazy-sequence => 'b hit-bound-lazy-sequence*
where
hb-map f Empty = Empty
| hb-map f (Insert x xq) = Insert (Option.map f x) (hb-map f xq)

lemma *[code]:*
hb-map f xq = Lazy-Sequence (%u. Option.map (%(x, xq'). (Option.map f x, hb-map f xq')) (yield xq))
⟨*proof*⟩

definition *hb-bind :: 'a hit-bound-lazy-sequence => ('a => 'b hit-bound-lazy-sequence) => 'b hit-bound-lazy-sequence*
where
[code]: hb-bind xq f = hb-flat (hb-map f xq)

definition *hb-if-seq :: bool => unit hit-bound-lazy-sequence*
where
hb-if-seq b = (if b then hb-single () else empty)

definition *hb-not-seq :: unit hit-bound-lazy-sequence => unit lazy-sequence*
where
hb-not-seq xq = (case yield xq of None => single () | Some (x, xq) => empty)

hide-type (**open**) *lazy-sequence*
hide-const (**open**) *Empty Insert Lazy-Sequence yield empty single append flat map bind if-seq iterate-upto not-seq*
hide-fact *yield.simps empty-def single-def append.simps flat.simps map.simps bind-def iterate-upto.simps if-seq-def not-seq-def*

end

50 DSequence: Depth-Limited Sequences with failure element

theory *DSequence*
imports *Lazy-Sequence Code-Numeral*
begin

types *'a dseq = code-numeral => bool => 'a Lazy-Sequence.lazy-sequence option*

definition *empty :: 'a dseq*
where
empty = (%i pol. Some Lazy-Sequence.empty)

definition *single :: 'a => 'a dseq*
where
single x = (%i pol. Some (Lazy-Sequence.single x))

fun *eval :: 'a dseq => code-numeral => bool => 'a Lazy-Sequence.lazy-sequence option*
where
eval f i pol = f i pol

definition *yield :: 'a dseq => code-numeral => bool => ('a * 'a dseq) option*
where
yield dseq i pol = (case eval dseq i pol of
None => None
| Some s => Option.map (apsnd (%r i pol. Some r)) (Lazy-Sequence.yield s))

fun *map-seq :: ('a => 'b dseq) => 'a Lazy-Sequence.lazy-sequence => 'b dseq*
where
map-seq f xq i pol = (case Lazy-Sequence.yield xq of
None => Some Lazy-Sequence.empty
| Some (x, xq') => (case eval (f x) i pol of
None => None
| Some yq => (case map-seq f xq' i pol of
None => None
| Some zq => Some (Lazy-Sequence.append yq zq))))

definition *bind* :: *'a dseq => ('a => 'b dseq) => 'b dseq*
where
bind x f = (%i pol.
if i = 0 then
(if pol then Some Lazy-Sequence.empty else None)
else
(case x (i − 1) pol of
None => None
| Some xq => map-seq f xq i pol))

definition *union* :: *'a dseq => 'a dseq => 'a dseq*
where
union x y = (%i pol. case (x i pol, y i pol) of
(Some xq, Some yq) => Some (Lazy-Sequence.append xq yq)
| - => None)

definition *if-seq* :: *bool => unit dseq*
where
if-seq b = (if b then single () else empty)

definition *not-seq* :: *unit dseq => unit dseq*
where
not-seq x = (%i pol. case x i (¬pol) of
None => Some Lazy-Sequence.empty
| Some xq => (case Lazy-Sequence.yield xq of
None => Some (Lazy-Sequence.single ())
| Some - => Some (Lazy-Sequence.empty)))

definition *map* :: *('a => 'b) => 'a dseq => 'b dseq*
where
map f g = (%i pol. case g i pol of
None => None
| Some xq => Some (Lazy-Sequence.map f xq))

⟨*ML*⟩

code-reserved *Eval DSequence*

hide-const (**open**) *empty single eval map-seq bind union if-seq not-seq map*
hide-fact (**open**) *empty-def single-def eval.simps map-seq.simps bind-def union-def*
if-seq-def not-seq-def map-def

end

theory *Random-Sequence*
imports *Quickcheck DSequence*

begin

types *'a random-dseq = code-numeral ⇒ code-numeral ⇒ Random.seed ⇒ ('a DSequence.dseq × Random.seed)*

definition *empty :: 'a random-dseq*
where
empty = (%nrandom size. Pair (DSequence.empty))

definition *single :: 'a => 'a random-dseq*
where
single x = (%nrandom size. Pair (DSequence.single x))

definition *bind :: 'a random-dseq => ('a ⇒ 'b random-dseq) ⇒ 'b random-dseq*
where
bind R f = (λnrandom size s. let
(P, s') = R nrandom size s;
(s1, s2) = Random.split-seed s'
in (DSequence.bind P (%a. fst (f a nrandom size s1)), s2))

definition *union :: 'a random-dseq => 'a random-dseq => 'a random-dseq*
where
union R1 R2 = (λnrandom size s. let
(S1, s') = R1 nrandom size s; (S2, s'') = R2 nrandom size s'
in (DSequence.union S1 S2, s''))

definition *if-random-dseq :: bool => unit random-dseq*
where
if-random-dseq b = (if b then single () else empty)

definition *not-random-dseq :: unit random-dseq => unit random-dseq*
where
not-random-dseq R = (λnrandom size s. let
(S, s') = R nrandom size s
in (DSequence.not-seq S, s'))

fun *Random :: (code-numeral ⇒ Random.seed ⇒ (('a × (unit ⇒ term)) × Random.seed)) ⇒ 'a random-dseq*
where
Random g nrandom = (%size. if nrandom <= 0 then (Pair DSequence.empty) else
(scomp (g size) (%r. scomp (Random g (nrandom − 1) size) (%rs. Pair (DSequence.union (DSequence.single (fst r)) rs)))))

definition *map :: ('a => 'b) => 'a random-dseq => 'b random-dseq*
where
map f P = bind P (single o f)

hide-const (**open**) *empty single bind union if-random-dseq not-random-dseq Random map*

hide-type *DSequence.dseq random-dseq*
hide-fact (**open**) *empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def Random.simps map-def*

end

51 New-DSequence: Depth-Limited Sequences with failure element

theory *New-DSequence*
imports *Random-Sequence*
begin

51.1 Positive Depth-Limited Sequence

types *'a pos-dseq = code-numeral => 'a Lazy-Sequence.lazy-sequence*

definition *pos-empty :: 'a pos-dseq*
where
pos-empty = (%i. Lazy-Sequence.empty)

definition *pos-single :: 'a => 'a pos-dseq*
where
pos-single x = (%i. Lazy-Sequence.single x)

definition *pos-bind :: 'a pos-dseq => ('a => 'b pos-dseq) => 'b pos-dseq*
where
pos-bind x f = (%i.
if i = 0 then
Lazy-Sequence.empty
else
Lazy-Sequence.bind (x (i − 1)) (%a. f a i))

definition *pos-union :: 'a pos-dseq => 'a pos-dseq => 'a pos-dseq*
where
pos-union xq yq = (%i. Lazy-Sequence.append (xq i) (yq i))

definition *pos-if-seq :: bool => unit pos-dseq*
where
pos-if-seq b = (if b then pos-single () else pos-empty)

definition *pos-iterate-upto :: (code-numeral => 'a) => code-numeral => code-numeral => 'a pos-dseq*
where

pos-iterate-upto f n m = (%i. Lazy-Sequence.iterate-upto f n m)

definition *pos-map :: ('a => 'b) => 'a pos-dseq => 'b pos-dseq*
where
pos-map f xq = (%i. Lazy-Sequence.map f (xq i))

51.2 Negative Depth-Limited Sequence

types *'a neg-dseq = code-numeral => 'a Lazy-Sequence.hit-bound-lazy-sequence*

definition *neg-empty :: 'a neg-dseq*
where
neg-empty = (%i. Lazy-Sequence.empty)

definition *neg-single :: 'a => 'a neg-dseq*
where
neg-single x = (%i. Lazy-Sequence.hb-single x)

definition *neg-bind :: 'a neg-dseq => ('a => 'b neg-dseq) => 'b neg-dseq*
where
neg-bind x f = (%i.
if i = 0 then
Lazy-Sequence.hit-bound
else
hb-bind (x (i − 1)) (%a. f a i))

definition *neg-union :: 'a neg-dseq => 'a neg-dseq => 'a neg-dseq*
where
neg-union x y = (%i. Lazy-Sequence.append (x i) (y i))

definition *neg-if-seq :: bool => unit neg-dseq*
where
neg-if-seq b = (if b then neg-single () else neg-empty)

definition *neg-iterate-upto*
where
neg-iterate-upto f n m = (%i. Lazy-Sequence.iterate-upto (%i. Some (f i)) n m)

definition *neg-map :: ('a => 'b) => 'a neg-dseq => 'b neg-dseq*
where
neg-map f xq = (%i. Lazy-Sequence.hb-map f (xq i))

51.3 Negation

definition *pos-not-seq :: unit neg-dseq => unit pos-dseq*
where
pos-not-seq xq = (%i. Lazy-Sequence.hb-not-seq (xq i))

definition *neg-not-seq :: unit pos-dseq => unit neg-dseq*
where

neg-not-seq x = (%i. case Lazy-Sequence.yield (x i) of
None => Lazy-Sequence.hb-single ()
| Some ((), xq) => Lazy-Sequence.empty)

hide-type (**open**) *pos-dseq neg-dseq*

hide-const (**open**)
pos-empty pos-single pos-bind pos-union pos-if-seq pos-iterate-upto pos-not-seq pos-map
neg-empty neg-single neg-bind neg-union neg-if-seq neg-iterate-upto neg-not-seq neg-map
hide-fact (**open**)
pos-empty-def pos-single-def pos-bind-def pos-union-def pos-if-seq-def pos-iterate-upto-def pos-not-seq-def pos-map-def
neg-empty-def neg-single-def neg-bind-def neg-union-def neg-if-seq-def neg-iterate-upto-def neg-not-seq-def neg-map-def

end

theory *New-Random-Sequence*
imports *Quickcheck New-DSequence*
begin

types *'a pos-random-dseq = code-numeral ⇒ code-numeral ⇒ Random.seed ⇒ 'a New-DSequence.pos-dseq*
types *'a neg-random-dseq = code-numeral ⇒ code-numeral ⇒ Random.seed ⇒ 'a New-DSequence.neg-dseq*

definition *pos-empty :: 'a pos-random-dseq*
where
pos-empty = (%nrandom size seed. New-DSequence.pos-empty)

definition *pos-single :: 'a => 'a pos-random-dseq*
where
pos-single x = (%nrandom size seed. New-DSequence.pos-single x)

definition *pos-bind :: 'a pos-random-dseq => ('a ⇒ 'b pos-random-dseq) ⇒ 'b pos-random-dseq*
where
pos-bind R f = (λnrandom size seed. New-DSequence.pos-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *pos-union :: 'a pos-random-dseq => 'a pos-random-dseq => 'a pos-random-dseq*
where
pos-union R1 R2 = (λnrandom size seed. New-DSequence.pos-union (R1 nrandom size seed) (R2 nrandom size seed))

definition *pos-if-random-dseq* :: *bool => unit pos-random-dseq*
where
pos-if-random-dseq b = (if b then pos-single () else pos-empty)

definition *pos-iterate-upto* :: *(code-numeral => 'a) => code-numeral => code-numeral => 'a pos-random-dseq*
where
pos-iterate-upto f n m = (λnrandom size seed. New-DSequence.pos-iterate-upto f n m)

definition *pos-not-random-dseq* :: *unit neg-random-dseq => unit pos-random-dseq*
where
pos-not-random-dseq R = (λnrandom size seed. New-DSequence.pos-not-seq (R nrandom size seed))

fun *iter* :: *(code-numeral * code-numeral => ('a * (unit => term)) * code-numeral * code-numeral) => code-numeral => code-numeral * code-numeral => 'a Lazy-Sequence.lazy-sequence*
where
iter random nrandom seed =
(if nrandom = 0 then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence (%u. let ((x, -), seed') = random seed in Some (x, iter random (nrandom − 1) seed')))

definition *Random* :: *(code-numeral ⇒ Random.seed ⇒ (('a × (unit ⇒ term)) × Random.seed)) ⇒ 'a pos-random-dseq*
where
Random g = (%nrandom size seed depth. iter (g size) nrandom seed)

definition *pos-map* :: *('a => 'b) => 'a pos-random-dseq => 'b pos-random-dseq*
where
pos-map f P = pos-bind P (pos-single o f)

definition *neg-empty* :: *'a neg-random-dseq*
where
neg-empty = (%nrandom size seed. New-DSequence.neg-empty)

definition *neg-single* :: *'a => 'a neg-random-dseq*
where
neg-single x = (%nrandom size seed. New-DSequence.neg-single x)

definition *neg-bind* :: *'a neg-random-dseq => ('a ⇒ 'b neg-random-dseq) ⇒ 'b neg-random-dseq*
where
neg-bind R f = (λnrandom size seed. New-DSequence.neg-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *neg-union* :: *'a neg-random-dseq => 'a neg-random-dseq => 'a neg-random-dseq*

where
neg-union R1 R2 = (λnrandom size seed. New-DSequence.neg-union (R1 nrandom size seed) (R2 nrandom size seed))

definition *neg-if-random-dseq :: bool => unit neg-random-dseq*
where
neg-if-random-dseq b = (if b then neg-single () else neg-empty)

definition *neg-iterate-upto :: (code-numeral => 'a) => code-numeral => code-numeral => 'a neg-random-dseq*
where
neg-iterate-upto f n m = (λnrandom size seed. New-DSequence.neg-iterate-upto f n m)

definition *neg-not-random-dseq :: unit pos-random-dseq => unit neg-random-dseq*
where
neg-not-random-dseq R = (λnrandom size seed. New-DSequence.neg-not-seq (R nrandom size seed))

definition *neg-map :: ('a => 'b) => 'a neg-random-dseq => 'b neg-random-dseq*
where
neg-map f P = neg-bind P (neg-single o f)

hide-const (**open**)
pos-empty pos-single pos-bind pos-union pos-if-random-dseq pos-iterate-upto pos-not-random-dseq iter Random pos-map
neg-empty neg-single neg-bind neg-union neg-if-random-dseq neg-iterate-upto neg-not-random-dseq neg-map
hide-type *New-DSequence.pos-dseq New-DSequence.neg-dseq pos-random-dseq neg-random-dseq*

end

52 Predicate-Compile: A compiler for predicates defined by introduction rules

theory *Predicate-Compile*
imports *New-Random-Sequence*
uses
Tools/Predicate-Compile/predicate-compile-aux.ML
Tools/Predicate-Compile/predicate-compile-compilations.ML
Tools/Predicate-Compile/predicate-compile-core.ML
Tools/Predicate-Compile/predicate-compile-data.ML
Tools/Predicate-Compile/predicate-compile-fun.ML
Tools/Predicate-Compile/predicate-compile-pred.ML
Tools/Predicate-Compile/predicate-compile-specialisation.ML

Tools/Predicate-Compile/predicate-compile.ML
begin

⟨*ML*⟩

end

53 Map: Maps

theory *Map*
imports *List*
begin

types ('*a*,'*b*) *map* = '*a* => '*b option* (**infixr** ~=> *0*)
translations (*type*) '*a* ~=> '*b* <= (*type*) '*a* => '*b option*

type-notation (*xsymbols*)
map (**infixr** ⇀ *0*)

abbreviation
empty :: '*a* ~=> '*b* **where**
empty == %*x*. *None*

definition
map-comp :: ('*b* ~=> '*c*) => ('*a* ~=> '*b*) => ('*a* ~=> '*c*) (**infixl** *o'-m 55*)
where
f o-m g = (λ*k*. *case g k of None* ⇒ *None* | *Some v* ⇒ *f v*)

notation (*xsymbols*)
map-comp (**infixl** ∘$_m$ *55*)

definition
map-add :: ('*a* ~=> '*b*) => ('*a* ~=> '*b*) => ('*a* ~=> '*b*) (**infixl** ++ *100*)
where
m1 ++ *m2* = (λ*x*. *case m2 x of None => m1 x* | *Some y => Some y*)

definition
restrict-map :: ('*a* ~=> '*b*) => '*a set* => ('*a* ~=> '*b*) (**infixl** |` *110*) **where**
m|`*A* = (λ*x*. *if x* : *A then m x else None*)

notation (*latex* **output**)
restrict-map (-|$_-$ [*111*,*110*] *110*)

definition
dom :: ('*a* ~=> '*b*) => '*a set* **where**
dom m = {*a*. *m a* ~= *None*}

definition

ran :: ('*a* ~=> '*b*) => '*b set* **where**
ran m = {*b*. *EX a*. *m a* = *Some b*}

definition
map-le :: ('*a* ~=> '*b*) => ('*a* ~=> '*b*) => *bool* (**infix** $\subseteq_m$ *50*) **where**
($m_1 \subseteq_m m_2$) = ($\forall a \in dom\ m_1.\ m_1\ a = m_2\ a$)

nonterminals
maplets maplet

syntax
-maplet :: ['*a*, '*a*] => *maplet* (- /|->/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[|->]/ -)
:: *maplet* => *maplets* (-)
-Maplets :: [*maplet*, *maplets*] => *maplets* (-,/ -)
-MapUpd :: ['*a* ~=> '*b*, *maplets*] => '*a* ~=> '*b* (-/'(-') [*900*,*0*]*900*)
-Map :: *maplets* => '*a* ~=> '*b* ((*1*[-]))

syntax (*xsymbols*)
-maplet :: ['*a*, '*a*] => *maplet* (- /↦/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[↦]/ -)

translations
-MapUpd m (*-Maplets xy ms*) == *-MapUpd* (*-MapUpd m xy*) *ms*
-MapUpd m (*-maplet x y*) == *m*(*x* := *CONST Some y*)
-Map ms == *-MapUpd* (*CONST empty*) *ms*
-Map (*-Maplets ms1 ms2*) <= *-MapUpd* (*-Map ms1*) *ms2*
-Maplets ms1 (*-Maplets ms2 ms3*) <= *-Maplets* (*-Maplets ms1 ms2*) *ms3*

primrec
map-of :: ('*a* × '*b*) *list* ⇒ '*a* ⇀ '*b* **where**
map-of [] = *empty*
| *map-of* (*p* # *ps*) = (*map-of ps*)(*fst p* ↦ *snd p*)

definition
map-upds :: ('*a* ⇀ '*b*) ⇒ '*a list* ⇒ '*b list* ⇒ '*a* ⇀ '*b* **where**
map-upds m xs ys = *m* ++ *map-of* (*rev* (*zip xs ys*))

translations
-MapUpd m (*-maplets x y*) == *CONST map-upds m x y*

lemma *map-of-Cons-code* [*code*]:
map-of [] *k* = *None*
map-of ((*l*, *v*) # *ps*) *k* = (*if l* = *k then Some v else map-of ps k*)
⟨*proof*⟩

53.1 *empty*

lemma *empty-upd-none* [*simp*]: *empty*(*x* := *None*) = *empty*

⟨*proof*⟩

53.2 *map-upd*

lemma *map-upd-triv*: $t\ k = Some\ x \Longrightarrow t(k|{->}x) = t$
⟨*proof*⟩

lemma *map-upd-nonempty* [*simp*]: $t(k|{->}x) \neq empty$
⟨*proof*⟩

lemma *map-upd-eqD1*:
assumes $m(a{\mapsto}x) = n(a{\mapsto}y)$
shows $x = y$
⟨*proof*⟩

lemma *map-upd-Some-unfold*:
$((m(a|{->}b))\ x = Some\ y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = Some\ y)$
⟨*proof*⟩

lemma *image-map-upd* [*simp*]: $x \notin A \Longrightarrow m(x \mapsto y)\ `\ A = m\ `\ A$
⟨*proof*⟩

lemma *finite-range-updI*: *finite* $(range\ f) \Longrightarrow$ *finite* $(range\ (f(a|{->}b)))$
⟨*proof*⟩

53.3 *map-of*

lemma *map-of-eq-None-iff*:
$(map\text{-}of\ xys\ x = None) = (x \notin fst\ `\ (set\ xys))$
⟨*proof*⟩

lemma *map-of-is-SomeD*: $map\text{-}of\ xys\ x = Some\ y \Longrightarrow (x,y) \in set\ xys$
⟨*proof*⟩

lemma *map-of-eq-Some-iff* [*simp*]:
$distinct(map\ fst\ xys) \Longrightarrow (map\text{-}of\ xys\ x = Some\ y) = ((x,y) \in set\ xys)$
⟨*proof*⟩

lemma *Some-eq-map-of-iff* [*simp*]:
$distinct(map\ fst\ xys) \Longrightarrow (Some\ y = map\text{-}of\ xys\ x) = ((x,y) \in set\ xys)$
⟨*proof*⟩

lemma *map-of-is-SomeI* [*simp*]: $[\![\ distinct(map\ fst\ xys);\ (x,y) \in set\ xys\ ]\!]$
$\Longrightarrow map\text{-}of\ xys\ x = Some\ y$
⟨*proof*⟩

lemma *map-of-zip-is-None* [*simp*]:
$length\ xs = length\ ys \Longrightarrow (map\text{-}of\ (zip\ xs\ ys)\ x = None) = (x \notin set\ xs)$
⟨*proof*⟩

lemma *map-of-zip-is-Some*:
assumes *length xs* = *length ys*
shows $x \in set\ xs \longleftrightarrow (\exists y.\ map\text{-}of\ (zip\ xs\ ys)\ x = Some\ y)$
⟨*proof*⟩

lemma *map-of-zip-upd*:
fixes $x :: 'a$ **and** $xs :: 'a\ list$ **and** $ys\ zs :: 'b\ list$
assumes *length ys* = *length xs*
and *length zs* = *length xs*
and $x \notin set\ xs$
and $map\text{-}of\ (zip\ xs\ ys)(x \mapsto y) = map\text{-}of\ (zip\ xs\ zs)(x \mapsto z)$
shows *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
⟨*proof*⟩

lemma *map-of-zip-inject*:
assumes *length ys* = *length xs*
and *length zs* = *length xs*
and *dist*: *distinct xs*
and *map-of*: *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
shows *ys* = *zs*
⟨*proof*⟩

lemma *map-of-zip-map*:
map-of (*zip xs* (*map f xs*)) = (λ*x*. *if x* ∈ *set xs then Some* (*f x*) *else None*)
⟨*proof*⟩

lemma *finite-range-map-of*: *finite* (*range* (*map-of xys*))
⟨*proof*⟩

lemma *map-of-SomeD*: *map-of xs k* = *Some y* ⟹ (*k*, *y*) ∈ *set xs*
⟨*proof*⟩

lemma *map-of-mapk-SomeI*:
inj f ==> *map-of t k* = *Some x* ==>
map-of (*map* (*split* (%*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
⟨*proof*⟩

lemma *weak-map-of-SomeI*: (*k*, *x*) : *set l* ==> ∃*x*. *map-of l k* = *Some x*
⟨*proof*⟩

lemma *map-of-filter-in*:
map-of xs k = *Some z* ⟹ *P k z* ⟹ *map-of* (*filter* (*split P*) *xs*) *k* = *Some z*
⟨*proof*⟩

lemma *map-of-map*:
map-of (*map* (λ(*k*, *v*). (*k*, *f v*)) *xs*) = *Option.map f* ∘ *map-of xs*
⟨*proof*⟩

lemma *dom-option-map*:

dom (λ*k*. *Option.map* (*f k*) (*m k*)) = *dom m*
⟨*proof*⟩

53.4 *Option.map* related

lemma *option-map-o-empty* [*simp*]: *Option.map f o empty* = *empty*
⟨*proof*⟩

lemma *option-map-o-map-upd* [*simp*]:
Option.map f o m(*a*|−>*b*) = (*Option.map f o m*)(*a*|−>*f b*)
⟨*proof*⟩

53.5 *map-comp* related

lemma *map-comp-empty* [*simp*]:
m ∘$_m$ *empty* = *empty*
empty ∘$_m$ *m* = *empty*
⟨*proof*⟩

lemma *map-comp-simps* [*simp*]:
m2 k = *None* ⟹ (*m1* ∘$_m$ *m2*) *k* = *None*
m2 k = *Some k′* ⟹ (*m1* ∘$_m$ *m2*) *k* = *m1 k′*
⟨*proof*⟩

lemma *map-comp-Some-iff*:
((*m1* ∘$_m$ *m2*) *k* = *Some v*) = (∃ *k′*. *m2 k* = *Some k′* ∧ *m1 k′* = *Some v*)
⟨*proof*⟩

lemma *map-comp-None-iff*:
((*m1* ∘$_m$ *m2*) *k* = *None*) = (*m2 k* = *None* ∨ (∃ *k′*. *m2 k* = *Some k′* ∧ *m1 k′* = *None*))
⟨*proof*⟩

53.6 ++

lemma *map-add-empty*[*simp*]: *m* ++ *empty* = *m*
⟨*proof*⟩

lemma *empty-map-add*[*simp*]: *empty* ++ *m* = *m*
⟨*proof*⟩

lemma *map-add-assoc*[*simp*]: *m1* ++ (*m2* ++ *m3*) = (*m1* ++ *m2*) ++ *m3*
⟨*proof*⟩

lemma *map-add-Some-iff*:
((*m* ++ *n*) *k* = *Some x*) = (*n k* = *Some x* | *n k* = *None* & *m k* = *Some x*)
⟨*proof*⟩

lemma *map-add-SomeD* [*dest*!]:
(*m* ++ *n*) *k* = *Some x* ⟹ *n k* = *Some x* ∨ *n k* = *None* ∧ *m k* = *Some x*

⟨*proof*⟩

lemma *map-add-find-right* [*simp*]: !!*xx*. *n k* = *Some xx* ==> (*m* ++ *n*) *k* = *Some xx*
⟨*proof*⟩

lemma *map-add-None* [*iff*]: ((*m* ++ *n*) *k* = *None*) = (*n k* = *None* & *m k* = *None*)
⟨*proof*⟩

lemma *map-add-upd*[*simp*]: *f* ++ *g*(*x*|->*y*) = (*f* ++ *g*)(*x*|->*y*)
⟨*proof*⟩

lemma *map-add-upds*[*simp*]: *m1* ++ (*m2*(*xs*[↦]*ys*)) = (*m1*++*m2*)(*xs*[↦]*ys*)
⟨*proof*⟩

lemma *map-add-upd-left*: *m*∉*dom e2* ⟹ *e1*(*m* ↦ *u1*) ++ *e2* = (*e1* ++ *e2*)(*m* ↦ *u1*)
⟨*proof*⟩

lemma *map-of-append*[*simp*]: *map-of* (*xs* @ *ys*) = *map-of ys* ++ *map-of xs*
⟨*proof*⟩

lemma *finite-range-map-of-map-add*:
finite (*range f*) ==> *finite* (*range* (*f* ++ *map-of l*))
⟨*proof*⟩

lemma *inj-on-map-add-dom* [*iff*]:
inj-on (*m* ++ *m'*) (*dom m'*) = *inj-on m'* (*dom m'*)
⟨*proof*⟩

lemma *map-upds-fold-map-upd*:
m(*ks*[↦]*vs*) = *foldl* (λ*m* (*k*, *v*). *m*(*k* ↦ *v*)) *m* (*zip ks vs*)
⟨*proof*⟩

lemma *map-add-map-of-foldr*:
m ++ *map-of ps* = *foldr* (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) *ps m*
⟨*proof*⟩

53.7 *restrict-map*

lemma *restrict-map-to-empty* [*simp*]: *m*|`{} = *empty*
⟨*proof*⟩

lemma *restrict-map-insert*: *f* |` (*insert a A*) = (*f* |` *A*)(*a* := *f a*)
⟨*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*⟩

lemma *map-of-map-restrict*:
map-of $(map\ (\lambda k.\ (k, f\ k))\ ks) = (Some \circ f)\ |`\ set\ ks$
⟨*proof*⟩

lemma *restrict-complement-singleton-eq*:
$f\ |`\ (-\ \{x\}) = f(x := None)$
⟨*proof*⟩

53.8 *map-upds*

lemma *map-upds-Nil1* [*simp*]: $m([]\ [|{-}{>}]\ bs) = m$
⟨*proof*⟩

lemma *map-upds-Nil2* [*simp*]: $m(as\ [|{-}{>}]\ []) = m$
⟨*proof*⟩

lemma *map-upds-Cons* [*simp*]: $m(a\#as\ [|->]\ b\#bs) = (m(a|->b))(as[|->]bs)$
⟨*proof*⟩

lemma *map-upds-append1* [*simp*]: $\bigwedge ys\ m.\ size\ xs < size\ ys \Longrightarrow$
$m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys)(x \mapsto ys!size\ xs)$
⟨*proof*⟩

lemma *map-upds-list-update2-drop* [*simp*]:
$[\![size\ xs \leq i;\ i < size\ ys]\!]$
$\Longrightarrow m(xs[\mapsto]ys[i:=y]) = m(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *map-upd-upds-conv-if*:
$(f(x|->y))(xs\ [|->]\ ys) =$
$(if\ x : set(take\ (length\ ys)\ xs)\ then\ f(xs\ [|->]\ ys)$
$else\ (f(xs\ [|->]\ ys))(x|->y))$
⟨*proof*⟩

lemma *map-upds-twist* [*simp*]:
$a\ ^{\sim}: set\ as\ ==>\ m(a|->b)(as[|->]bs) = m(as[|->]bs)(a|->b)$
⟨*proof*⟩

lemma *map-upds-apply-nontin* [*simp*]:
$x\ ^{\sim}: set\ xs\ ==>\ (f(xs[|->]ys))\ x = f\ x$
⟨*proof*⟩

lemma *fun-upds-append-drop* [*simp*]:
$size\ xs = size\ ys \Longrightarrow m(xs@zs[\mapsto]ys) = m(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *fun-upds-append2-drop* [*simp*]:
$size\ xs = size\ ys \Longrightarrow m(xs[\mapsto]ys@zs) = m(xs[\mapsto]ys)$
⟨*proof*⟩

lemma *restrict-map-upds*[*simp*]:
$[\![\ length\ xs = length\ ys;\ set\ xs \subseteq D\ ]\!]$
$\Longrightarrow m(xs\ [\mapsto]\ ys)|`D = (m|`(D - set\ xs))(xs\ [\mapsto]\ ys)$
⟨*proof*⟩

53.9 *dom*

lemma *dom-eq-empty-conv* [*simp*]: $dom\ f = \{\} \longleftrightarrow f = empty$
⟨*proof*⟩

lemma *domI*: $m\ a = Some\ b\ ==>\ a : dom\ m$
⟨*proof*⟩

lemma *domD*: $a : dom\ m \Longrightarrow \exists b.\ m\ a = Some\ b$
⟨*proof*⟩

lemma *domIff* [*iff*, *simp del*]: $(a : dom\ m) = (m\ a \sim= None)$
⟨*proof*⟩

lemma *dom-empty* [*simp*]: $dom\ empty = \{\}$
⟨*proof*⟩

lemma *dom-fun-upd* [*simp*]:
$dom(f(x := y)) = (if\ y{=}None\ then\ dom\ f - \{x\}\ else\ insert\ x\ (dom\ f))$
⟨*proof*⟩

lemma *dom-if*:
$dom\ (\lambda x.\ if\ P\ x\ then\ f\ x\ else\ g\ x) = dom\ f \cap \{x.\ P\ x\} \cup dom\ g \cap \{x.\ \neg\ P\ x\}$
⟨*proof*⟩

lemma *dom-map-of-conv-image-fst*:
$dom\ (map\text{-}of\ xys) = fst\ `\ set\ xys$
⟨*proof*⟩

lemma *dom-map-of-zip* [*simp*]: $[|\ length\ xs = length\ ys;\ distinct\ xs\ |] \Longrightarrow$
$dom(map\text{-}of(zip\ xs\ ys)) = set\ xs$
⟨*proof*⟩

lemma *finite-dom-map-of*: $finite\ (dom\ (map\text{-}of\ l))$
⟨*proof*⟩

lemma *dom-map-upds* [*simp*]:
$dom(m(xs[|{-}{>}]ys)) = set(take\ (length\ ys)\ xs)\ Un\ dom\ m$
⟨*proof*⟩

lemma *dom-map-add* [*simp*]: $dom(m{+}{+}n) = dom\ n\ Un\ dom\ m$
⟨*proof*⟩

lemma *dom-override-on* [*simp*]:
$dom(override\text{-}on\ f\ g\ A) =$
$(dom\ f\ - \{a.\ a : A - dom\ g\})\ Un\ \{a.\ a : A\ Int\ dom\ g\}$
⟨*proof*⟩

lemma *map-add-comm*: $dom\ m1 \cap dom\ m2 = \{\} \Longrightarrow m1{+}{+}m2 = m2{+}{+}m1$
⟨*proof*⟩

lemma *map-add-dom-app-simps*:
$[\![\ m{\in}dom\ l2\ ]\!] \Longrightarrow (l1{+}{+}l2)\ m = l2\ m$
$[\![\ m{\notin}dom\ l1\ ]\!] \Longrightarrow (l1{+}{+}l2)\ m = l2\ m$
$[\![\ m{\notin}dom\ l2\ ]\!] \Longrightarrow (l1{+}{+}l2)\ m = l1\ m$
⟨*proof*⟩

lemma *dom-const* [*simp*]:
dom (λ*x*. *Some* (*f x*)) = *UNIV*
⟨*proof*⟩

lemma *finite-map-freshness*:
finite (*dom* (*f* :: *'a* ⇀ *'b*)) ⟹ ¬ *finite* (*UNIV* :: *'a set*) ⟹
∃ *x*. *f x* = *None*
⟨*proof*⟩

lemma *dom-minus*:
f x = *None* ⟹ *dom f* − *insert x A* = *dom f* − *A*
⟨*proof*⟩

lemma *insert-dom*:
f x = *Some y* ⟹ *insert x* (*dom f*) = *dom f*
⟨*proof*⟩

lemma *map-of-map-keys*:
set xs = *dom m* ⟹ *map-of* (*map* (λ*k*. (*k*, *the* (*m k*))) *xs*) = *m*
⟨*proof*⟩

53.10 *ran*

lemma *ranI*: *m a* = *Some b* ==> *b* : *ran m*
⟨*proof*⟩

lemma *ran-empty* [*simp*]: *ran empty* = {}
⟨*proof*⟩

lemma *ran-map-upd* [*simp*]: *m a* = *None* ==> *ran*(*m*(*a*|−>*b*)) = *insert b* (*ran m*)
⟨*proof*⟩

lemma *ran-distinct*:
assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd* ‘ *set al*
⟨*proof*⟩

53.11 *map-le*

lemma *map-le-empty* [*simp*]: *empty* ⊆$_m$ *g*
⟨*proof*⟩

lemma *upd-None-map-le* [*simp*]: *f*(*x* := *None*) ⊆$_m$ *f*
⟨*proof*⟩

lemma *map-le-upd*[*simp*]: *f* ⊆$_m$ *g* ==> *f*(*a* := *b*) ⊆$_m$ *g*(*a* := *b*)

⟨*proof*⟩

lemma *map-le-imp-upd-le* [*simp*]: $m1 \subseteq_m m2 \Longrightarrow m1(x := None) \subseteq_m m2(x \mapsto y)$
⟨*proof*⟩

lemma *map-le-upds* [*simp*]:
$f \subseteq_m g \Longrightarrow f(as\ [|->]\ bs) \subseteq_m g(as\ [|->]\ bs)$
⟨*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*⟩

lemma *dom-eq-singleton-conv*: $dom\ f = \{x\} \longleftrightarrow (\exists\, v.\ f = [x \mapsto v])$
⟨*proof*⟩

53.12 Various

lemma *set-map-of-compr*:
assumes *distinct*: *distinct* (*map fst xs*)
shows *set xs* = {(*k*, *v*). *map-of xs k* = *Some v*}
⟨*proof*⟩

lemma *map-of-inject-set*:
assumes *distinct*: *distinct* (*map fst xs*) *distinct* (*map fst ys*)
shows *map-of xs* = *map-of ys* ⟷ *set xs* = *set ys* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

end

54 Quotient: Definition of Quotient Types

theory *Quotient*
imports *Plain Sledgehammer*
uses
(~~/*src*/*HOL*/*Tools*/*Quotient*/*quotient-info.ML*)
(~~/*src*/*HOL*/*Tools*/*Quotient*/*quotient-typ.ML*)
(~~/*src*/*HOL*/*Tools*/*Quotient*/*quotient-def.ML*)
(~~/*src*/*HOL*/*Tools*/*Quotient*/*quotient-term.ML*)
(~~/*src*/*HOL*/*Tools*/*Quotient*/*quotient-tacs.ML*)
begin

Basic definition for equivalence relations that are represented by predicates.

definition
$equivp\ E \equiv \forall x\ y.\ E\ x\ y = (E\ x = E\ y)$

definition
$reflp\ E \equiv \forall x.\ E\ x\ x$

definition
$symp\ E \equiv \forall x\ y.\ E\ x\ y \longrightarrow E\ y\ x$

definition
$transp\ E \equiv \forall x\ y\ z.\ E\ x\ y \wedge E\ y\ z \longrightarrow E\ x\ z$

lemma *equivp-reflp-symp-transp*:
shows $equivp\ E = (reflp\ E \wedge symp\ E \wedge transp\ E)$
⟨*proof*⟩

lemma *equivp-reflp*:
shows $equivp\ E \Longrightarrow E\ x\ x$
⟨*proof*⟩

lemma *equivp-symp*:
shows $equivp\ E \Longrightarrow E\ x\ y \Longrightarrow E\ y\ x$
⟨*proof*⟩

lemma *equivp-transp*:
shows $equivp\ E \Longrightarrow E\ x\ y \Longrightarrow E\ y\ z \Longrightarrow E\ x\ z$
⟨*proof*⟩

lemma *equivpI*:
assumes *reflp R symp R transp R*
shows *equivp R*
⟨*proof*⟩

lemma *identity-equivp*:
shows *equivp* (*op* =)
⟨*proof*⟩

Partial equivalences: not yet used anywhere

definition
part-equivp E ≡ (∃ *x*. *E x x*) ∧ (∀ *x y*. *E x y* = (*E x x* ∧ *E y y* ∧ (*E x* = *E y*)))

lemma *equivp-implies-part-equivp*:
assumes *a*: *equivp E*
shows *part-equivp E*
⟨*proof*⟩

Composition of Relations

abbreviation
rel-conj (**infixr** *OOO* 75)
where
r1 OOO r2 ≡ *r1 OO r2 OO r1*

lemma *eq-comp-r*:
shows ((*op* =) *OOO R*) = *R*
⟨*proof*⟩

54.1 Respects predicate

definition
Respects
where
Respects R x ≡ *R x x*

lemma *in-respects*:
shows (*x* ∈ *Respects R*) = *R x x*
⟨*proof*⟩

54.2 Function map and function relation

definition
fun-map (**infixr** −−−> 55)
where
[*simp*]: *fun-map f g h x* = *g* (*h* (*f x*))

definition
fun-rel (**infixr** ===> 55)
where
[*simp*]: *fun-rel E1 E2 f g* = (∀ *x y*. *E1 x y* ⟶ *E2* (*f x*) (*g y*))

lemma *fun-relI* [*intro*]:
assumes ⋀*a b*. *P a b* ⟹ *Q* (*x a*) (*y b*)
shows (*P* ===> *Q*) *x y*

⟨*proof*⟩

lemma *fun-map-id*:
shows $(id \text{ }---> id) = id$
⟨*proof*⟩

lemma *fun-rel-eq*:
shows $((op =) ===> (op =)) = (op =)$
⟨*proof*⟩

lemma *fun-rel-id*:
assumes *a*: $\bigwedge x\ y.\ R1\ x\ y \Longrightarrow R2\ (f\ x)\ (g\ y)$
shows $(R1 ===> R2)\ f\ g$
⟨*proof*⟩

lemma *fun-rel-id-asm*:
assumes *a*: $\bigwedge x\ y.\ R1\ x\ y \Longrightarrow (A \longrightarrow R2\ (f\ x)\ (g\ y))$
shows $A \longrightarrow (R1 ===> R2)\ f\ g$
⟨*proof*⟩

54.3 Quotient Predicate

definition
$Quotient\ E\ Abs\ Rep \equiv$
$(\forall a.\ Abs\ (Rep\ a) = a) \wedge (\forall a.\ E\ (Rep\ a)\ (Rep\ a)) \wedge$
$(\forall r\ s.\ E\ r\ s = (E\ r\ r \wedge E\ s\ s \wedge (Abs\ r = Abs\ s)))$

lemma *Quotient-abs-rep*:
assumes *a*: $Quotient\ E\ Abs\ Rep$
shows $Abs\ (Rep\ a) = a$
⟨*proof*⟩

lemma *Quotient-rep-reflp*:
assumes *a*: $Quotient\ E\ Abs\ Rep$
shows $E\ (Rep\ a)\ (Rep\ a)$
⟨*proof*⟩

lemma *Quotient-rel*:
assumes *a*: $Quotient\ E\ Abs\ Rep$
shows $E\ r\ s = (E\ r\ r \wedge E\ s\ s \wedge (Abs\ r = Abs\ s))$
⟨*proof*⟩

lemma *Quotient-rel-rep*:
assumes *a*: $Quotient\ R\ Abs\ Rep$
shows $R\ (Rep\ a)\ (Rep\ b) = (a = b)$
⟨*proof*⟩

lemma *Quotient-rep-abs*:
assumes *a*: $Quotient\ R\ Abs\ Rep$

shows $R\ r\ r \Longrightarrow R\ (Rep\ (Abs\ r))\ r$
⟨*proof*⟩

lemma *Quotient-rel-abs*:
assumes *a*: $Quotient\ E\ Abs\ Rep$
shows $E\ r\ s \Longrightarrow Abs\ r = Abs\ s$
⟨*proof*⟩

lemma *Quotient-symp*:
assumes *a*: $Quotient\ E\ Abs\ Rep$
shows $symp\ E$
⟨*proof*⟩

lemma *Quotient-transp*:
assumes *a*: $Quotient\ E\ Abs\ Rep$
shows $transp\ E$
⟨*proof*⟩

lemma *identity-quotient*:
shows $Quotient\ (op =)\ id\ id$
⟨*proof*⟩

lemma *fun-quotient*:
assumes *q1*: $Quotient\ R1\ abs1\ rep1$
and *q2*: $Quotient\ R2\ abs2\ rep2$
shows $Quotient\ (R1 ===> R2)\ (rep1 ---> abs2)\ (abs1 ---> rep2)$
⟨*proof*⟩

lemma *abs-o-rep*:
assumes *a*: $Quotient\ R\ Abs\ Rep$
shows $Abs\ o\ Rep = id$
⟨*proof*⟩

lemma *equals-rsp*:
assumes *q*: $Quotient\ R\ Abs\ Rep$
and *a*: $R\ xa\ xb\ R\ ya\ yb$
shows $R\ xa\ ya = R\ xb\ yb$
⟨*proof*⟩

lemma *lambda-prs*:
assumes *q1*: $Quotient\ R1\ Abs1\ Rep1$
and *q2*: $Quotient\ R2\ Abs2\ Rep2$
shows $(Rep1 ---> Abs2)\ (\lambda x.\ Rep2\ (f\ (Abs1\ x))) = (\lambda x.\ f\ x)$
⟨*proof*⟩

lemma *lambda-prs1*:
assumes *q1*: $Quotient\ R1\ Abs1\ Rep1$
and *q2*: $Quotient\ R2\ Abs2\ Rep2$
shows $(Rep1 ---> Abs2)\ (\lambda x.\ (Abs1 ---> Rep2)\ f\ x) = (\lambda x.\ f\ x)$

⟨*proof*⟩

lemma *rep-abs-rsp*:
assumes *q*: *Quotient R Abs Rep*
and *a*: *R x1 x2*
shows *R x1* (*Rep* (*Abs x2*))
⟨*proof*⟩

lemma *rep-abs-rsp-left*:
assumes *q*: *Quotient R Abs Rep*
and *a*: *R x1 x2*
shows *R* (*Rep* (*Abs x1*)) *x2*
⟨*proof*⟩

In the following theorem R1 can be instantiated with anything, but we know some of the types of the Rep and Abs functions; so by solving Quotient assumptions we can get a unique R1 that will be provable; which is why we need to use *apply-rsp* and not the primed version

lemma *apply-rsp*:
fixes *f g*::*'a* ⇒ *'c*
assumes *q*: *Quotient R1 Abs1 Rep1*
and *a*: (*R1* ===> *R2*) *f g R1 x y*
shows *R2* (*f x*) (*g y*)
⟨*proof*⟩

lemma *apply-rsp'*:
assumes *a*: (*R1* ===> *R2*) *f g R1 x y*
shows *R2* (*f x*) (*g y*)
⟨*proof*⟩

54.4 lemmas for regularisation of ball and bex

lemma *ball-reg-eqv*:
fixes *P* :: *'a* ⇒ *bool*
assumes *a*: *equivp R*
shows *Ball* (*Respects R*) *P* = (*All P*)
⟨*proof*⟩

lemma *bex-reg-eqv*:
fixes *P* :: *'a* ⇒ *bool*
assumes *a*: *equivp R*
shows *Bex* (*Respects R*) *P* = (*Ex P*)
⟨*proof*⟩

lemma *ball-reg-right*:
assumes *a*: ⋀*x*. *R x* ⟹ *P x* ⟶ *Q x*
shows *All P* ⟶ *Ball R Q*
⟨*proof*⟩

lemma *bex-reg-left*:
assumes *a*: $\bigwedge x.\ R\ x \Longrightarrow Q\ x \longrightarrow P\ x$
shows $Bex\ R\ Q \longrightarrow Ex\ P$
⟨*proof*⟩

lemma *ball-reg-left*:
assumes *a*: *equivp R*
shows $(\bigwedge x.\ (Q\ x \longrightarrow P\ x)) \Longrightarrow Ball\ (Respects\ R)\ Q \longrightarrow All\ P$
⟨*proof*⟩

lemma *bex-reg-right*:
assumes *a*: *equivp R*
shows $(\bigwedge x.\ (Q\ x \longrightarrow P\ x)) \Longrightarrow Ex\ Q \longrightarrow Bex\ (Respects\ R)\ P$
⟨*proof*⟩

lemma *ball-reg-eqv-range*:
fixes $P::'a \Rightarrow bool$
and $x::'a$
assumes *a*: *equivp R2*
shows $(Ball\ (Respects\ (R1 ===> R2))\ (\lambda f.\ P\ (f\ x)) = All\ (\lambda f.\ P\ (f\ x)))$
⟨*proof*⟩

lemma *bex-reg-eqv-range*:
assumes *a*: *equivp R2*
shows $(Bex\ (Respects\ (R1 ===> R2))\ (\lambda f.\ P\ (f\ x)) = Ex\ (\lambda f.\ P\ (f\ x)))$
⟨*proof*⟩

lemma *all-reg*:
assumes *a*: $!x :: 'a.\ (P\ x\ --> Q\ x)$
and *b*: *All P*
shows *All Q*
⟨*proof*⟩

lemma *ex-reg*:
assumes *a*: $!x :: 'a.\ (P\ x\ --> Q\ x)$
and *b*: *Ex P*
shows *Ex Q*
⟨*proof*⟩

lemma *ball-reg*:
assumes *a*: $!x :: 'a.\ (R\ x\ --> P\ x\ --> Q\ x)$
and *b*: *Ball R P*
shows *Ball R Q*
⟨*proof*⟩

lemma *bex-reg*:
assumes *a*: $!x :: 'a.\ (R\ x\ --> P\ x\ --> Q\ x)$
and *b*: *Bex R P*

shows *Bex R Q*
⟨*proof*⟩

lemma *ball-all-comm*:
assumes $\bigwedge y.\ (\forall x{\in}P.\ A\ x\ y) \longrightarrow (\forall x.\ B\ x\ y)$
shows $(\forall x{\in}P.\ \forall y.\ A\ x\ y) \longrightarrow (\forall x.\ \forall y.\ B\ x\ y)$
⟨*proof*⟩

lemma *bex-ex-comm*:
assumes $(\exists y.\ \exists x.\ A\ x\ y) \longrightarrow (\exists y.\ \exists x{\in}P.\ B\ x\ y)$
shows $(\exists x.\ \exists y.\ A\ x\ y) \longrightarrow (\exists x{\in}P.\ \exists y.\ B\ x\ y)$
⟨*proof*⟩

54.5 Bounded abstraction

definition
Babs :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where
$x \in p \Longrightarrow Babs\ p\ m\ x = m\ x$

lemma *babs-rsp*:
assumes *q*: *Quotient R1 Abs1 Rep1*
and *a*: *(R1 ===> R2) f g*
shows *(R1 ===> R2) (Babs (Respects R1) f) (Babs (Respects R1) g)*
⟨*proof*⟩

lemma *babs-prs*:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows *((Rep1 ---> Abs2) (Babs (Respects R1) ((Abs1 ---> Rep2) f))) = f*
⟨*proof*⟩

lemma *babs-simp*:
assumes *q*: *Quotient R1 Abs Rep*
shows *((R1 ===> R2) (Babs (Respects R1) f) (Babs (Respects R1) g)) = ((R1 ===> R2) f g)*
⟨*proof*⟩

lemma *babs-reg-eqv*:
shows *equivp R* $\Longrightarrow$ *Babs (Respects R) P = P*
⟨*proof*⟩

lemma *ball-rsp*:
assumes *a*: *(R ===> (op =)) f g*

shows *Ball* (*Respects R*) *f* = *Ball* (*Respects R*) *g*
⟨*proof*⟩

lemma *bex-rsp*:
assumes *a*: (*R* ===> (*op* =)) *f g*
shows (*Bex* (*Respects R*) *f* = *Bex* (*Respects R*) *g*)
⟨*proof*⟩

lemma *bex1-rsp*:
assumes *a*: (*R* ===> (*op* =)) *f g*
shows *Ex1* (λ*x*. *x* ∈ *Respects R* ∧ *f x*) = *Ex1* (λ*x*. *x* ∈ *Respects R* ∧ *g x*)
⟨*proof*⟩

lemma *all-prs*:
assumes *a*: *Quotient R absf repf*
shows *Ball* (*Respects R*) ((*absf* −−−> *id*) *f*) = *All f*
⟨*proof*⟩

lemma *ex-prs*:
assumes *a*: *Quotient R absf repf*
shows *Bex* (*Respects R*) ((*absf* −−−> *id*) *f*) = *Ex f*
⟨*proof*⟩

54.6 *Bex1-rel* quantifier

definition
Bex1-rel :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*a* ⇒ *bool*) ⇒ *bool*
where
Bex1-rel R P ⟷ (∃ *x* ∈ *Respects R*. *P x*) ∧ (∀ *x* ∈ *Respects R*. ∀ *y* ∈ *Respects R*. ((*P x* ∧ *P y*) ⟶ (*R x y*)))

lemma *bex1-rel-aux*:
⟦∀ *xa ya*. *R xa ya* ⟶ *x xa* = *y ya*; *Bex1-rel R x*⟧ ⟹ *Bex1-rel R y*
⟨*proof*⟩

lemma *bex1-rel-aux2*:
⟦∀ *xa ya*. *R xa ya* ⟶ *x xa* = *y ya*; *Bex1-rel R y*⟧ ⟹ *Bex1-rel R x*
⟨*proof*⟩

lemma *bex1-rel-rsp*:
assumes *a*: *Quotient R absf repf*
shows ((*R* ===> *op* =) ===> *op* =) (*Bex1-rel R*) (*Bex1-rel R*)
⟨*proof*⟩

lemma *ex1-prs*:
assumes *a*: *Quotient R absf repf*
shows ((*absf* −−−> *id*) −−−> *id*) (*Bex1-rel R*) *f* = *Ex1 f*

⟨*proof*⟩

lemma *bex1-bexeq-reg*: (∃!x∈*Respects R*. *P x*) ⟶ (*Bex1-rel R* (λx. *P x*))
⟨*proof*⟩

54.7 Various respects and preserve lemmas

lemma *quot-rel-rsp*:
assumes *a*: *Quotient R Abs Rep*
shows (*R* ===> *R* ===> *op* =) *R R*
⟨*proof*⟩

lemma *o-prs*:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
and *q3*: *Quotient R3 Abs3 Rep3*
shows ((*Abs2* ---> *Rep3*) ---> (*Abs1* ---> *Rep2*) ---> (*Rep1* ---> *Abs3*)) *op* ∘ = *op* ∘
and (*id* ---> (*Abs1* ---> *id*) ---> *Rep1* ---> *id*) *op* ∘ = *op* ∘
⟨*proof*⟩

lemma *o-rsp*:
((*R2* ===> *R3*) ===> (*R1* ===> *R2*) ===> (*R1* ===> *R3*)) *op* ∘ *op* ∘
(*op* = ===> (*R1* ===> *op* =) ===> *R1* ===> *op* =) *op* ∘ *op* ∘
⟨*proof*⟩

lemma *cond-prs*:
assumes *a*: *Quotient R absf repf*
shows *absf* (*if a then repf b else repf c*) = (*if a then b else c*)
⟨*proof*⟩

lemma *if-prs*:
assumes *q*: *Quotient R Abs Rep*
shows (*id* ---> *Rep* ---> *Rep* ---> *Abs*) *If* = *If*
⟨*proof*⟩

lemma *if-rsp*:
assumes *q*: *Quotient R Abs Rep*
shows (*op* = ===> *R* ===> *R* ===> *R*) *If If*
⟨*proof*⟩

lemma *let-prs*:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows (*Rep2* ---> (*Abs2* ---> *Rep1*) ---> *Abs1*) *Let* = *Let*
⟨*proof*⟩

lemma *let-rsp*:
shows (*R1* ===> (*R1* ===> *R2*) ===> *R2*) *Let Let*

⟨*proof*⟩

locale *quot-type* =
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$
and $Abs :: ('a \Rightarrow bool) \Rightarrow 'b$
and $Rep :: 'b \Rightarrow ('a \Rightarrow bool)$
assumes *equivp*: $equivp\ R$
and *rep-prop*: $\bigwedge y.\ \exists x.\ Rep\ y = R\ x$
and *rep-inverse*: $\bigwedge x.\ Abs\ (Rep\ x) = x$
and *abs-inverse*: $\bigwedge x.\ (Rep\ (Abs\ (R\ x))) = (R\ x)$
and *rep-inject*: $\bigwedge x\ y.\ (Rep\ x = Rep\ y) = (x = y)$
begin

definition
$abs :: 'a \Rightarrow 'b$
where
$abs\ x \equiv Abs\ (R\ x)$

definition
$rep :: 'b \Rightarrow 'a$
where
$rep\ a = Eps\ (Rep\ a)$

lemma *homeier-lem9*:
shows $R\ (Eps\ (R\ x)) = R\ x$
⟨*proof*⟩

theorem *homeier-thm10*:
shows $abs\ (rep\ a) = a$
⟨*proof*⟩

lemma *homeier-lem7*:
shows $(R\ x = R\ y) = (Abs\ (R\ x) = Abs\ (R\ y))$ (**is** *?LHS* = *?RHS*)
⟨*proof*⟩

theorem *homeier-thm11*:
shows $R\ r\ r' = (abs\ r = abs\ r')$
⟨*proof*⟩

lemma *rep-refl*:
shows $R\ (rep\ a)\ (rep\ a)$
⟨*proof*⟩

lemma *rep-abs-rsp*:
shows $R\ f\ (rep\ (abs\ g)) = R\ f\ g$
and $R\ (rep\ (abs\ g))\ f = R\ g\ f$
⟨*proof*⟩

lemma *Quotient*:
shows *Quotient R abs rep*
⟨*proof*⟩

end

54.8 ML setup

Auxiliary data for the quotient package

⟨*ML*⟩

declare [[*map fun* = (*fun-map*, *fun-rel*)]]

lemmas [*quot-thm*] = *fun-quotient*
lemmas [*quot-respect*] = *quot-rel-rsp if-rsp o-rsp let-rsp*
lemmas [*quot-preserve*] = *if-prs o-prs let-prs*
lemmas [*quot-equiv*] = *identity-equivp*

Lemmas about simplifying id's.

lemmas [*id-simps*] =
id-def[*symmetric*]
fun-map-id
id-apply
id-o
o-id
eq-comp-r

Translation functions for the lifting process.

⟨*ML*⟩

Definitions of the quotient types.

⟨*ML*⟩

Definitions for quotient constants.

⟨*ML*⟩

An auxiliary constant for recording some information about the lifted theorem in a tactic.

definition
Quot-True (*x* :: *'a*) ≡ *True*

lemma
shows *QT-all*: *Quot-True* (*All P*) ⟹ *Quot-True P*
and *QT-ex*: *Quot-True* (*Ex P*) ⟹ *Quot-True P*
and *QT-ex1*: *Quot-True* (*Ex1 P*) ⟹ *Quot-True P*
and *QT-lam*: *Quot-True* (λ*x*. *P x*) ⟹ (⋀*x*. *Quot-True* (*P x*))
and *QT-ext*: (⋀*x*. *Quot-True* (*a x*) ⟹ *f x* = *g x*) ⟹ (*Quot-True a* ⟹ *f* = *g*)
⟨*proof*⟩

lemma *QT-imp*: *Quot-True a* ≡ *Quot-True b*
⟨*proof*⟩

Tactics for proving the lifted theorems

⟨*ML*⟩

54.9 Methods / Interface

⟨*ML*⟩

no-notation
rel-conj (**infixr** *OOO 75*) **and**
fun-map (**infixr** *−−−> 55*) **and**
fun-rel (**infixr** *===> 55*)

end

55 Refute: Refute

theory *Refute*
imports *Hilbert-Choice List*
uses
Tools/refute.ML
Tools/refute-isar.ML
begin

⟨*ML*⟩

```
(* ------------------------------------------------------------------------- *)
(* REFUTE                                                                    *)
(*                                                                           *)
(* We use a SAT solver to search for a (finite) model that refutes a given   *)
(* HOL formula.                                                              *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* NOTE                                                                      *)
(*                                                                           *)
(* I strongly recommend that you install a stand-alone SAT solver if you     *)
(* want to use 'refute'.  For details see 'HOL/Tools/sat_solver.ML'.  If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME'  *)
(* in 'etc/settings'.                                                        *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
```

```
(* USAGE                                                                    *)
(*                                                                          *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples.  The supported   *)
(* parameters are explained below.                                          *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* CURRENT LIMITATIONS                                                      *)
(*                                                                          *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and          *)
(* inductively defined sets.  Constants are unfolded automatically, and sort *)
(* axioms are added as well.  Other, user-asserted axioms however are       *)
(* ignored.  Inductive datatypes and recursive functions are supported, but  *)
(* may lead to spurious countermodels.                                      *)
(*                                                                          *)
(* The (space) complexity of the algorithm is non-elementary.               *)
(*                                                                          *)
(* Schematic type variables are not supported.                              *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* PARAMETERS                                                               *)
(*                                                                          *)
(* The following global parameters are currently supported (and required,   *)
(* except for "expect"):                                                    *)
(*                                                                          *)
(* 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.                 *)
(* "no_assms"    bool    If "true", assumptions in structured proofs are    *)
(*                       not considered.                                    *)
(* "expect"      string  Expected result ("genuine", "potential", "none", or *)
(*                       "unknown").                                        *)
(*                                                                          *)
(* 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

56 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*,
no-assms=*false*]

⟨*ML*⟩

end

57 Nitpick: Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

theory *Nitpick*
imports *Map Quotient SAT*
uses (*Tools/Nitpick/kodkod.ML*)
(*Tools/Nitpick/kodkod-sat.ML*)
(*Tools/Nitpick/nitpick-util.ML*)
(*Tools/Nitpick/nitpick-hol.ML*)
(*Tools/Nitpick/nitpick-preproc.ML*)
(*Tools/Nitpick/nitpick-mono.ML*)
(*Tools/Nitpick/nitpick-scope.ML*)
(*Tools/Nitpick/nitpick-peephole.ML*)
(*Tools/Nitpick/nitpick-rep.ML*)
(*Tools/Nitpick/nitpick-nut.ML*)
(*Tools/Nitpick/nitpick-kodkod.ML*)
(*Tools/Nitpick/nitpick-model.ML*)
(*Tools/Nitpick/nitpick.ML*)
(*Tools/Nitpick/nitpick-isar.ML*)
(*Tools/Nitpick/nitpick-tests.ML*)
(*Tools/Nitpick/minipick.ML*)
begin

typedecl *bisim-iterator*

axiomatization *unknown* :: *'a*
and *is-unknown* :: *'a* ⇒ *bool*
and *undefined-fast-The* :: *'a*
and *undefined-fast-Eps* :: *'a*
and *bisim* :: *bisim-iterator* ⇒ *'a* ⇒ *'a* ⇒ *bool*
and *bisim-iterator-max* :: *bisim-iterator*
and *Quot* :: *'a* ⇒ *'b*
and *safe-The* :: (*'a* ⇒ *bool*) ⇒ *'a*
and *safe-Eps* :: (*'a* ⇒ *bool*) ⇒ *'a*

datatype (*'a*, *'b*) *fin-fun* = *FinFun* (*'a* ⇒ *'b*)
datatype (*'a*, *'b*) *fun-box* = *FunBox* (*'a* ⇒ *'b*)
datatype (*'a*, *'b*) *pair-box* = *PairBox* *'a* *'b*

typedecl *unsigned-bit*
typedecl *signed-bit*

datatype *'a word* = *Word* (*'a set*)

Alternative definitions.

lemma *If-def* [*nitpick-def*, *no-atp*]:
(*if* *P* *then* *Q* *else* *R*) ≡ (*P* ⟶ *Q*) ∧ (¬ *P* ⟶ *R*)
⟨*proof*⟩

lemma *Ex1-def* [*nitpick-def*, *no-atp*]:
$Ex1\ P \equiv \exists x.\ P = \{x\}$
⟨*proof*⟩

lemma *split-def* [*nitpick-def*]: $split\ f = (\lambda p.\ f\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

lemma *rtrancl-def* [*nitpick-def*, *no-atp*]: $r^* \equiv (r^+)^=$
⟨*proof*⟩

lemma *rtranclp-def* [*nitpick-def*, *no-atp*]:
$rtranclp\ r\ a\ b \equiv (a = b \vee tranclp\ r\ a\ b)$
⟨*proof*⟩

lemma *tranclp-def* [*nitpick-def*, *no-atp*]:
$tranclp\ r\ a\ b \equiv trancl\ (split\ r)\ (a, b)$
⟨*proof*⟩

definition $refl' :: ('a \times 'a \Rightarrow bool) \Rightarrow bool$ **where**
$refl'\ r \equiv \forall x.\ (x, x) \in r$

definition $wf' :: ('a \times 'a \Rightarrow bool) \Rightarrow bool$ **where**
$wf'\ r \equiv acyclic\ r \wedge (finite\ r \vee unknown)$

axiomatization *wf-wfrec* :: $('a \times 'a \Rightarrow bool) \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

definition $wf\text{-}wfrec' :: ('a \times 'a \Rightarrow bool) \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
[*nitpick-simp*]: $wf\text{-}wfrec'\ R\ F\ x = F\ (Recdef.cut\ (wf\text{-}wfrec\ R\ F)\ R\ x)\ x$

definition $wfrec' :: ('a \times 'a \Rightarrow bool) \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
$wfrec'\ R\ F\ x \equiv if\ wf\ R\ then\ wf\text{-}wfrec'\ R\ F\ x$
$\quad else\ THE\ y.\ wfrec\text{-}rel\ R\ (\%f\ x.\ F\ (Recdef.cut\ f\ R\ x)\ x)\ x\ y$

definition $card' :: ('a \Rightarrow bool) \Rightarrow nat$ **where**
$card'\ A \equiv if\ finite\ A\ then\ length\ (safe\text{-}Eps\ (\lambda xs.\ set\ xs = A \wedge distinct\ xs))\ else\ 0$

definition $setsum' :: ('a \Rightarrow 'b{::}comm\text{-}monoid\text{-}add) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b$ **where**
$setsum'\ f\ A \equiv if\ finite\ A\ then\ listsum\ (map\ f\ (safe\text{-}Eps\ (\lambda xs.\ set\ xs = A \wedge distinct\ xs)))\ else\ 0$

inductive $fold\text{-}graph' :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b \Rightarrow bool$ **where**
$fold\text{-}graph'\ f\ z\ \{\}\ z\ |$
$[\![x \in A;\ fold\text{-}graph'\ f\ z\ (A - \{x\})\ y]\!] \Longrightarrow fold\text{-}graph'\ f\ z\ A\ (f\ x\ y)$

The following lemmas are not strictly necessary but they help the *special_level* optimization.

lemma *The-psimp* [*nitpick-psimp*, *no-atp*]:

$P = \{x\} \Longrightarrow \mathit{The}\ P = x$
⟨*proof*⟩

lemma *Eps-psimp* [*nitpick-psimp*, *no-atp*]:
$[\![P\ x; \neg\ P\ y; \mathit{Eps}\ P = y]\!] \Longrightarrow \mathit{Eps}\ P = x$
⟨*proof*⟩

lemma *unit-case-def* [*nitpick-def*, *no-atp*]:
unit-case x u ≡ *x*
⟨*proof*⟩

declare *unit.cases* [*nitpick-simp del*]

lemma *nat-case-def* [*nitpick-def*, *no-atp*]:
nat-case x f n ≡ *if n* = *0 then x else f* (*n* − *1*)
⟨*proof*⟩

declare *nat.cases* [*nitpick-simp del*]

lemma *list-size-simp* [*nitpick-simp*, *no-atp*]:
list-size f xs = (*if xs* = [] *then 0*
else Suc (*f* (*hd xs*) + *list-size f* (*tl xs*)))
size xs = (*if xs* = [] *then 0 else Suc* (*size* (*tl xs*)))
⟨*proof*⟩

Auxiliary definitions used to provide an alternative representation for *rat* and *real*.

function *nat-gcd* :: *nat* ⇒ *nat* ⇒ *nat* **where**
[*simp del*]: *nat-gcd x y* = (*if y* = *0 then x else nat-gcd y* (*x mod y*))
⟨*proof*⟩
termination
⟨*proof*⟩

definition *nat-lcm* :: *nat* ⇒ *nat* ⇒ *nat* **where**
nat-lcm x y = *x* ∗ *y div* (*nat-gcd x y*)

definition *int-gcd* :: *int* ⇒ *int* ⇒ *int* **where**
int-gcd x y = *int* (*nat-gcd* (*nat* (*abs x*)) (*nat* (*abs y*)))

definition *int-lcm* :: *int* ⇒ *int* ⇒ *int* **where**
int-lcm x y = *int* (*nat-lcm* (*nat* (*abs x*)) (*nat* (*abs y*)))

definition *Frac* :: *int* × *int* ⇒ *bool* **where**
Frac ≡ λ(*a*, *b*). *b* > *0* ∧ *int-gcd a b* = *1*

axiomatization *Abs-Frac* :: *int* × *int* ⇒ *'a*
and *Rep-Frac* :: *'a* ⇒ *int* × *int*

definition *zero-frac* :: *'a* **where**

zero-frac ≡ *Abs-Frac* (*0*, *1*)

definition *one-frac* :: *'a* **where**
one-frac ≡ *Abs-Frac* (*1*, *1*)

definition *num* :: *'a* ⇒ *int* **where**
num ≡ *fst o Rep-Frac*

definition *denom* :: *'a* ⇒ *int* **where**
denom ≡ *snd o Rep-Frac*

function *norm-frac* :: *int* ⇒ *int* ⇒ *int* × *int* **where**
[*simp del*]: *norm-frac a b* = (*if b* < *0 then norm-frac* (− *a*) (− *b*)
else if a = *0* ∨ *b* = *0 then* (*0*, *1*)
else let c = *int-gcd a b in* (*a div c*, *b div c*))
⟨*proof*⟩
termination ⟨*proof*⟩

definition *frac* :: *int* ⇒ *int* ⇒ *'a* **where**
frac a b ≡ *Abs-Frac* (*norm-frac a b*)

definition *plus-frac* :: *'a* ⇒ *'a* ⇒ *'a* **where**
[*nitpick-simp*]:
plus-frac q r = (*let d* = *int-lcm* (*denom q*) (*denom r*) *in*
frac (*num q* ∗ (*d div denom q*) + *num r* ∗ (*d div denom r*)) *d*)

definition *times-frac* :: *'a* ⇒ *'a* ⇒ *'a* **where**
[*nitpick-simp*]:
times-frac q r = *frac* (*num q* ∗ *num r*) (*denom q* ∗ *denom r*)

definition *uminus-frac* :: *'a* ⇒ *'a* **where**
uminus-frac q ≡ *Abs-Frac* (− *num q*, *denom q*)

definition *number-of-frac* :: *int* ⇒ *'a* **where**
number-of-frac n ≡ *Abs-Frac* (*n*, *1*)

definition *inverse-frac* :: *'a* ⇒ *'a* **where**
inverse-frac q ≡ *frac* (*denom q*) (*num q*)

definition *less-eq-frac* :: *'a* ⇒ *'a* ⇒ *bool* **where**
[*nitpick-simp*]:
less-eq-frac q r ⟷ *num* (*plus-frac q* (*uminus-frac r*)) ≤ *0*

definition *of-frac* :: *'a* ⇒ *'b*::{*inverse*,*ring-1*} **where**
of-frac q ≡ *of-int* (*num q*) / *of-int* (*denom q*)

⟨*ML*⟩

hide-const (**open**) *unknown is-unknown undefined-fast-The undefined-fast-Eps bisim*

bisim-iterator-max Quot safe-The safe-Eps FinFun FunBox PairBox Word refl′ wf′ wf-wfrec wf-wfrec′ wfrec′ card′ setsum′ fold-graph′ nat-gcd nat-lcm int-gcd int-lcm Frac Abs-Frac Rep-Frac zero-frac one-frac num denom norm-frac frac plus-frac times-frac uminus-frac number-of-frac inverse-frac less-eq-frac of-frac

hide-type (**open**) *bisim-iterator fin-fun fun-box pair-box unsigned-bit signed-bit word*

hide-fact (**open**) *If-def Ex1-def split-def rtrancl-def rtranclp-def tranclp-def refl′-def wf′-def wf-wfrec′-def wfrec′-def card′-def setsum′-def fold-graph′-def The-psimp Eps-psimp unit-case-def nat-case-def list-size-simp nat-gcd-def nat-lcm-def int-gcd-def int-lcm-def Frac-def zero-frac-def one-frac-def num-def denom-def norm-frac-def frac-def plus-frac-def times-frac-def uminus-frac-def number-of-frac-def inverse-frac-def less-eq-frac-def of-frac-def*

end

58 SMT: Bindings to Satisfiability Modulo Theories (SMT) solvers

theory *SMT*
imports *List*
uses
~~/*src*/*Tools*/*cache-io.ML*
(*Tools*/*SMT*/*smt-monomorph.ML*)
(*Tools*/*SMT*/*smt-normalize.ML*)
(*Tools*/*SMT*/*smt-translate.ML*)
(*Tools*/*SMT*/*smt-solver.ML*)
(*Tools*/*SMT*/*smtlib-interface.ML*)
(*Tools*/*SMT*/*z3-proof-parser.ML*)
(*Tools*/*SMT*/*z3-proof-tools.ML*)
(*Tools*/*SMT*/*z3-proof-literals.ML*)
(*Tools*/*SMT*/*z3-proof-reconstruction.ML*)
(*Tools*/*SMT*/*z3-model.ML*)
(*Tools*/*SMT*/*z3-interface.ML*)
(*Tools*/*SMT*/*z3-solver.ML*)
(*Tools*/*SMT*/*cvc3-solver.ML*)
(*Tools*/*SMT*/*yices-solver.ML*)
begin

58.1 Triggers for quantifier instantiation

Some SMT solvers support triggers for quantifier instantiation. Each trigger consists of one ore more patterns. A pattern may either be a list of positive subterms (each being tagged by "pat"), or a list of negative subterms (each being tagged by "nopat").

When an SMT solver finds a term matching a positive pattern (a pattern with positive subterms only), it instantiates the corresponding quantifier accordingly. Negative patterns inhibit quantifier instantiations. Each pattern should mention all preceding bound variables.

datatype *pattern* = *Pattern*

definition *pat* :: *'a* ⇒ *pattern* **where** *pat* - = *Pattern*
definition *nopat* :: *'a* ⇒ *pattern* **where** *nopat* - = *Pattern*

definition *trigger* :: *pattern list list* ⇒ *bool* ⇒ *bool*
where *trigger* - *P* = *P*

58.2 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.

definition *fun-app* **where** *fun-app f x* = *f x*

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

lemmas *array-rules* = *ext fun-upd-apply fun-upd-same fun-upd-other fun-upd-upd fun-app-def*

58.3 First-order logic

Some SMT solvers require a strict separation between formulas and terms. When translating higher-order into first-order problems, all uninterpreted constants (those not builtin in the target solver) are treated as function symbols in the first-order sense. Their occurrences as head symbols in atoms (i.e., as predicate symbols) is turned into terms by equating such atoms with *True* using the following term-level equation symbol.

definition *term-eq* :: *bool* ⇒ *bool* ⇒ *bool* **where** *term-eq x y* = (*x* = *y*)

58.4 Integer division and modulo for Z3

definition *z3div* :: *int* ⇒ *int* ⇒ *int* **where**
z3div k l = (*if* 0 ≤ *l then k div l else* −(*k div* (−*l*)))

definition *z3mod* :: *int* ⇒ *int* ⇒ *int* **where**
z3mod k l = (*if* 0 ≤ *l then k mod l else k mod* (−*l*))

lemma *div-by-z3div*: *k div l* = (
if k = 0 ∨ *l* = 0 *then* 0
else if (0 < *k* ∧ 0 < *l*) ∨ (*k* < 0 ∧ 0 < *l*) *then z3div k l*
else z3div (−*k*) (−*l*))

⟨*proof*⟩

lemma *mod-by-z3mod*: *k mod l* = (
if l = *0 then k*
else if k = *0 then 0*
else if (*0* < *k* ∧ *0* < *l*) ∨ (*k* < *0* ∧ *0* < *l*) *then z3mod k l*
else − *z3mod* (−*k*) (−*l*))
⟨*proof*⟩

58.5 Setup

⟨*ML*⟩

58.6 Configuration

The current configuration can be printed by the command *smt-status*, which shows the values of most options.

58.7 General configuration options

The option *smt-solver* can be used to change the target SMT solver. The possible values are *cvc3*, *yices*, and *z3*. It is advisable to locally install the selected solver, although this is not necessary for *cvc3* and *z3*, which can also be used over an Internet-based service.

When using local SMT solvers, the path to their binaries should be declared by setting the following environment variables: *CVC3-SOLVER*, *YICES-SOLVER*, and *Z3-SOLVER*.

declare [[*smt-solver* = *z3*]]

Since SMT solvers are potentially non-terminating, there is a timeout (given in seconds) to restrict their runtime. A value greater than 120 (seconds) is in most cases not advisable.

declare [[*smt-timeout* = *20*]]

58.8 Certificates

By setting the option *smt-certificates* to the name of a file, all following applications of an SMT solver a cached in that file. Any further application of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending *.certs* instead of *.thy*) as the certificates file.

declare [[*smt-certificates* =]]

The option *smt-fixed* controls whether only stored certificates are should be used or invocation of an SMT solver is allowed. When set to *true*, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to *false* and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

declare [[*smt-fixed* = *false*]]

58.9 Tracing

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option *smt-trace* should be set to *true*.

declare [[*smt-trace* = *false*]]

58.10 Z3-specific options

Z3 is the only SMT solver whose proofs are checked (or reconstructed) in Isabelle (all other solvers are implemented as oracles). Enabling or disabling proof reconstruction for Z3 is controlled by the option *z3-proofs*.

declare [[*z3-proofs* = *true*]]

From the set of assumptions given to Z3, those assumptions used in the proof are traced when the option *z3-trace-assms* is set to *true*.

declare [[*z3-trace-assms* = *false*]]

Z3 provides several commandline options to tweak its behaviour. They can be configured by writing them literally as value for the option *z3-options*.

declare [[*z3-options* =]]

58.11 Schematic rules for Z3 proof reconstruction

Several prof rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in *z3-rule* are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in *z3-simp* are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

lemmas [*z3-rule*] =
refl eq-commute conj-commute disj-commute simp-thms nnf-simps
ring-distribs field-simps times-divide-eq-right times-divide-eq-left
if-True if-False not-not

lemma [*z3-rule*]:
$(P \longrightarrow Q) = (Q \vee \neg P)$
$(\neg P \longrightarrow Q) = (P \vee Q)$

$(\neg P \longrightarrow Q) = (Q \vee P)$
⟨*proof*⟩

lemma [*z3-rule*]:
$((P = Q) \longrightarrow R) = (R \mid (Q = (\neg P)))$
⟨*proof*⟩

lemma [*z3-rule*]:
$((\neg P) = P) = \mathit{False}$
$(P = (\neg P)) = \mathit{False}$
$(P \neq Q) = (Q = (\neg P))$
$(P = Q) = ((\neg P \vee Q) \wedge (P \vee \neg Q))$
$(P \neq Q) = ((\neg P \vee \neg Q) \wedge (P \vee Q))$
⟨*proof*⟩

lemma [*z3-rule*]:
$(\mathit{if}\ P\ \mathit{then}\ P\ \mathit{else}\ \neg P) = \mathit{True}$
$(\mathit{if}\ \neg P\ \mathit{then}\ \neg P\ \mathit{else}\ P) = \mathit{True}$
$(\mathit{if}\ P\ \mathit{then}\ \mathit{True}\ \mathit{else}\ \mathit{False}) = P$
$(\mathit{if}\ P\ \mathit{then}\ \mathit{False}\ \mathit{else}\ \mathit{True}) = (\neg P)$
$(\mathit{if}\ \neg P\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ P\ \mathit{then}\ y\ \mathit{else}\ x)$
⟨*proof*⟩

lemma [*z3-rule*]:
$P = Q \vee P \vee Q$
$P = Q \vee \neg P \vee \neg Q$
$(\neg P) = Q \vee \neg P \vee Q$
$(\neg P) = Q \vee P \vee \neg Q$
$P = (\neg Q) \vee \neg P \vee Q$
$P = (\neg Q) \vee P \vee \neg Q$
$P \neq Q \vee P \vee \neg Q$
$P \neq Q \vee \neg P \vee Q$
$P \neq (\neg Q) \vee P \vee Q$
$(\neg P) \neq Q \vee P \vee Q$
$P \vee Q \vee P \neq (\neg Q)$
$P \vee Q \vee (\neg P) \neq Q$
$P \vee \neg Q \vee P \neq Q$
$\neg P \vee Q \vee P \neq Q$
⟨*proof*⟩

lemma [*z3-rule*]:
$0 + (x::\mathit{int}) = x$
$x + 0 = x$
$0 * x = 0$
$1 * x = x$
$x + y = y + x$
⟨*proof*⟩

hide-type (**open**) *pattern*
hide-const *Pattern term-eq*
hide-const (**open**) *trigger pat nopat fun-app z3div z3mod*

end

59 Main: Main HOL

theory *Main*
imports *Plain Predicate-Compile Nitpick SMT*
begin

Classical Higher-order Logic – only "Main", excluding real and complex numbers etc.

See further [1]

end

References

[1] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.