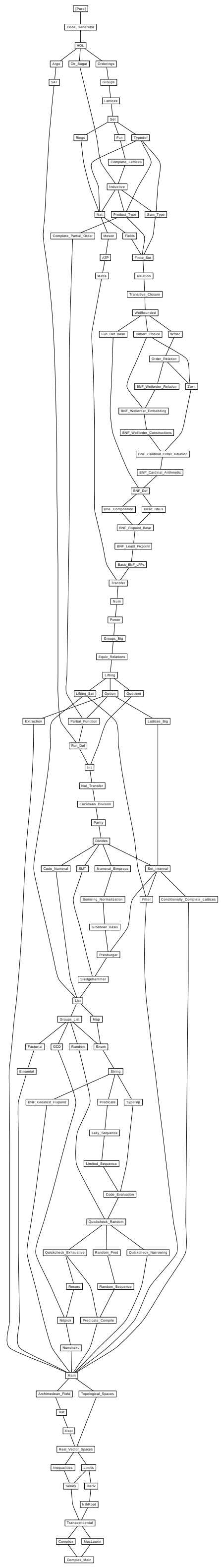
Isabelle/HOL — Higher-Order Logic

October 8, 2017

Contents


[Pure]
Code_Generator
HOL
Argo
Ctr_Sugar
Orderings
SAT
Groups
Lattices
Set
Rings
Fun
Typedef
Complete_Lattices
Inductive
Nat
Product_Type
Sum_Type
Complete_Partial_Order
Meson
Fields
ATP
Finite_Set
Metis
Relation
Transitive_Closure
Wellfounded
Fun_Def_Base
Hilbert_Choice
Wfrec
Order_Relation
BNF_Wellorder_Relation
Zorn
BNF_Wellorder_Embedding
BNF_Wellorder_Constructions
BNF_Cardinal_Order_Relation
BNF_Cardinal_Arithmetic
BNF_Def
BNF_Composition
Basic_BNFs
BNF_Fixpoint_Base
BNF_Least_Fixpoint
Basic_BNF_LFPs
Transfer
Num
Power
Groups_Big
Equiv_Relations
Lifting
Lifting_Set
Option
Quotient
Extraction
Partial_Function
Lattices_Big
Fun_Def
Int
Nat_Transfer
Euclidean_Division
Parity
Divides
Code_Numeral
SMT
Numeral_Simprocs
Set_Interval
Semiring_Normalization
Filter
Conditionally_Complete_Lattices
Groebner_Basis
Presburger
Sledgehammer
List
Groups_List
Map
Factorial
GCD
Random
Enum
Binomial
String
BNF_Greatest_Fixpoint
Predicate
Typerep
Lazy_Sequence
Limited_Sequence
Code_Evaluation
Quickcheck_Random
Quickcheck_Exhaustive
Random_Pred
Quickcheck_Narrowing
Record
Random_Sequence
Nitpick
Predicate_Compile
Nunchaku
Main
Archimedean_Field
Topological_Spaces
Rat
Real
Real_Vector_Spaces
Inequalities
Limits
Series
Deriv
NthRoot
Transcendental
Complex
MacLaurin
Complex_Main

1 Loading the code generator and related modules

theory *Code-Generator*
imports *Pure*
keywords
  print-codeproc code-thms code-deps :: *diag* **and**
  export-code code-identifier code-printing code-reserved
    code-monad code-reflect :: *thy-decl* **and**
  checking **and**
  datatypes functions module-name file
    constant type-constructor type-class class-relation class-instance code-module
    :: *quasi-command*
begin

⟨*ML*⟩

code-datatype $TYPE('a::\{\})$

definition *holds* :: *prop* **where**
  $holds \equiv ((\lambda x::prop.\ x) \equiv (\lambda x.\ x))$

lemma *holds*: *PROP holds*
  ⟨*proof*⟩

code-datatype *holds*

lemma *implies-code* [*code*]:
  $(PROP\ holds \Longrightarrow PROP\ P) \equiv PROP\ P$
  $(PROP\ P \Longrightarrow PROP\ holds) \equiv PROP\ holds$
⟨*proof*⟩

⟨*ML*⟩

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

end

2 The basis of Higher-Order Logic

theory *HOL*
imports *Pure* *~~/src/Tools/Code-Generator*
keywords
  try solve-direct quickcheck print-coercions print-claset
    print-induct-rules :: *diag* **and**
  quickcheck-params :: *thy-decl*
begin

⟨*ML*⟩

2.1 Primitive logic

2.1.1 Core syntax

⟨*ML*⟩
default-sort *type*
⟨*ML*⟩

axiomatization where *fun-arity*: $OFCLASS('a \Rightarrow 'b, \textit{type-class})$
instance *fun* :: (*type*, *type*) *type* ⟨*proof*⟩

axiomatization where *itself-arity*: $OFCLASS('a\ \textit{itself}, \textit{type-class})$
instance *itself* :: (*type*) *type* ⟨*proof*⟩

typedecl *bool*

judgment *Trueprop* :: $bool \Rightarrow prop$ ((-) *5*)

axiomatization *implies* :: $[bool, bool] \Rightarrow bool$ (**infixr** $\longrightarrow$ *25*)
  and *eq* :: $['a, 'a] \Rightarrow bool$ (**infixl** = *50*)
  and *The* :: $('a \Rightarrow bool) \Rightarrow 'a$

2.1.2 Defined connectives and quantifiers

definition *True* :: *bool*
  where $True \equiv ((\lambda x::bool.\ x) = (\lambda x.\ x))$

definition *All* :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** $\forall$ *10*)
  where $All\ P \equiv (P = (\lambda x.\ True))$

definition *Ex* :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** $\exists$ *10*)
  where $Ex\ P \equiv \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q$

definition *False* :: *bool*
  where $False \equiv (\forall P.\ P)$

definition *Not* :: $bool \Rightarrow bool$ ($\neg$ - [*40*] *40*)
  where *not-def*: $\neg\ P \equiv P \longrightarrow False$

definition *conj* :: $[bool, bool] \Rightarrow bool$ (**infixr** $\wedge$ *35*)
  where *and-def*: $P \wedge Q \equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$

definition *disj* :: $[bool, bool] \Rightarrow bool$ (**infixr** $\vee$ *30*)
  where *or-def*: $P \vee Q \equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

definition *Ex1* :: $('a \Rightarrow bool) \Rightarrow bool$
  where $Ex1\ P \equiv \exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow y = x)$

2.1.3 Additional concrete syntax

syntax ($ASCII$)
  $\mathit{\text{-}Ex1} :: pttrn \Rightarrow bool \Rightarrow bool$ ((3EX! -./ -) [0, 10] 10)
syntax ($input$)
  $\mathit{\text{-}Ex1} :: pttrn \Rightarrow bool \Rightarrow bool$ ((3?! -./ -) [0, 10] 10)
syntax $\mathit{\text{-}Ex1} :: pttrn \Rightarrow bool \Rightarrow bool$ ((3∃!-./ -) [0, 10] 10)
translations $\exists !x.\ P \rightleftharpoons CONST\ Ex1\ (\lambda x.\ P)$

$\langle ML\rangle$

syntax
  $\mathit{\text{-}Not\text{-}Ex} :: idts \Rightarrow bool \Rightarrow bool$ ((3∄-./ -) [0, 10] 10)
  $\mathit{\text{-}Not\text{-}Ex1} :: pttrn \Rightarrow bool \Rightarrow bool$ ((3∄!-./ -) [0, 10] 10)
translations
  $\nexists x.\ P \rightleftharpoons \neg\ (\exists x.\ P)$
  $\nexists !x.\ P \rightleftharpoons \neg\ (\exists !x.\ P)$

abbreviation $not\text{-}equal :: ['a,\ 'a] \Rightarrow bool$ (**infixl** $\neq$ 50)
  where $x \neq y \equiv \neg\ (x = y)$

notation (**output**)
  eq (**infix** = 50) **and**
  $not\text{-}equal$ (**infix** $\neq$ 50)

notation ($ASCII$ **output**)
  $not\text{-}equal$ (**infix** ~= 50)

notation ($ASCII$)
  Not (~ - [40] 40) **and**
  $conj$ (**infixr** & 35) **and**
  $disj$ (**infixr** | 30) **and**
  $implies$ (**infixr** −−> 25) **and**
  $not\text{-}equal$ (**infixl** ~= 50)

abbreviation (iff)
  $iff :: [bool,\ bool] \Rightarrow bool$ (**infixr** $\longleftrightarrow$ 25)
  where $A \longleftrightarrow B \equiv A = B$

syntax $\mathit{\text{-}The} :: [pttrn,\ bool] \Rightarrow 'a$ ((3THE -./ -) [0, 10] 10)
translations $THE\ x.\ P \rightleftharpoons CONST\ The\ (\lambda x.\ P)$
$\langle ML\rangle$

nonterminal $letbinds$ **and** $letbind$
syntax
  $\mathit{\text{-}bind} :: [pttrn,\ 'a] \Rightarrow letbind$ ((2- =/ -) 10)
  $:: letbind \Rightarrow letbinds$ (-)
  $\mathit{\text{-}binds} :: [letbind,\ letbinds] \Rightarrow letbinds$ (-;/ -)

$-Let$ :: $[letbinds,\ 'a] \Rightarrow 'a$ $((let\ (-)/\ in\ (-))\ [0,\ 10]\ 10)$

nonterminal *case-syn* **and** *cases-syn*
syntax
$-case\text{-}syntax :: ['a,\ cases\text{-}syn] \Rightarrow 'b\ \ ((case\ -\ of/\ -)\ 10)$
$-case1 :: ['a,\ 'b] \Rightarrow case\text{-}syn\ \ ((2\text{-}\ \Rightarrow/\ -)\ 10)$
$:: case\text{-}syn \Rightarrow cases\text{-}syn\ \ (-)$
$-case2 :: [case\text{-}syn,\ cases\text{-}syn] \Rightarrow cases\text{-}syn\ \ (-/\ |\ -)$
syntax (*ASCII*)
$-case1 :: ['a,\ 'b] \Rightarrow case\text{-}syn\ \ ((2\text{-}\ =>/\ -)\ 10)$

notation (*ASCII*)
All (**binder** *ALL* *10*) **and**
Ex (**binder** *EX* *10*)

notation (*input*)
All (**binder** ! *10*) **and**
Ex (**binder** ? *10*)

2.1.4 Axioms and basic definitions

axiomatization where
refl: $t = (t::'a)$ **and**
subst: $s = t \Longrightarrow P\ s \Longrightarrow P\ t$ **and**
ext: $(\bigwedge x::'a.\ (f\ x\ ::'b) = g\ x) \Longrightarrow (\lambda x.\ f\ x) = (\lambda x.\ g\ x)$
— 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 **and**

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

axiomatization where
impI: $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$ **and**
mp: $[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q$ **and**

iff: $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)$ **and**
True-or-False: $(P = True) \vee (P = False)$

definition *If* :: $bool \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ ((if\ (-)/\ then\ (-)/\ else\ (-))\ [0,\ 0,\ 10]\ 10)$
where $If\ P\ x\ y \equiv (THE\ z::'a.\ (P = True \longrightarrow z = x) \wedge (P = False \longrightarrow z = y))$

definition *Let* :: $'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$
where $Let\ s\ f \equiv f\ s$

translations
$-Let\ (-binds\ b\ bs)\ e\ \rightleftharpoons -Let\ b\ (-Let\ bs\ e)$
$let\ x = a\ in\ e\ \rightleftharpoons CONST\ Let\ a\ (\lambda x.\ e)$

axiomatization *undefined* :: $'a$

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

2.2 Fundamental rules

2.2.1 Equality

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

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

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

lemma *trans-sym* [*Pure.elim?*]: $r = s \Longrightarrow t = s \Longrightarrow r = t$
⟨*proof*⟩

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

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

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

For calculational reasoning:

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

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

2.2.2 Congruence rules for application

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

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

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

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

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

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

⟨*ML*⟩

2.2.3 Equality of booleans – iff

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

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

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

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

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

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

2.2.4 True

lemma *TrueI*: *True*
⟨*proof*⟩

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

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

2.2.5 Universal quantifier

lemma *allI*:
assumes $\bigwedge x::'a.\ P\ x$
shows $\forall x.\ P\ x$
⟨*proof*⟩

lemma *spec*: $\forall x::'a.\ P\ x \Longrightarrow P\ x$
⟨*proof*⟩

lemma *allE*:

assumes *major*: $\forall x.\ P\ x$
and *minor*: $P\ x \Longrightarrow R$
shows R
$\langle proof \rangle$

lemma *all-dupE*:
assumes *major*: $\forall x.\ P\ x$
and *minor*: $[\![P\ x;\ \forall x.\ P\ x]\!] \Longrightarrow R$
shows R
$\langle proof \rangle$

2.2.6 False

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

lemma *FalseE*: $False \Longrightarrow P$
$\langle proof \rangle$

lemma *False-neq-True*: $False = True \Longrightarrow P$
$\langle proof \rangle$

2.2.7 Negation

lemma *notI*:
assumes $P \Longrightarrow False$
shows $\neg\ P$
$\langle proof \rangle$

lemma *False-not-True*: $False \neq True$
$\langle proof \rangle$

lemma *True-not-False*: $True \neq False$
$\langle proof \rangle$

lemma *notE*: $[\![\neg\ P;\ P]\!] \Longrightarrow R$
$\langle proof \rangle$

lemma *notI2*: $(P \Longrightarrow \neg\ Pa) \Longrightarrow (P \Longrightarrow Pa) \Longrightarrow \neg\ P$
$\langle proof \rangle$

2.2.8 Implication

lemma *impE*:
assumes $P \longrightarrow Q\ P\ Q \Longrightarrow R$
shows R
$\langle proof \rangle$

Reduces Q to $P \longrightarrow Q$, allowing substitution in P.

lemma *rev-mp*: $[\![P;\ P \longrightarrow Q]\!] \Longrightarrow Q$

⟨*proof*⟩

lemma *contrapos-nn*:
assumes *major*: $\neg\ Q$
and *minor*: $P \Longrightarrow Q$
shows $\neg\ P$
⟨*proof*⟩

Not used at all, but we already have the other 3 combinations.

lemma *contrapos-pn*:
assumes *major*: Q
and *minor*: $P \Longrightarrow \neg\ Q$
shows $\neg\ P$
⟨*proof*⟩

lemma *not-sym*: $t \neq s \Longrightarrow s \neq t$
⟨*proof*⟩

lemma *eq-neq-eq-imp-neq*: $[\![x = a;\ a \neq b;\ b = y]\!] \Longrightarrow x \neq y$
⟨*proof*⟩

2.2.9 Existential quantifier

lemma *exI*: $P\ x \Longrightarrow \exists\, x{::}'a.\ P\ x$
⟨*proof*⟩

lemma *exE*:
assumes *major*: $\exists\, x{::}'a.\ P\ x$
and *minor*: $\bigwedge x.\ P\ x \Longrightarrow Q$
shows Q
⟨*proof*⟩

2.2.10 Conjunction

lemma *conjI*: $[\![P;\ Q]\!] \Longrightarrow P \wedge Q$
⟨*proof*⟩

lemma *conjunct1*: $[\![P \wedge Q]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *conjunct2*: $[\![P \wedge Q]\!] \Longrightarrow Q$
⟨*proof*⟩

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

lemma *context-conjI*:

assumes P $P \Longrightarrow Q$
shows $P \wedge Q$
⟨*proof*⟩

2.2.11 Disjunction

lemma *disjI1*: $P \Longrightarrow P \vee Q$
⟨*proof*⟩

lemma *disjI2*: $Q \Longrightarrow P \vee Q$
⟨*proof*⟩

lemma *disjE*:
assumes *major*: $P \vee Q$
and *minorP*: $P \Longrightarrow R$
and *minorQ*: $Q \Longrightarrow R$
shows R
⟨*proof*⟩

2.2.12 Classical logic

lemma *classical*:
assumes *prem*: $\neg P \Longrightarrow P$
shows P
⟨*proof*⟩

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

notE with premises exchanged; it discharges $\neg R$ so that it can be used to make elimination rules.

lemma *rev-notE*:
assumes *premp*: P
and *premnot*: $\neg R \Longrightarrow \neg P$
shows R
⟨*proof*⟩

Double negation law.

lemma *notnotD*: $\neg\neg P \Longrightarrow P$
⟨*proof*⟩

lemma *contrapos-pp*:
assumes *p1*: Q
and *p2*: $\neg P \Longrightarrow \neg Q$
shows P
⟨*proof*⟩

2.2.13 Unique existence

lemma *ex1I*:

assumes $P\ a\ \bigwedge x.\ P\ x \Longrightarrow x = a$
shows $\exists !x.\ P\ x$
⟨*proof*⟩

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

lemma *ex-ex1I*:
assumes *ex-prem*: $\exists\, x.\ P\ x$
and *eq*: $\bigwedge x\ y.\ [\![P\ x;\ P\ y]\!] \Longrightarrow x = y$
shows $\exists !x.\ P\ x$
⟨*proof*⟩

lemma *ex1E*:
assumes *major*: $\exists !x.\ P\ x$
and *minor*: $\bigwedge x.\ [\![P\ x;\ \forall\, y.\ P\ y \longrightarrow y = x]\!] \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *ex1-implies-ex*: $\exists !x.\ P\ x \Longrightarrow \exists\, x.\ P\ x$
⟨*proof*⟩

2.2.14 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes $\neg\ Q \Longrightarrow P$
shows $P \vee Q$
⟨*proof*⟩

lemma *excluded-middle*: $\neg\ P \vee P$
⟨*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*: $P \Longrightarrow Q$
and *prem2*: $\neg\ P \Longrightarrow Q$
shows Q
⟨*proof*⟩

Classical implies ($\longrightarrow$) elimination.

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

This version of $\longrightarrow$ elimination works on Q before P. It works best for those cases in which P holds "almost everywhere". Can't install as default: would break old proofs.

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

Classical $\longleftrightarrow$ elimination.

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

lemma *exCI*:
assumes $\forall x.\ \neg\ P\ x \Longrightarrow P\ a$
shows $\exists x.\ P\ x$
⟨*proof*⟩

2.2.15 Intuitionistic Reasoning

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

lemma *allE′*:
assumes *1*: $\forall x.\ P\ x$
and *2*: $P\ x \Longrightarrow \forall x.\ P\ x \Longrightarrow Q$
shows Q
⟨*proof*⟩

lemma *notE′*:
assumes *1*: $\neg\ P$
and *2*: $\neg\ P \Longrightarrow P$
shows R
⟨*proof*⟩

lemma *TrueE*: *True* $\Longrightarrow P \Longrightarrow P$ ⟨*proof*⟩
lemma *notFalseE*: $\neg$ *False* $\Longrightarrow P \Longrightarrow P$ ⟨*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*

2.2.16 Atomizing meta-level connectives

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

lemma *atomize-all* [*atomize*]: $(\bigwedge x.\ P\ x) \equiv \mathit{Trueprop}\ (\forall x.\ P\ x)$
⟨*proof*⟩

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

lemma *atomize-not*: $(A \Longrightarrow \mathit{False}) \equiv \mathit{Trueprop}\ (\neg\ A)$
⟨*proof*⟩

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

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

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

2.2.17 Atomizing elimination rules

lemma *atomize-exL*[*atomize-elim*]: $(\bigwedge x.\ P\ x \Longrightarrow Q) \equiv ((\exists x.\ P\ x) \Longrightarrow Q)$
⟨*proof*⟩

lemma *atomize-conjL*[*atomize-elim*]: $(A \Longrightarrow B \Longrightarrow C) \equiv (A \wedge B \Longrightarrow C)$
⟨*proof*⟩

lemma *atomize-disjL*[*atomize-elim*]: $((A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C) \equiv ((A \vee B \Longrightarrow C) \Longrightarrow C)$
⟨*proof*⟩

lemma *atomize-elimL*[*atomize-elim*]: $(\bigwedge B.\ (A \Longrightarrow B) \Longrightarrow B) \equiv \mathit{Trueprop}\ A$ ⟨*proof*⟩

2.3 Package setup

⟨*ML*⟩

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.

named-theorems *no-atp theorems that should be filtered out by Sledgehammer*

2.3.2 Classical Reasoner setup

lemma *imp-elim*: $P \longrightarrow Q \Longrightarrow (\neg\ R \Longrightarrow P) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *swap*: $\neg\ P \Longrightarrow (\neg\ R \Longrightarrow P) \Longrightarrow R$
⟨*proof*⟩

lemma *thin-refl*: $[\![x = x;\ PROP\ W]\!] \Longrightarrow PROP\ W$ ⟨*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 *exI* [*intro*]

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

⟨*ML*⟩

lemma *contrapos-np*: $\neg\ Q \Longrightarrow (\neg\ P \Longrightarrow Q) \Longrightarrow P$
⟨*proof*⟩

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

declare *ext* [*intro*]

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

Better than *ex1E* for classical reasoner: needs no quantifier duplication!

lemma *alt-ex1E* [*elim*!]:
assumes *major*: $\exists!x.\ P\ x$
and *prem*: $\bigwedge x.\ [\![P\ x;\ \forall\ y\ y'.\ P\ y \wedge P\ y' \longrightarrow y = y']\!] \Longrightarrow R$

shows R
⟨*proof*⟩

⟨*ML*⟩

2.3.3 THE: definite description operator

lemma *the-equality* [*intro*]:
assumes $P\ a$
and $\bigwedge x.\ P\ x \Longrightarrow x = a$
shows $(THE\ x.\ P\ x) = a$
⟨*proof*⟩

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

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

Easier to apply than *theI*: only one occurrence of P.

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

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

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

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

2.3.4 Simplifier

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

lemma *simp-thms*:
shows *not-not*: $(\neg\ \neg\ P) = P$
and *Not-eq-iff*: $((\neg\ P) = (\neg\ Q)) = (P = Q)$
and
$(P \neq Q) = (P = (\neg\ Q))$
$(P \vee \neg P) = True \quad (\neg\ P \vee P) = True$
$(x = x) = True$

and *not-True-eq-False* [*code*]: $(\neg\ True) = False$
and *not-False-eq-True* [*code*]: $(\neg\ False) = True$
and
$(\neg\ P) \neq P \quad P \neq (\neg\ P)$
$(True = P) = P$
and *eq-True*: $(P = True) = P$
and $(False = P) = (\neg\ P)$
and *eq-False*: $(P = False) = (\neg\ P)$
and
$(True \longrightarrow P) = P \quad (False \longrightarrow P) = True$
$(P \longrightarrow True) = True \quad (P \longrightarrow P) = True$
$(P \longrightarrow False) = (\neg\ P) \quad (P \longrightarrow \neg\ P) = (\neg\ P)$
$(P \wedge True) = P \quad (True \wedge P) = P$
$(P \wedge False) = False \quad (False \wedge P) = False$
$(P \wedge P) = P \quad (P \wedge (P \wedge Q)) = (P \wedge Q)$
$(P \wedge \neg\ P) = False \quad (\neg\ P \wedge P) = False$
$(P \vee True) = True \quad (True \vee P) = True$
$(P \vee False) = P \quad (False \vee P) = P$
$(P \vee P) = P \quad (P \vee (P \vee Q)) = (P \vee Q)$ **and**
$(\forall x.\ P) = P \quad (\exists x.\ P) = P \quad \exists x.\ x = t \quad \exists x.\ t = x$
and
$\bigwedge P.\ (\exists x.\ x = t \wedge P\ x) = P\ t$
$\bigwedge P.\ (\exists x.\ t = x \wedge P\ x) = P\ t$
$\bigwedge P.\ (\forall x.\ x = t \longrightarrow P\ x) = P\ t$
$\bigwedge P.\ (\forall x.\ t = x \longrightarrow P\ x) = P\ t$
$(\forall x.\ x \neq t) = False \quad (\forall x.\ t \neq x) = False$
⟨*proof*⟩

lemma *disj-absorb*: $A \vee A \longleftrightarrow A$
⟨*proof*⟩

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

lemma *conj-absorb*: $A \wedge A \longleftrightarrow A$
⟨*proof*⟩

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

lemma *eq-ac*:
shows *eq-commute*: $a = b \longleftrightarrow b = a$
and *iff-left-commute*: $(P \longleftrightarrow (Q \longleftrightarrow R)) \longleftrightarrow (Q \longleftrightarrow (P \longleftrightarrow R))$
and *iff-assoc*: $((P \longleftrightarrow Q) \longleftrightarrow R) \longleftrightarrow (P \longleftrightarrow (Q \longleftrightarrow R))$
⟨*proof*⟩

lemma *neq-commute*: $a \neq b \longleftrightarrow b \neq a$ ⟨*proof*⟩

lemma *conj-comms*:

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

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

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

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

lemma *conj-disj-distribL*: $P \wedge (Q \vee R) \longleftrightarrow P \wedge Q \vee P \wedge R$ ⟨*proof*⟩
lemma *conj-disj-distribR*: $(P \vee Q) \wedge R \longleftrightarrow P \wedge R \vee Q \wedge R$ ⟨*proof*⟩

lemma *disj-conj-distribL*: $P \vee (Q \wedge R) \longleftrightarrow (P \vee Q) \wedge (P \vee R)$ ⟨*proof*⟩
lemma *disj-conj-distribR*: $(P \wedge Q) \vee R \longleftrightarrow (P \vee R) \wedge (Q \vee R)$ ⟨*proof*⟩

lemma *imp-conjR*: $(P \longrightarrow (Q \wedge R)) = ((P \longrightarrow Q) \wedge (P \longrightarrow R))$ ⟨*proof*⟩
lemma *imp-conjL*: $((P \wedge Q) \longrightarrow R) = (P \longrightarrow (Q \longrightarrow R))$ ⟨*proof*⟩
lemma *imp-disjL*: $((P \vee Q) \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ ⟨*proof*⟩

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

lemma *imp-disj-not1*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg Q \longrightarrow P \longrightarrow R)$ ⟨*proof*⟩
lemma *imp-disj-not2*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg R \longrightarrow P \longrightarrow Q)$ ⟨*proof*⟩

lemma *imp-disj1*: $((P \longrightarrow Q) \vee R) \longleftrightarrow (P \longrightarrow Q \vee R)$ ⟨*proof*⟩
lemma *imp-disj2*: $(Q \vee (P \longrightarrow R)) \longleftrightarrow (P \longrightarrow Q \vee R)$ ⟨*proof*⟩

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

lemma *de-Morgan-disj*: $\neg (P \vee Q) \longleftrightarrow \neg P \wedge \neg Q$ ⟨*proof*⟩
lemma *de-Morgan-conj*: $\neg (P \wedge Q) \longleftrightarrow \neg P \vee \neg Q$ ⟨*proof*⟩
lemma *not-imp*: $\neg (P \longrightarrow Q) \longleftrightarrow P \wedge \neg Q$ ⟨*proof*⟩
lemma *not-iff*: $P \neq Q \longleftrightarrow (P \longleftrightarrow \neg Q)$ ⟨*proof*⟩
lemma *disj-not1*: $\neg P \vee Q \longleftrightarrow (P \longrightarrow Q)$ ⟨*proof*⟩
lemma *disj-not2*: $P \vee \neg Q \longleftrightarrow (Q \longrightarrow P)$ ⟨*proof*⟩
lemma *imp-conv-disj*: $(P \longrightarrow Q) \longleftrightarrow (\neg P) \vee Q$ ⟨*proof*⟩
lemma *disj-imp*: $P \vee Q \longleftrightarrow \neg P \longrightarrow Q$ ⟨*proof*⟩

lemma *iff-conv-conj-imp*: $(P \longleftrightarrow Q) \longleftrightarrow (P \longrightarrow Q) \wedge (Q \longrightarrow P)$ ⟨*proof*⟩

lemma *cases-simp*: $(P \longrightarrow Q) \wedge (\neg P \longrightarrow Q) \longleftrightarrow Q$
— Avoids duplication of subgoals after *if-split*, when the true and false
— cases boil down to the same thing.

$\langle proof \rangle$

lemma *not-all*: $\neg (\forall x.\ P\ x) \longleftrightarrow (\exists x.\ \neg P\ x)$ $\langle proof \rangle$
lemma *imp-all*: $((\forall x.\ P\ x) \longrightarrow Q) \longleftrightarrow (\exists x.\ P\ x \longrightarrow Q)$ $\langle proof \rangle$
lemma *not-ex*: $\neg (\exists x.\ P\ x) \longleftrightarrow (\forall x.\ \neg P\ x)$ $\langle proof \rangle$
lemma *imp-ex*: $((\exists x.\ P\ x) \longrightarrow Q) \longleftrightarrow (\forall x.\ P\ x \longrightarrow Q)$ $\langle proof \rangle$
lemma *all-not-ex*: $(\forall x.\ P\ x) \longleftrightarrow \neg (\exists x.\ \neg P\ x)$ $\langle proof \rangle$

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

lemma *ex-disj-distrib*: $(\exists x.\ P\ x \lor Q\ x) \longleftrightarrow (\exists x.\ P\ x) \lor (\exists x.\ Q\ x)$ $\langle proof \rangle$
lemma *all-conj-distrib*: $(\forall x.\ P\ x \land Q\ x) \longleftrightarrow (\forall x.\ P\ x) \land (\forall x.\ Q\ x)$ $\langle proof \rangle$

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

lemma *conj-cong*: $P = P' \Longrightarrow (P' \Longrightarrow Q = Q') \Longrightarrow (P \land Q) = (P' \land Q')$
$\langle proof \rangle$

lemma *rev-conj-cong*: $Q = Q' \Longrightarrow (Q' \Longrightarrow P = P') \Longrightarrow (P \land Q) = (P' \land Q')$
$\langle proof \rangle$

The | congruence rule: not included by default!

lemma *disj-cong*: $P = P' \Longrightarrow (\neg P' \Longrightarrow Q = Q') \Longrightarrow (P \lor Q) = (P' \lor Q')$
$\langle proof \rangle$

if-then-else rules

lemma *if-True* [*code*]: $(\mathit{if}\ \mathit{True}\ \mathit{then}\ x\ \mathit{else}\ y) = x$
$\langle proof \rangle$

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

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

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

lemma *if-split*: $P\ (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) = ((Q \longrightarrow P\ x) \land (\neg Q \longrightarrow P\ y))$
$\langle proof \rangle$

lemma *if-split-asm*: $P\ (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) = (\neg ((Q \land \neg P\ x) \lor (\neg Q \land \neg P\ y)))$
$\langle proof \rangle$

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

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

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

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

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

lemma *Eq-TrueI*: $P \Longrightarrow P \equiv$ *True* ⟨*proof*⟩
lemma *Eq-FalseI*: $\neg P \Longrightarrow P \equiv$ *False* ⟨*proof*⟩

let rules for simproc

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

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

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

definition *simp-implies* :: *prop* ⇒ *prop* ⇒ *prop* (**infixr** *=simp=> 1*)
where *simp-implies* ≡ *op* ⟹

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

lemma *simp-impliesE*:
assumes *PQ*: *PROP P =simp=> PROP Q*
and *P*: *PROP P*
and *QR*: *PROP Q* ⟹ *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 $(y = x) \equiv \mathit{False}$ from premise $\neg\ (x = y)$:

⟨*ML*⟩

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

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

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

lemma *implies-False-swap*:
$\mathit{NO\text{-}MATCH}\ (\mathit{Trueprop}\ \mathit{False})\ P \Longrightarrow$
$(\mathit{False} \Longrightarrow \mathit{PROP}\ P \Longrightarrow \mathit{PROP}\ Q) \equiv (\mathit{PROP}\ P \Longrightarrow \mathit{False} \Longrightarrow \mathit{PROP}\ Q)$
⟨*proof*⟩

lemma *ex-simps*:
$\bigwedge P\ Q.\ (\exists x.\ P\ x \wedge Q) = ((\exists x.\ P\ x) \wedge Q)$
$\bigwedge P\ Q.\ (\exists x.\ P \wedge Q\ x) = (P \wedge (\exists x.\ Q\ x))$
$\bigwedge P\ Q.\ (\exists x.\ P\ x \vee Q) = ((\exists x.\ P\ x) \vee Q)$
$\bigwedge P\ Q.\ (\exists x.\ P \vee Q\ x) = (P \vee (\exists x.\ Q\ x))$
$\bigwedge P\ Q.\ (\exists x.\ P\ x \longrightarrow Q) = ((\forall x.\ P\ x) \longrightarrow Q)$
$\bigwedge P\ Q.\ (\exists x.\ P \longrightarrow Q\ x) = (P \longrightarrow (\exists x.\ Q\ x))$
— Miniscoping: pushing in existential quantifiers.
⟨*proof*⟩

lemma *all-simps*:

$\bigwedge P\ Q.\ (\forall x.\ P\ x \wedge Q) = ((\forall x.\ P\ x) \wedge Q)$
$\bigwedge P\ Q.\ (\forall x.\ P \wedge Q\ x) = (P \wedge (\forall x.\ Q\ x))$
$\bigwedge P\ Q.\ (\forall x.\ P\ x \vee Q) = ((\forall x.\ P\ x) \vee Q)$
$\bigwedge P\ Q.\ (\forall x.\ P \vee Q\ x) = (P \vee (\forall x.\ Q\ x))$
$\bigwedge P\ Q.\ (\forall x.\ P\ x \longrightarrow Q) = ((\exists x.\ P\ x) \longrightarrow Q)$
$\bigwedge P\ Q.\ (\forall x.\ P \longrightarrow Q\ x) = (P \longrightarrow (\forall x.\ Q\ x))$
— Miniscoping: pushing in universal quantifiers.
⟨*proof*⟩

lemmas [*simp*] =
triv-forall-equality — prunes params
True-implies-equals implies-True-equals — prune *True* in asms
False-implies-equals — prune *False* in asms
if-True
if-False
if-cancel
if-eq-cancel
imp-disjL — In general it seems wrong to add distributive laws by default: they might cause exponential blow-up. But *imp-disjL* has been in for a while and cannot be removed without affecting existing proofs. Moreover, rewriting by $(P \vee Q \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ might be justified on the grounds that it allows simplification of R in the two cases.
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*] = *if-split*

⟨*ML*⟩

Simplifies x assuming c and y assuming $\neg\ c$.

lemma *if-cong*:
assumes $b = c$
and $c \Longrightarrow x = u$
and $\neg\ c \Longrightarrow y = v$

shows $(\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ c\ \mathit{then}\ u\ \mathit{else}\ v)$
$\langle \mathit{proof} \rangle$

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

lemma *if-weak-cong* [*cong*]:
assumes $b = c$
shows $(\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ c\ \mathit{then}\ x\ \mathit{else}\ y)$
$\langle \mathit{proof} \rangle$

Prevents simplification of t: much faster

lemma *let-weak-cong*:
assumes $a = b$
shows $(\mathit{let}\ x = a\ \mathit{in}\ t\ x) = (\mathit{let}\ x = b\ \mathit{in}\ t\ x)$
$\langle \mathit{proof} \rangle$

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

lemma *eq-cong2*:
assumes $u = u'$
shows $(t \equiv u) \equiv (t \equiv u')$
$\langle \mathit{proof} \rangle$

lemma *if-distrib*: $f\ (\mathit{if}\ c\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ c\ \mathit{then}\ f\ x\ \mathit{else}\ f\ y)$
$\langle \mathit{proof} \rangle$

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

lemma *fun-eq-iff*: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$
$\langle \mathit{proof} \rangle$

2.3.5 Generic cases and induction

Rule projections:

$\langle \mathit{ML} \rangle$

context
begin

qualified definition $\mathit{induct\text{-}forall}\ P \equiv \forall x.\ P\ x$
qualified definition $\mathit{induct\text{-}implies}\ A\ B \equiv A \longrightarrow B$
qualified definition $\mathit{induct\text{-}equal}\ x\ y \equiv x = y$
qualified definition $\mathit{induct\text{-}conj}\ A\ B \equiv A \wedge B$
qualified definition $\mathit{induct\text{-}true} \equiv \mathit{True}$
qualified definition $\mathit{induct\text{-}false} \equiv \mathit{False}$

lemma *induct-forall-eq*: $(\bigwedge x.\ P\ x) \equiv \mathit{Trueprop}\ (\mathit{induct\text{-}forall}\ (\lambda x.\ P\ x))$
$\langle \mathit{proof} \rangle$

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

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

lemma *induct-conj-eq*: $(A$ &&& $B) \equiv$ *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*] = *induct-atomize′*
lemmas *induct-rulify* [*symmetric*] = *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 $B \Longrightarrow$ *PROP* $C) \equiv (A \Longrightarrow B \Longrightarrow$ *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*]: ($\bigwedge x$. *induct-equal* x $t \Longrightarrow$ *PROP* P $x) \equiv$ *PROP* P t
⟨*proof*⟩

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

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

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

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

lemma [*induct-simp*]: $(\bigwedge x::'a::\{\}.\ \mathit{induct\text{-}true}) \equiv \mathit{Trueprop}\ \mathit{induct\text{-}true}$
⟨*proof*⟩

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

lemma [*induct-simp*]: $x = x \longleftrightarrow \mathit{True}$
⟨*proof*⟩

end

⟨*ML*⟩

2.3.6 Coherent logic

⟨*ML*⟩

2.3.7 Reorienting equalities

⟨*ML*⟩

2.4 Other simple lemmas and lemma duplicates

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

lemma *choice-eq*: $(\forall x.\ \exists!y.\ P\ x\ y) = (\exists!f.\ \forall x.\ P\ x\ (f\ x))$
⟨*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*⟩

3 *NO-MATCH* simproc

The simplification procedure can be used to avoid simplification of terms of a certain form.

definition *NO-MATCH* :: $'a \Rightarrow 'b \Rightarrow bool$
  where *NO-MATCH pat val* $\equiv$ *True*

lemma *NO-MATCH-cong*[*cong*]: *NO-MATCH pat val* = *NO-MATCH pat val*
  ⟨*proof*⟩

declare [[*coercion-args NO-MATCH* − −]]

⟨*ML*⟩

This setup ensures that a rewrite rule of the form *NO-MATCH pat val* $\Longrightarrow$ t is only applied, if the pattern *pat* does not match the value *val*.

Tagging a premise of a simp rule with ASSUMPTION forces the simplifier not to simplify the argument and to solve it by an assumption.

definition *ASSUMPTION* :: $bool \Rightarrow bool$
  where *ASSUMPTION* $A \equiv A$

lemma *ASSUMPTION-cong*[*cong*]: *ASSUMPTION* A = *ASSUMPTION* A
  ⟨*proof*⟩

lemma *ASSUMPTION-I*: $A \Longrightarrow$ *ASSUMPTION* A
  ⟨*proof*⟩

lemma *ASSUMPTION-D*: *ASSUMPTION* $A \Longrightarrow A$
  ⟨*proof*⟩

⟨*ML*⟩

3.1 Code generator setup

3.1.1 Generic code generator preprocessor setup

lemma *conj-left-cong*: $P \longleftrightarrow Q \Longrightarrow P \land R \longleftrightarrow Q \land R$
  ⟨*proof*⟩

lemma *disj-left-cong*: $P \longleftrightarrow Q \Longrightarrow P \lor R \longleftrightarrow Q \lor R$
  ⟨*proof*⟩

⟨*ML*⟩

3.1.2 Equality

class *equal* =
  fixes *equal* :: $'a \Rightarrow 'a \Rightarrow bool$

assumes *equal-eq*: $equal\ x\ y \longleftrightarrow x = y$
begin

lemma *equal*: $equal = (op\ =)$
⟨*proof*⟩

lemma *equal-refl*: $equal\ x\ x \longleftrightarrow True$
⟨*proof*⟩

lemma *eq-equal*: $(op\ =) \equiv equal$
⟨*proof*⟩

end

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

⟨*ML*⟩

3.1.3 Generic code generator foundation

Datatype *bool*

code-datatype *True False*

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

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

lemma [*code*]:
shows $(False \longrightarrow P) \longleftrightarrow True$
and $(True \longrightarrow P) \longleftrightarrow P$
and $(P \longrightarrow False) \longleftrightarrow \neg P$
and $(P \longrightarrow True) \longleftrightarrow True$
⟨*proof*⟩

More about *prop*

lemma [*code nbe*]:
shows $(True \Longrightarrow PROP\ Q) \equiv PROP\ Q$
and $(PROP\ Q \Longrightarrow True) \equiv Trueprop\ True$

and $(P \Longrightarrow R) \equiv \mathit{Trueprop}\ (P \longrightarrow R)$
⟨*proof*⟩

lemma *Trueprop-code* [*code*]: *Trueprop True* ≡ *Code-Generator.holds*
⟨*proof*⟩

declare *Trueprop-code* [*symmetric*, *code-post*]

Equality

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

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

definition *equal-itself* :: $'a$ *itself* ⇒ $'a$ *itself* ⇒ *bool*
where *equal-itself* $x\ y \longleftrightarrow x = y$

instance
⟨*proof*⟩

end

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

⟨*ML*⟩

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

⟨*ML*⟩

Cases

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

⟨*ML*⟩

declare [[*code abort*: *undefined*]]

3.1.4 Generic code generator target languages

type *bool*

code-printing

type-constructor *bool* ⇀
(*SML*) *bool* **and** (*OCaml*) *bool* **and** (*Haskell*) *Bool* **and** (*Scala*) *Boolean*
| **constant** *True* ⇀
(*SML*) *true* **and** (*OCaml*) *true* **and** (*Haskell*) *True* **and** (*Scala*) *true*
| **constant** *False* ⇀
(*SML*) *false* **and** (*OCaml*) *false* **and** (*Haskell*) *False* **and** (*Scala*) *false*

code-reserved *SML*
bool true false

code-reserved *OCaml*
bool

code-reserved *Scala*
Boolean

code-printing
constant *Not* ⇀
(*SML*) *not* **and** (*OCaml*) *not* **and** (*Haskell*) *not* **and** (*Scala*) '! -
| **constant** *HOL.conj* ⇀
(*SML*) **infixl** *1 andalso* **and** (*OCaml*) **infixl** *3* && **and** (*Haskell*) **infixr** *3* && **and** (*Scala*) **infixl** *3* &&
| **constant** *HOL.disj* ⇀
(*SML*) **infixl** *0 orelse* **and** (*OCaml*) **infixl** *2* || **and** (*Haskell*) **infixl** *2* || **and** (*Scala*) **infixl** *1* ||
| **constant** *HOL.implies* ⇀
(*SML*) !(*if* (-)/ *then* (-)/ *else true*)
and (*OCaml*) !(*if* (-)/ *then* (-)/ *else true*)
and (*Haskell*) !(*if* (-)/ *then* (-)/ *else True*)
and (*Scala*) !(*if* ((-))/ (-)/ *else true*)
| **constant** *If* ⇀
(*SML*) !(*if* (-)/ *then* (-)/ *else* (-))
and (*OCaml*) !(*if* (-)/ *then* (-)/ *else* (-))
and (*Haskell*) !(*if* (-)/ *then* (-)/ *else* (-))
and (*Scala*) !(*if* ((-))/ (-)/ *else* (-))

code-reserved *SML*
not

code-reserved *OCaml*
not

code-identifier
code-module *Pure* ⇀
(*SML*) *HOL* **and** (*OCaml*) *HOL* **and** (*Haskell*) *HOL* **and** (*Scala*) *HOL*

Using built-in Haskell equality.

code-printing
type-class *equal* ⇀ (*Haskell*) *Eq*

| **constant** *HOL.equal* ⇀ (*Haskell*) **infix** *4* ==
| **constant** *HOL.eq* ⇀ (*Haskell*) **infix** *4* ==

undefined

code-printing
  constant *undefined* ⇀
    (*SML*) !(*raise*/ *Fail*/ *undefined*)
    and (*OCaml*) *failwith*/ *undefined*
    and (*Haskell*) *error*/ *undefined*
    and (*Scala*) !*sys*.*error*(*undefined*)

3.1.5 Evaluation and normalization by evaluation

⟨*ML*⟩

3.2 Counterexample Search Units

3.2.1 Quickcheck

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

3.2.2 Nitpick setup

named-theorems *nitpick-unfold alternative definitions of constants as needed by Nitpick*
  and *nitpick-simp equational specification of constants as needed by Nitpick*
  and *nitpick-psimp partial equational specification of constants as needed by Nitpick*
  and *nitpick-choice-spec choice specification of constants as needed by Nitpick*

declare *if-bool-eq-conj* [*nitpick-unfold*, *no-atp*]
  and *if-bool-eq-disj* [*no-atp*]

3.3 Preprocessing for the predicate compiler

named-theorems *code-pred-def alternative definitions of constants for the Predicate Compiler*
  and *code-pred-inline inlining definitions for the Predicate Compiler*
  and *code-pred-simp simplification rules for the optimisations in the Predicate Compiler*

3.4 Legacy tactics and ML bindings

⟨*ML*⟩

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

end

4 Abstract orderings

theory *Orderings*
imports *HOL*
keywords *print-orders* :: *diag*
begin

⟨*ML*⟩

4.1 Abstract ordering

locale *ordering* =
fixes *less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\leq$ *50*)
and *less* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $<$ *50*)
assumes *strict-iff-order*: $a < b \longleftrightarrow a \leq b \wedge a \neq b$
assumes *refl*: $a \leq a$ — not *iff*: makes problems due to multiple (dual) interpretations
and *antisym*: $a \leq b \Longrightarrow b \leq a \Longrightarrow a = b$
and *trans*: $a \leq b \Longrightarrow b \leq c \Longrightarrow a \leq c$
begin

lemma *strict-implies-order*:
$a < b \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *strict-implies-not-eq*:
$a < b \Longrightarrow a \neq b$
⟨*proof*⟩

lemma *not-eq-order-implies-strict*:
$a \neq b \Longrightarrow a \leq b \Longrightarrow a < b$
⟨*proof*⟩

lemma *order-iff-strict*:
$a \leq b \longleftrightarrow a < b \vee a = b$
⟨*proof*⟩

lemma *irrefl*: — not *iff*: makes problems due to multiple (dual) interpretations
$\neg\ a < a$
⟨*proof*⟩

lemma *asym*:
$a < b \Longrightarrow b < a \Longrightarrow False$
⟨*proof*⟩

lemma *strict-trans1*:
$a \leq b \Longrightarrow b < c \Longrightarrow a < c$
⟨*proof*⟩

lemma *strict-trans2*:

$a < b \Longrightarrow b \leq c \Longrightarrow a < c$
⟨*proof*⟩

lemma *strict-trans*:
$a < b \Longrightarrow b < c \Longrightarrow a < c$
⟨*proof*⟩

end

Alternative introduction rule with bias towards strict order

lemma *ordering-strictI*:
fixes *less-eq* (**infix** $\leq$ *50*)
and *less* (**infix** $<$ *50*)
assumes *less-eq-less*: $\bigwedge a\ b.\ a \leq b \longleftrightarrow a < b \vee a = b$
assumes *asym*: $\bigwedge a\ b.\ a < b \Longrightarrow \neg\ b < a$
assumes *irrefl*: $\bigwedge a.\ \neg\ a < a$
assumes *trans*: $\bigwedge a\ b\ c.\ a < b \Longrightarrow b < c \Longrightarrow a < c$
shows *ordering less-eq less*
⟨*proof*⟩

lemma *ordering-dualI*:
fixes *less-eq* (**infix** $\leq$ *50*)
and *less* (**infix** $<$ *50*)
assumes *ordering* $(\lambda a\ b.\ b \leq a)$ $(\lambda a\ b.\ b < a)$
shows *ordering less-eq less*
⟨*proof*⟩

locale *ordering-top* = *ordering* +
fixes *top* :: $'a$ $(\top)$
assumes *extremum* [*simp*]: $a \leq \top$
begin

lemma *extremum-uniqueI*:
$\top \leq a \Longrightarrow a = \top$
⟨*proof*⟩

lemma *extremum-unique*:
$\top \leq a \longleftrightarrow a = \top$
⟨*proof*⟩

lemma *extremum-strict* [*simp*]:
$\neg\ (\top < a)$
⟨*proof*⟩

lemma *not-eq-extremum*:
$a \neq \top \longleftrightarrow a < \top$
⟨*proof*⟩

end

4.2 Syntactic orders

class *ord* =
  fixes *less-eq* :: $'a \Rightarrow 'a \Rightarrow$ *bool*
    and *less* :: $'a \Rightarrow 'a \Rightarrow$ *bool*
begin

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

abbreviation (*input*)
  greater-eq (**infix** $\geq$ *50*)
  where $x \geq y \equiv y \leq x$

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

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

notation (*input*)
  greater-eq (**infix** >= *50*)

end

4.3 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

4.4 Partial orders

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

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

sublocale *order*: *ordering less-eq less* + *dual-order*: *ordering greater-eq greater*
⟨*proof*⟩

Reflexivity.

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*: $x \le y \Longrightarrow x < y \lor x = y$
⟨*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*: $a \ne b \Longrightarrow a \le b \Longrightarrow a < b$
⟨*proof*⟩

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

Asymmetry.

lemma *eq-iff*: $x = y \longleftrightarrow x \le y \land y \le x$
⟨*proof*⟩

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

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

Least value operator

definition (**in** *ord*)
  $\mathit{Least} :: ('a \Rightarrow \mathit{bool}) \Rightarrow 'a$ (**binder** *LEAST 10*) **where**
  $\mathit{Least}\ P = (\mathit{THE}\ x.\ P\ x \land (\forall y.\ P\ y \longrightarrow x \le y))$

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

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

Greatest value operator

definition *Greatest* :: $('a \Rightarrow bool) \Rightarrow 'a$ (**binder** *GREATEST* *10*) **where**
$Greatest\ P = (THE\ x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow x \geq y))$

lemma *GreatestI2-order*:
$[\![\ P\ x;$
$\bigwedge y.\ P\ y \Longrightarrow x \geq y;$
$\bigwedge x.\ [\![\ P\ x; \forall y.\ P\ y \longrightarrow x \geq y\ ]\!] \Longrightarrow Q\ x\ ]\!]$
$\Longrightarrow Q\ (Greatest\ P)$
⟨*proof*⟩

lemma *Greatest-equality*:
$[\![\ P\ x;\ \bigwedge y.\ P\ y \Longrightarrow x \geq y\ ]\!] \Longrightarrow Greatest\ P = x$
⟨*proof*⟩

end

lemma *ordering-orderI*:
fixes *less-eq* (**infix** $\leq$ *50*)
and *less* (**infix** $<$ *50*)
assumes *ordering less-eq less*
shows *class.order less-eq less*
⟨*proof*⟩

lemma *order-strictI*:
fixes *less* (**infix** $\sqsubset$ *50*)
and *less-eq* (**infix** $\sqsubseteq$ *50*)
assumes $\bigwedge a\ b.\ a \sqsubseteq b \longleftrightarrow a \sqsubset b \vee a = b$
assumes $\bigwedge a\ b.\ a \sqsubset b \Longrightarrow \neg\ b \sqsubset a$
assumes $\bigwedge a.\ \neg\ a \sqsubset a$
assumes $\bigwedge a\ b\ c.\ a \sqsubset b \Longrightarrow b \sqsubset c \Longrightarrow a \sqsubset c$
shows *class.order less-eq less*
⟨*proof*⟩

context *order*
begin

Dual order

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

end

4.5 Linear (total) orders

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

lemma *less-linear*: $x < y \lor x = y \lor y < x$
$\langle proof \rangle$

lemma *le-less-linear*: $x \le y \lor y < x$
$\langle proof \rangle$

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

lemma (**in** *linorder*) *le-cases3*:
$[\![[\![x \le y; y \le z]\!] \Longrightarrow P; [\![y \le x; x \le z]\!] \Longrightarrow P; [\![x \le z; z \le y]\!] \Longrightarrow P;$
$[\![z \le y; y \le x]\!] \Longrightarrow P; [\![y \le z; z \le x]\!] \Longrightarrow P; [\![z \le x; x \le y]\!] \Longrightarrow P]\!] \Longrightarrow P$
$\langle proof \rangle$

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

lemma *linorder-wlog*[*case-names le sym*]:
$(\bigwedge a\ b.\ a \le b \Longrightarrow P\ a\ b) \Longrightarrow (\bigwedge a\ b.\ P\ b\ a \Longrightarrow P\ a\ b) \Longrightarrow P\ a\ b$
$\langle proof \rangle$

lemma *not-less*: $\neg\ x < y \longleftrightarrow y \le x$
$\langle proof \rangle$

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

lemma *not-le*: $\neg\ x \le y \longleftrightarrow y < x$
$\langle proof \rangle$

lemma *neq-iff*: $x \ne y \longleftrightarrow x < y \lor y < x$
$\langle proof \rangle$

lemma *neqE*: $x \ne y \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R$
$\langle proof \rangle$

lemma *antisym-conv1*: $\neg\ x < y \Longrightarrow x \le y \longleftrightarrow x = y$
$\langle proof \rangle$

lemma *antisym-conv2*: $x \le y \Longrightarrow \neg\ x < y \longleftrightarrow x = y$
$\langle proof \rangle$

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

lemma *leI*: $\neg\ x < y \Longrightarrow y \le x$

⟨*proof*⟩

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

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

lemma *linorder-less-wlog*[*case-names less refl sym*]:
$[\![\bigwedge a\ b.\ a < b \Longrightarrow P\ a\ b;\ \ \bigwedge a.\ P\ a\ a;\ \ \bigwedge a\ b.\ P\ b\ a \Longrightarrow P\ a\ b]\!] \Longrightarrow P\ a\ b$
⟨*proof*⟩

Dual order

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

end

Alternative introduction rule with bias towards strict order

lemma *linorder-strictI*:
fixes *less-eq* (**infix** $\le$ *50*)
and *less* (**infix** $<$ *50*)
assumes *class.order less-eq less*
assumes *trichotomy*: $\bigwedge a\ b.\ a < b \vee a = b \vee b < a$
shows *class.linorder less-eq less*
⟨*proof*⟩

4.6 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* = :: $'a \Rightarrow 'a \Rightarrow$ *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*⟩

4.7 Bounded quantifiers

syntax (*ASCII*)
  -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
  -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 (*input*)
  -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*)

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

⟨*ML*⟩

4.8 Transitivity reasoning

context *ord*
begin

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

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

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

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

end

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

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

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

lemmas [*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* [*no-atp*]:
$a = b ==> b > c ==> a > c$
$a > b ==> b = c ==> a > c$
$a = b ==> b >= c ==> a >= c$
$a >= b ==> b = c ==> a >= c$
$(x::'a::order) >= y ==> y >= x ==> x = y$
$(x::'a::order) >= y ==> y >= z ==> x >= z$
$(x::'a::order) > y ==> y >= z ==> x > z$
$(x::'a::order) >= y ==> y > z ==> x > z$
$(a::'a::order) > b ==> b > a ==> P$
$(x::'a::order) > y ==> y > z ==> x > z$
$(a::'a::order) >= b ==> a\ \tilde{}= b ==> a > b$
$(a::'a::order)\ \tilde{}= b ==> a >= b ==> a > b$

$a = f\ b \implies b > c \implies (\bigwedge x\ y.\ x > y \implies f\ x > f\ y) \implies a > f\ c$
$a > b \implies f\ b = c \implies (\bigwedge x\ y.\ x > y \implies f\ x > f\ y) \implies f\ a > c$
$a = f\ b \implies b >= c \implies (\bigwedge x\ y.\ x >= y \implies f\ x >= f\ y) \implies a >= f\ c$
$a >= b \implies f\ b = c \implies (\bigwedge x\ y.\ x >= y \implies f\ x >= f\ y) \implies f\ a >= c$
⟨*proof*⟩

lemma *xt2* [*no-atp*]:
$(a::'a::order) >= f\ b ==> b >= c ==> (!!x\ y.\ x >= y ==> f\ x >= f\ y) ==>$
$a >= f\ c$
⟨*proof*⟩

lemma *xt3* [*no-atp*]: $(a::'a::order) >= b ==> (f\ b::'b::order) >= c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> f\ a >= c$
⟨*proof*⟩

lemma *xt4* [*no-atp*]: $(a::'a::order) > f\ b ==> (b::'b::order) >= c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt5* [*no-atp*]: $(a::'a::order) > b ==> (f\ b::'b::order) >= c==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> f\ a > c$
⟨*proof*⟩

lemma *xt6* [*no-atp*]: $(a::'a::order) >= f\ b ==> b > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt7* [*no-atp*]: $(a::'a::order) >= b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> f\ a > c$
⟨*proof*⟩

lemma *xt8* [*no-atp*]: $(a::'a::order) > f\ b ==> (b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
⟨*proof*⟩

lemma *xt9* [*no-atp*]: $(a::'a::order) > b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> f\ a > c$
⟨*proof*⟩

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

4.9 Monotonicity

context *order*
begin

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

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*⟩

lemma *monoE*:
  fixes $f :: 'a \Rightarrow 'b::order$
  assumes $mono\ f$
  assumes $x \leq y$
  obtains $f\ x \leq f\ y$
⟨*proof*⟩

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

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

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

lemma *antimonoE*:
  fixes $f :: 'a \Rightarrow 'b::order$
  assumes $antimono\ f$
  assumes $x \leq y$
  obtains $f\ x \geq f\ y$
⟨*proof*⟩

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

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

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

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

end

context *linorder*
begin

lemma *mono-invE*:
  fixes $f :: 'a \Rightarrow 'b::order$
  assumes *mono f*
  assumes $f\ x < f\ y$
  obtains $x \leq y$
⟨*proof*⟩

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*⟩

end

4.10 min and max – fundamental

definition (**in** *ord*) $min :: 'a \Rightarrow 'a \Rightarrow 'a$ **where**
  $min\ a\ b = (if\ a \leq b\ then\ a\ else\ b)$

definition (**in** *ord*) $max :: 'a \Rightarrow 'a \Rightarrow 'a$ **where**
  $max\ a\ b = (if\ a \leq b\ then\ b\ else\ a)$

lemma *min-absorb1*: $x \leq y \Longrightarrow min\ x\ y = x$
  ⟨*proof*⟩

lemma *max-absorb2*: $x \leq y \Longrightarrow max\ x\ y = y$
  ⟨*proof*⟩

lemma *min-absorb2*: $(y::'a::order) \leq x \Longrightarrow min\ x\ y = y$

⟨*proof*⟩

lemma *max-absorb1*: $(y::'a::order) \leq x \Longrightarrow max\ x\ y = x$
⟨*proof*⟩

lemma *max-min-same* [*simp*]:
fixes $x\ y :: 'a :: linorder$
shows $max\ x\ (min\ x\ y) = x$ $max\ (min\ x\ y)\ x = x$ $max\ (min\ x\ y)\ y = y$ $max\ y\ (min\ x\ y) = y$
⟨*proof*⟩

4.11 (Unique) top and bottom elements

class *bot* =
fixes *bot* :: $'a$ (⊥)

class *order-bot* = *order* + *bot* +
assumes *bot-least*: $\bot \leq a$
begin

sublocale *bot*: *ordering-top greater-eq greater bot*
⟨*proof*⟩

lemma *le-bot*:
$a \leq \bot \Longrightarrow a = \bot$
⟨*proof*⟩

lemma *bot-unique*:
$a \leq \bot \longleftrightarrow a = \bot$
⟨*proof*⟩

lemma *not-less-bot*:
$\neg\ a < \bot$
⟨*proof*⟩

lemma *bot-less*:
$a \neq \bot \longleftrightarrow \bot < a$
⟨*proof*⟩

end

class *top* =
fixes *top* :: $'a$ (⊤)

class *order-top* = *order* + *top* +
assumes *top-greatest*: $a \leq \top$
begin

sublocale *top*: *ordering-top less-eq less top*

⟨*proof*⟩

lemma *top-le*:
$\top \leq a \Longrightarrow a = \top$
⟨*proof*⟩

lemma *top-unique*:
$\top \leq a \longleftrightarrow a = \top$
⟨*proof*⟩

lemma *not-top-less*:
$\neg \top < a$
⟨*proof*⟩

lemma *less-top*:
$a \neq \top \longleftrightarrow a < \top$
⟨*proof*⟩

end

4.12 Dense orders

class *dense-order* = *order* +
assumes *dense*: $x < y \Longrightarrow (\exists z.\ x < z \wedge z < y)$

class *dense-linorder* = *linorder* + *dense-order*
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*⟩

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

lemma *dense-ge-bounded*:
fixes $x\ y\ z :: 'a$

assumes $z < x$
assumes $*$: $\bigwedge w.\ [\![z < w ; w < x]\!] \Longrightarrow y \leq w$
shows $y \leq z$
⟨*proof*⟩

end

class *no-top* = *order* +
assumes *gt-ex*: $\exists\, y.\ x < y$

class *no-bot* = *order* +
assumes *lt-ex*: $\exists\, y.\ y < x$

class *unbounded-dense-linorder* = *dense-linorder* + *no-top* + *no-bot*

4.13 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 *LeastI2-wellorder*:
assumes $P\ a$
and $\bigwedge a.\ [\![P\ a; \forall\, b.\ P\ b \longrightarrow a \leq b]\!] \Longrightarrow Q\ a$
shows $Q\ (Least\ P)$
⟨*proof*⟩

lemma *LeastI2-wellorder-ex*:
assumes $\exists\, x.\ P\ x$
and $\bigwedge a.\ [\![P\ a; \forall\, b.\ P\ b \longrightarrow a \leq b]\!] \Longrightarrow Q\ a$
shows $Q\ (Least\ P)$
⟨*proof*⟩

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

lemma *exists-least-iff*: $(\exists\, n.\ P\ n) \longleftrightarrow (\exists\, n.\ P\ n \wedge (\forall\, m < n.\ \neg\ P\ m))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

end

4.14 Order on *bool*

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

definition
le-bool-def [*simp*]: $P \le Q \longleftrightarrow P \longrightarrow Q$

definition
[*simp*]: $(P{::}bool) < Q \longleftrightarrow \neg\ P \wedge Q$

definition
[*simp*]: $\bot \longleftrightarrow False$

definition
[*simp*]: $\top \longleftrightarrow True$

instance ⟨*proof*⟩

end

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

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

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

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

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

lemma *top-boolI*: $\top$
⟨*proof*⟩

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

4.15 Order on - ⇒ -

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

definition
le-fun-def: $f \le g \longleftrightarrow (\forall x.\ f\ x \le g\ x)$

definition
$(f{::}'a \Rightarrow 'b) < g \longleftrightarrow f \le g \wedge \neg (g \le f)$

instance ⟨*proof*⟩

end

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

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

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

definition
$\bot = (\lambda x.\ \bot)$

instance ⟨*proof*⟩

end

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

lemma *bot-apply* [*simp*, *code*]:
$\bot\ x = \bot$
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

definition
[*no-atp*]: $\top = (\lambda x.\ \top)$

instance ⟨*proof*⟩

end

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

lemma *top-apply* [*simp*, *code*]:
$\top\ x = \top$
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

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

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

lemma *mono-compose*: *mono* $Q \Longrightarrow$ *mono* $(\lambda i\ x.\ Q\ i\ (f\ x))$
⟨*proof*⟩

4.16 Order on unary and binary predicates

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

lemma *predicate1D*:
$P \le Q \Longrightarrow P\ x \Longrightarrow Q\ x$
⟨*proof*⟩

lemma *rev-predicate1D*:
$P\ x \Longrightarrow P \le Q \Longrightarrow Q\ x$
⟨*proof*⟩

lemma *predicate2I*:
assumes *PQ*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y$
shows $P \le Q$

⟨*proof*⟩

lemma *predicate2D*:
$P \leq Q \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
⟨*proof*⟩

lemma *rev-predicate2D*:
$P\ x\ y \Longrightarrow P \leq Q \Longrightarrow Q\ x\ y$
⟨*proof*⟩

lemma *bot1E* [*no-atp*]: $\bot\ x \Longrightarrow P$
⟨*proof*⟩

lemma *bot2E*: $\bot\ x\ y \Longrightarrow P$
⟨*proof*⟩

lemma *top1I*: $\top\ x$
⟨*proof*⟩

lemma *top2I*: $\top\ x\ y$
⟨*proof*⟩

4.17 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

5 Groups, also combined with orderings

theory *Groups*
  imports *Orderings*
begin

5.1 Dynamic facts

named-theorems *ac-simps associativity and commutativity simplification rules*
  and *algebra-simps algebra simplification rules*
  and *field-simps algebra simplification rules for fields*

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*.

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

5.2 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*]: $\mathbf{1} * a = a$
assumes *right-neutral* [*simp*]: $a * \mathbf{1} = a$

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

sublocale *monoid*
⟨*proof*⟩

end

locale *group* = *semigroup* +
fixes z :: $'a$ (**1**)
fixes *inverse* :: $'a \Rightarrow 'a$
assumes *group-left-neutral*: $\mathbf{1} * a = a$
assumes *left-inverse* [*simp*]: *inverse* $a * a = \mathbf{1}$
begin

lemma *left-cancel*: $a * b = a * c \longleftrightarrow b = c$
⟨*proof*⟩

sublocale *monoid*
⟨*proof*⟩

lemma *inverse-unique*:
assumes $a * b = \mathbf{1}$
shows *inverse* $a = b$
⟨*proof*⟩

lemma *inverse-neutral* [*simp*]: *inverse* $\mathbf{1} = \mathbf{1}$
⟨*proof*⟩

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

lemma *right-inverse* [*simp*]: $a *$ *inverse* $a = \mathbf{1}$
⟨*proof*⟩

lemma *inverse-distrib-swap*: *inverse* $(a * b) =$ *inverse* $b *$ *inverse* a
⟨*proof*⟩

lemma *right-cancel*: $b * a = c * a \longleftrightarrow b = c$
⟨*proof*⟩

end

5.3 Generic operations

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

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

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

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* :: $'a \Rightarrow 'a$ (− - [*81*] *80*)

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

5.4 Semigroups and Monoids

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

sublocale *add*: *semigroup plus*
 ⟨*proof*⟩

end

hide-fact *add-assoc*

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

sublocale *add*: *abel-semigroup plus*
  ⟨*proof*⟩

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

lemmas *add-ac* = *add.assoc add.commute add.left-commute*

end

hide-fact *add-commute*

lemmas *add-ac* = *add.assoc add.commute add.left-commute*

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

sublocale *mult*: *semigroup times*
  ⟨*proof*⟩

end

hide-fact *mult-assoc*

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

sublocale *mult*: *abel-semigroup times*
  ⟨*proof*⟩

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

lemmas *mult-ac* = *mult.assoc mult.commute mult.left-commute*

end

hide-fact *mult-commute*

lemmas *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*: $a + 0 = a$
begin

sublocale *add*: *monoid plus 0*
⟨*proof*⟩

end

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

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

subclass *monoid-add*
⟨*proof*⟩

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

end

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

sublocale *mult*: *monoid times 1*
⟨*proof*⟩

end

lemma *one-reorient*: $1 = x \longleftrightarrow x = 1$
⟨*proof*⟩

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

subclass *monoid-mult*
⟨*proof*⟩

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

end

class *cancel-semigroup-add* = *semigroup-add* +

assumes *add-left-imp-eq*: $a + b = a + c \Longrightarrow b = c$
assumes *add-right-imp-eq*: $b + a = c + a \Longrightarrow b = c$
begin

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

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

end

class *cancel-ab-semigroup-add* = *ab-semigroup-add* + *minus* +
assumes *add-diff-cancel-left'* [*simp*]: $(a + b) - a = b$
assumes *diff-diff-add* [*algebra-simps*, *field-simps*]: $a - b - c = a - (b + c)$
begin

lemma *add-diff-cancel-right'* [*simp*]: $(a + b) - b = a$
⟨*proof*⟩

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

lemma *add-diff-cancel-left* [*simp*]: $(c + a) - (c + b) = a - b$
⟨*proof*⟩

lemma *add-diff-cancel-right* [*simp*]: $(a + c) - (b + c) = a - b$
⟨*proof*⟩

lemma *diff-right-commute*: $a - c - b = a - b - c$
⟨*proof*⟩

end

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

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

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

lemma *add-implies-diff*:
assumes $c + b = a$
shows $c = a - b$
⟨*proof*⟩

lemma *add-cancel-right-right* [*simp*]: $a = a + b \longleftrightarrow b = 0$

(**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *add-cancel-right-left* [*simp*]: $a = b + a \longleftrightarrow b = 0$
⟨*proof*⟩

lemma *add-cancel-left-right* [*simp*]: $a + b = a \longleftrightarrow b = 0$
⟨*proof*⟩

lemma *add-cancel-left-left* [*simp*]: $b + a = a \longleftrightarrow b = 0$
⟨*proof*⟩

end

class *comm-monoid-diff* = *cancel-comm-monoid-add* +
assumes *zero-diff* [*simp*]: $0 - a = 0$
begin

lemma *diff-add-zero* [*simp*]: $a - (a + b) = 0$
⟨*proof*⟩

end

5.5 Groups

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

lemma *diff-conv-add-uminus*: $a - b = a + (- b)$
⟨*proof*⟩

sublocale *add*: *group plus 0 uminus*
⟨*proof*⟩

lemma *minus-unique*: $a + b = 0 \Longrightarrow - a = b$
⟨*proof*⟩

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *add-eq-0-iff2*: $a + b = 0 \longleftrightarrow a = - b$
⟨*proof*⟩

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

lemma *add-eq-0-iff*: $a + b = 0 \longleftrightarrow b = - a$
⟨*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-diff-diff-swap*: $a - (b + c) = a - c - b$
⟨*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-eq2* [*algebra-simps*, *field-simps*]: $a - (b - c) = (a + c) - b$
⟨*proof*⟩

lemma *diff-eq-diff-eq*: $a - b = c - d \Longrightarrow a = b \longleftrightarrow c = d$
⟨*proof*⟩

end

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

subclass *group-add*
⟨*proof*⟩

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

lemma *uminus-add-conv-diff* [*simp*]: $- a + b = b - a$

⟨*proof*⟩

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

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

end

5.6 (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

Strict monotonicity in both arguments

class *strict-ordered-ab-semigroup-add* = *ordered-ab-semigroup-add* +
assumes *add-strict-mono*: $a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$

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*⟩

subclass *strict-ordered-ab-semigroup-add*
⟨*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*: $\mathit{max}\ x\ y + z = \mathit{max}\ (x + z)\ (y + z)$
⟨*proof*⟩

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

lemma *max-add-distrib-right*: $x + \mathit{max}\ y\ z = \mathit{max}\ (x + y)\ (x + z)$

⟨*proof*⟩

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

end

5.7 Support for reasoning about signs

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

lemma *add-nonneg-nonneg* [*simp*]: $0 \le a \Longrightarrow 0 \le b \Longrightarrow 0 \le a + b$
⟨*proof*⟩

lemma *add-nonpos-nonpos*: $a \le 0 \Longrightarrow b \le 0 \Longrightarrow a + b \le 0$
⟨*proof*⟩

lemma *add-nonneg-eq-0-iff*: $0 \le x \Longrightarrow 0 \le y \Longrightarrow x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemma *add-nonpos-eq-0-iff*: $x \le 0 \Longrightarrow y \le 0 \Longrightarrow x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

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

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

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

lemma *add-decreasing2*: $c \le 0 \Longrightarrow a \le b \Longrightarrow a + c \le b$
⟨*proof*⟩

lemma *add-pos-nonneg*: $0 < a \Longrightarrow 0 \le b \Longrightarrow 0 < a + b$
⟨*proof*⟩

lemma *add-pos-pos*: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a + b$
⟨*proof*⟩

lemma *add-nonneg-pos*: $0 \le a \Longrightarrow 0 < b \Longrightarrow 0 < a + b$
⟨*proof*⟩

lemma *add-neg-nonpos*: $a < 0 \Longrightarrow b \le 0 \Longrightarrow a + b < 0$
⟨*proof*⟩

lemma *add-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow a + b < 0$
⟨*proof*⟩

lemma *add-nonpos-neg*: $a \leq 0 \Longrightarrow b < 0 \Longrightarrow a + b < 0$
⟨*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

end

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

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

end

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

subclass *ordered-cancel-ab-semigroup-add* ⟨*proof*⟩
subclass *strict-ordered-comm-monoid-add* ⟨*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 *ordered-ab-semigroup-monoid-add-imp-le* = *monoid-add* + *ordered-ab-semigroup-add-imp-le*
begin

lemma *add-less-same-cancel1* [*simp*]: $b + a < b \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *add-less-same-cancel2* [*simp*]: $a + b < b \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *less-add-same-cancel1* [*simp*]: $a < a + b \longleftrightarrow 0 < b$
⟨*proof*⟩

lemma *less-add-same-cancel2* [*simp*]: $a < b + a \longleftrightarrow 0 < b$
⟨*proof*⟩

lemma *add-le-same-cancel1* [*simp*]: $b + a \leq b \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *add-le-same-cancel2* [*simp*]: $a + b \leq b \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *le-add-same-cancel1* [*simp*]: $a \leq a + b \longleftrightarrow 0 \leq b$
⟨*proof*⟩

lemma *le-add-same-cancel2* [*simp*]: $a \leq b + a \longleftrightarrow 0 \leq b$
⟨*proof*⟩

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

subclass *ordered-cancel-comm-monoid-add*
⟨*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-monoid-add-imp-le*
⟨*proof*⟩

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

lemma *min-diff-distrib-left*: $min\ x\ y - z = min\ (x - z)\ (y - z)$
⟨*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 *diff-less-0-iff-less* [*simp*]: $a - b < 0 \longleftrightarrow a < b$
⟨*proof*⟩

lemmas *less-iff-diff-less-0* = *diff-less-0-iff-less* [*symmetric*]

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-gt-0-iff-gt* [*simp*]: $a - b > 0 \longleftrightarrow a > b$
⟨*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 *diff-le-0-iff-le* [*simp*]: $a - b \leq 0 \longleftrightarrow a \leq b$
⟨*proof*⟩

lemmas *le-iff-diff-le-0* = *diff-le-0-iff-le* [*symmetric*]

lemma *diff-ge-0-iff-ge* [*simp*]: $a - b \geq 0 \longleftrightarrow a \geq b$
⟨*proof*⟩

lemma *diff-eq-diff-less*: $a - b = c - d \Longrightarrow a < b \longleftrightarrow c < d$
⟨*proof*⟩

lemma *diff-eq-diff-less-eq*: $a - b = c - d \Longrightarrow a \leq b \longleftrightarrow c \leq d$
⟨*proof*⟩

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

lemma *diff-left-mono*: $b \leq a \Longrightarrow c - a \leq c - b$
⟨*proof*⟩

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

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

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

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

end

⟨*ML*⟩

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 *equal-neg-zero* [*simp*]: $a = -\ a \longleftrightarrow a = 0$

⟨*proof*⟩

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

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

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

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

lemma *less-neg-neg* [*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 *minus-max-eq-min*: $- \mathit{max}\ x\ y = \mathit{min}\ (- x)\ (- y)$
⟨*proof*⟩

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

end

class *abs* =
fixes *abs* :: $'a \Rightarrow 'a$ (|-|)

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

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*]: $0 = |a| \longleftrightarrow a = 0$
⟨*proof*⟩

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

lemma *abs-le-self-iff* [*simp*]: $|a| \leq a \longleftrightarrow 0 \leq a$
⟨*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

lemma *dense-eq0-I*:
  fixes $x::'a::\{dense\text{-}linorder,ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\}$
  shows $(\bigwedge e.\ 0 < e \Longrightarrow |x| \leq e) \Longrightarrow x = 0$
  ⟨*proof*⟩

hide-fact (**open**) *ab-diff-conv-add-uminus add-0 mult-1 ab-left-minus*

lemmas *add-0* = *add-0-left*
lemmas *mult-1* = *mult-1-left*
lemmas *ab-left-minus* = *left-minus*
lemmas *diff-diff-eq* = *diff-diff-add*

5.8 Canonically ordered monoids

Canonically ordered monoids are never groups.

class *canonically-ordered-monoid-add* = *comm-monoid-add* + *order* +
assumes *le-iff-add*: $a \leq b \longleftrightarrow (\exists\, c.\ b = a + c)$
begin

lemma *zero-le*[*simp*]: $0 \leq x$
⟨*proof*⟩

lemma *le-zero-eq*[*simp*]: $n \leq 0 \longleftrightarrow n = 0$
⟨*proof*⟩

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

lemma *zero-less-iff-neq-zero*: $0 < n \longleftrightarrow n \neq 0$
⟨*proof*⟩

This theorem is useful with *blast*

lemma *gr-zeroI*: $(n = 0 \Longrightarrow \mathit{False}) \Longrightarrow 0 < n$
⟨*proof*⟩

lemma *not-gr-zero*[*simp*]: $\neg\ 0 < n \longleftrightarrow n = 0$
⟨*proof*⟩

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

lemma *gr-implies-not-zero*: $m < n \Longrightarrow n \neq 0$
⟨*proof*⟩

lemma *add-eq-0-iff-both-eq-0*[*simp*]: $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemma *zero-eq-add-iff-both-eq-0*[*simp*]: $0 = x + y \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemmas *zero-order* = *zero-le le-zero-eq not-less-zero zero-less-iff-neq-zero not-gr-zero*
— This should be attributed with [*iff*], but then *blast* fails in *Set*.

end

class *ordered-cancel-comm-monoid-diff* =
canonically-ordered-monoid-add + *comm-monoid-diff* + *ordered-ab-semigroup-add-imp-le*
begin

context
fixes $a\ b :: 'a$

assumes *le*: $a \le b$
begin

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

lemma *add-diff-assoc*: $c + (b - a) = c + b - a$
⟨*proof*⟩

lemma *add-diff-assoc2*: $b - a + c = b + c - a$
⟨*proof*⟩

lemma *diff-add-assoc*: $c + b - a = c + (b - a)$
⟨*proof*⟩

lemma *diff-add-assoc2*: $b + c - a = b - a + c$
⟨*proof*⟩

lemma *diff-diff-right*: $c - (b - a) = c + a - b$
⟨*proof*⟩

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

lemma *le-add-diff*: $c \le b + c - a$
⟨*proof*⟩

lemma *le-imp-diff-is-add*: $a \le b \Longrightarrow b - a = c \longleftrightarrow b = c + a$
⟨*proof*⟩

lemma *le-diff-conv2*: $c \le b - a \longleftrightarrow c + a \le b$
(**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

end

end

5.9 Tools setup

lemma *add-mono-thms-linordered-semiring*:
fixes $i\ j\ k$ :: *'a::ordered-ab-semigroup-add*
shows $i \le j \land k \le l \Longrightarrow i + k \le j + l$
and $i = j \land k \le l \Longrightarrow i + k \le j + l$
and $i \le j \land k = l \Longrightarrow i + k \le j + l$
and $i = j \land k = l \Longrightarrow i + k = j + l$
⟨*proof*⟩

lemma *add-mono-thms-linordered-field*:

fixes $i\ j\ k :: 'a::ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add$
shows $i < j \land k = l \Longrightarrow i + k < j + l$
and $i = j \land k < l \Longrightarrow i + k < j + l$
and $i < j \land k \leq l \Longrightarrow i + k < j + l$
and $i \leq j \land k < l \Longrightarrow i + k < j + l$
and $i < j \land k < l \Longrightarrow i + k < j + l$
⟨*proof*⟩

code-identifier
code-module *Groups* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

6 Abstract lattices

theory *Lattices*
imports *Groups*
begin

6.1 Abstract semilattice

These 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*]: $a * a = a$
begin

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

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

end

locale *semilattice-neutr* = *semilattice* + *comm-monoid*

locale *semilattice-order* = *semilattice* +
fixes *less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\leq$ *50*)
and *less* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $<$ *50*)
assumes *order-iff*: $a \leq b \longleftrightarrow a = a * b$
and *strict-order-iff*: $a < b \longleftrightarrow a = a * b \land a \neq b$
begin

lemma *orderI*: $a = a * b \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *orderE*:
  assumes $a \leq b$
  obtains $a = a * b$
  ⟨*proof*⟩

sublocale *ordering less-eq less*
⟨*proof*⟩

lemma *cobounded1* [*simp*]: $a * b \leq a$
  ⟨*proof*⟩

lemma *cobounded2* [*simp*]: $a * b \leq b$
  ⟨*proof*⟩

lemma *boundedI*:
  assumes $a \leq b$ **and** $a \leq c$
  shows $a \leq b * c$
⟨*proof*⟩

lemma *boundedE*:
  assumes $a \leq b * c$
  obtains $a \leq b$ **and** $a \leq c$
  ⟨*proof*⟩

lemma *bounded-iff* [*simp*]: $a \leq b * c \longleftrightarrow a \leq b \wedge a \leq c$
  ⟨*proof*⟩

lemma *strict-boundedE*:
  assumes $a < b * c$
  obtains $a < b$ **and** $a < c$
  ⟨*proof*⟩

lemma *coboundedI1*: $a \leq c \Longrightarrow a * b \leq c$
  ⟨*proof*⟩

lemma *coboundedI2*: $b \leq c \Longrightarrow a * b \leq c$
  ⟨*proof*⟩

lemma *strict-coboundedI1*: $a < c \Longrightarrow a * b < c$
  ⟨*proof*⟩

lemma *strict-coboundedI2*: $b < c \Longrightarrow a * b < c$
  ⟨*proof*⟩

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

lemma *absorb1*: $a \leq b \Longrightarrow a * b = a$
  ⟨*proof*⟩

lemma *absorb2*: $b \leq a \Longrightarrow a * b = b$
$\langle proof \rangle$

lemma *absorb-iff1*: $a \leq b \longleftrightarrow a * b = a$
$\langle proof \rangle$

lemma *absorb-iff2*: $b \leq a \longleftrightarrow a * b = b$
$\langle proof \rangle$

end

locale *semilattice-neutr-order* = *semilattice-neutr* + *semilattice-order*
begin

sublocale *ordering-top less-eq less* **1**
$\langle proof \rangle$

end

Passive interpretations for boolean operators

lemma *semilattice-neutr-and*:
semilattice-neutr HOL.conj True
$\langle proof \rangle$

lemma *semilattice-neutr-or*:
semilattice-neutr HOL.disj False
$\langle proof \rangle$

6.2 Syntactic infimum and supremum operations

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

class *sup* =
fixes *sup* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcup$ *65*)

6.3 Concrete lattices

class *semilattice-inf* = *order* + *inf* +
assumes *inf-le1* [*simp*]: $x \sqcap y \leq x$
and *inf-le2* [*simp*]: $x \sqcap y \leq y$
and *inf-greatest*: $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$

class *semilattice-sup* = *order* + *sup* +
assumes *sup-ge1* [*simp*]: $x \leq x \sqcup y$
and *sup-ge2* [*simp*]: $y \leq x \sqcup y$
and *sup-least*: $y \leq x \Longrightarrow z \leq x \Longrightarrow y \sqcup z \leq x$
begin

Dual lattice.

lemma *dual-semilattice*: *class.semilattice-inf sup greater-eq greater*
⟨*proof*⟩

end

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

6.3.1 Intro and elim rules

context *semilattice-inf*
begin

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

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

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

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

lemma *le-inf-iff*: $x \leq y \sqcap z \longleftrightarrow x \leq y \wedge x \leq z$
⟨*proof*⟩

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

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

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

end

context *semilattice-sup*
begin

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

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

lemma *le-supI*: $a \leq x \Longrightarrow b \leq x \Longrightarrow a \sqcup b \leq x$

⟨*proof*⟩

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

lemma *le-sup-iff*: $x \sqcup y \leq z \longleftrightarrow x \leq z \wedge y \leq z$
⟨*proof*⟩

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

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

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

end

6.3.2 Equational laws

context *semilattice-inf*
begin

sublocale *inf*: *semilattice inf*
⟨*proof*⟩

sublocale *inf*: *semilattice-order inf less-eq less*
⟨*proof*⟩

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-right-idem*: $(x \sqcap y) \sqcap y = x \sqcap y$
⟨*proof*⟩

lemma *inf-absorb1*: $x \leq y \Longrightarrow x \sqcap y = x$

⟨*proof*⟩

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

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

end

context *semilattice-sup*
begin

sublocale *sup*: *semilattice sup*
⟨*proof*⟩

sublocale *sup*: *semilattice-order sup greater-eq greater*
⟨*proof*⟩

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* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
⟨*proof*⟩

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

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

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

end

context *lattice*
begin

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

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

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

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

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

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

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

end

6.3.3 Strict order

context *semilattice-inf*
begin

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

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

end

context *semilattice-sup*
begin

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

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

end

6.4 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 sup* (*op* $\geq$) (*op* $>$) *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

6.5 Bounded lattices and boolean algebras

class *bounded-semilattice-inf-top* = *semilattice-inf* + *order-top*
begin

sublocale *inf-top*: *semilattice-neutr inf top*
+ *inf-top*: *semilattice-neutr-order inf top less-eq less*
⟨*proof*⟩

end

class *bounded-semilattice-sup-bot* = *semilattice-sup* + *order-bot*
begin

sublocale *sup-bot*: *semilattice-neutr sup bot*
+ *sup-bot*: *semilattice-neutr-order sup bot greater-eq greater*
⟨*proof*⟩

end

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

subclass *bounded-semilattice-sup-bot* ⟨*proof*⟩

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

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

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

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

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

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

end

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

subclass *bounded-semilattice-inf-top* ⟨*proof*⟩

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

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

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

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

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

end

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

subclass *bounded-lattice-bot* ⟨*proof*⟩
subclass *bounded-lattice-top* ⟨*proof*⟩

lemma *dual-bounded-lattice*: *class.bounded-lattice sup greater-eq greater inf* $\top$ $\bot$
⟨*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 $(\lambda x\ y.\ x \sqcup - y)$ *uminus sup greater-eq greater inf* $\top$ $\bot$
⟨*proof*⟩

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

lemma *compl-sup-top* [*simp*]: $- x \sqcup x = \top$
⟨*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*:
assumes $x \leq y$
shows $- y \leq - x$
⟨*proof*⟩

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

lemma *compl-le-swap1*:
assumes $y \leq - x$
shows $x \leq -y$
⟨*proof*⟩

lemma *compl-le-swap2*:
assumes $- y \leq x$
shows $- x \leq y$
⟨*proof*⟩

lemma *compl-less-compl-iff*: $- x < - y \longleftrightarrow y < x$
⟨*proof*⟩

lemma *compl-less-swap1*:
assumes $y < - x$
shows $x < - y$
⟨*proof*⟩

lemma *compl-less-swap2*:
assumes $- y < x$
shows $- x < y$
⟨*proof*⟩

lemma *sup-cancel-left1*: *sup* (*sup* x a) (*sup* $(- x)$ b) = *top*
⟨*proof*⟩

lemma *sup-cancel-left2*: *sup* (*sup* $(- x)$ a) (*sup* x b) = *top*
⟨*proof*⟩

lemma *inf-cancel-left1*: *inf* (*inf* x a) (*inf* $(- x)$ b) = *bot*
⟨*proof*⟩

lemma *inf-cancel-left2*: *inf* (*inf* $(- x)$ a) (*inf* x b) = *bot*
⟨*proof*⟩

declare *inf-compl-bot* [*simp*]
and *sup-compl-top* [*simp*]

lemma *sup-compl-top-left1* [*simp*]: *sup* (− *x*) (*sup x y*) = *top*
⟨*proof*⟩

lemma *sup-compl-top-left2* [*simp*]: *sup x* (*sup* (− *x*) *y*) = *top*
⟨*proof*⟩

lemma *inf-compl-bot-left1* [*simp*]: *inf* (− *x*) (*inf x y*) = *bot*
⟨*proof*⟩

lemma *inf-compl-bot-left2* [*simp*]: *inf x* (*inf* (− *x*) *y*) = *bot*
⟨*proof*⟩

lemma *inf-compl-bot-right* [*simp*]: *inf x* (*inf y* (− *x*)) = *bot*
⟨*proof*⟩

end

⟨*ML*⟩

6.6 *min*/*max* as special case of lattice

context *linorder*
begin

sublocale *min*: *semilattice-order min less-eq less*
+ *max*: *semilattice-order max greater-eq greater*
⟨*proof*⟩

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 *min-max-distrib1*: *min* (*max b c*) *a* = *max* (*min b a*) (*min c a*)
⟨*proof*⟩

lemma *min-max-distrib2*: $min\ a\ (max\ b\ c) = max\ (min\ a\ b)\ (min\ a\ c)$
⟨*proof*⟩

lemma *max-min-distrib1*: $max\ (min\ b\ c)\ a = min\ (max\ b\ a)\ (max\ c\ a)$
⟨*proof*⟩

lemma *max-min-distrib2*: $max\ a\ (min\ b\ c) = min\ (max\ a\ b)\ (max\ a\ c)$
⟨*proof*⟩

lemmas *min-max-distribs* = *min-max-distrib1 min-max-distrib2 max-min-distrib1 max-min-distrib2*

lemma *split-min* [*no-atp*]: $P\ (min\ i\ j) \longleftrightarrow (i \le j \longrightarrow P\ i) \wedge (\neg\ i \le j \longrightarrow P\ j)$
⟨*proof*⟩

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

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

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

end

lemma *inf-min*: $inf = (min :: 'a::\{semilattice\text{-}inf,linorder\} \Rightarrow 'a \Rightarrow 'a)$
⟨*proof*⟩

lemma *sup-max*: $sup = (max :: 'a::\{semilattice\text{-}sup,linorder\} \Rightarrow 'a \Rightarrow 'a)$
⟨*proof*⟩

6.7 Uniqueness of inf and sup

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

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

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

6.8 Lattice on *bool*

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

definition *bool-Compl-def* [*simp*]: $uminus = Not$

definition *bool-diff-def* [*simp*]: $A - B \longleftrightarrow A \wedge \neg B$

definition [*simp*]: $P \sqcap Q \longleftrightarrow P \wedge Q$

definition [*simp*]: $P \sqcup Q \longleftrightarrow P \vee Q$

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*⟩

6.9 Lattice on - ⇒ -

instantiation *fun* :: (*type*, *semilattice-sup*) *semilattice-sup*
begin

definition $f \sqcup g = (\lambda x.\ f\ x \sqcup g\ x)$

lemma *sup-apply* [*simp*, *code*]: $(f \sqcup g)\ x = f\ x \sqcup g\ x$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *fun* :: (*type*, *semilattice-inf*) *semilattice-inf*
begin

definition $f \sqcap g = (\lambda x.\ f\ x \sqcap g\ x)$

lemma *inf-apply* [*simp*, *code*]: $(f \sqcap g)\ x = f\ x \sqcap g\ x$

⟨*proof*⟩

instance ⟨*proof*⟩

end

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

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*: $- A = (\lambda x. - A\ x)$

lemma *uminus-apply* [*simp*, *code*]: $(- A)\ x = - (A\ x)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

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

lemma *minus-apply* [*simp*, *code*]: $(A - B)\ x = A\ x - B\ x$
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

6.10 Lattice on unary and binary predicates

lemma *inf1I*: $A\ x \Longrightarrow B\ x \Longrightarrow (A \sqcap B)\ x$
⟨*proof*⟩

lemma *inf2I*: $A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow (A \sqcap B)\ x\ y$
⟨*proof*⟩

lemma *inf1E*: $(A \sqcap B)\ x \Longrightarrow (A\ x \Longrightarrow B\ x \Longrightarrow P) \Longrightarrow P$

⟨*proof*⟩

lemma *inf2E*: $(A \sqcap B)\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *inf1D1*: $(A \sqcap B)\ x \Longrightarrow A\ x$
⟨*proof*⟩

lemma *inf2D1*: $(A \sqcap B)\ x\ y \Longrightarrow A\ x\ y$
⟨*proof*⟩

lemma *inf1D2*: $(A \sqcap B)\ x \Longrightarrow B\ x$
⟨*proof*⟩

lemma *inf2D2*: $(A \sqcap B)\ x\ y \Longrightarrow B\ x\ y$
⟨*proof*⟩

lemma *sup1I1*: $A\ x \Longrightarrow (A \sqcup B)\ x$
⟨*proof*⟩

lemma *sup2I1*: $A\ x\ y \Longrightarrow (A \sqcup B)\ x\ y$
⟨*proof*⟩

lemma *sup1I2*: $B\ x \Longrightarrow (A \sqcup B)\ x$
⟨*proof*⟩

lemma *sup2I2*: $B\ x\ y \Longrightarrow (A \sqcup B)\ x\ y$
⟨*proof*⟩

lemma *sup1E*: $(A \sqcup B)\ x \Longrightarrow (A\ x \Longrightarrow P) \Longrightarrow (B\ x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *sup2E*: $(A \sqcup B)\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow P) \Longrightarrow (B\ x\ y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI*: $(\neg\ B\ x \Longrightarrow A\ x) \Longrightarrow (A \sqcup B)\ x$
⟨*proof*⟩

lemma *sup2CI*: $(\neg\ B\ x\ y \Longrightarrow A\ x\ y) \Longrightarrow (A \sqcup B)\ x\ y$
⟨*proof*⟩

end

7 Set theory for higher-order logic

theory *Set*
imports *Lattices*

begin

7.1 Sets as predicates

typedecl $'a$ *set*

axiomatization *Collect* :: $('a \Rightarrow bool) \Rightarrow 'a\ set$ — comprehension
and *member* :: $'a \Rightarrow 'a\ set \Rightarrow bool$ — membership
where *mem-Collect-eq* [*iff*, *code-unfold*]: $member\ a\ (Collect\ P) = P\ a$
and *Collect-mem-eq* [*simp*]: $Collect\ (\lambda x.\ member\ x\ A) = A$

notation
member $(op\ \in)$ **and**
member $((\text{-}/ \in \text{-})\ [51, 51]\ 50)$

abbreviation *not-member*
where $not\text{-}member\ x\ A \equiv \neg\ (x \in A)$ — non-membership
notation
not-member $(op\ \notin)$ **and**
not-member $((\text{-}/ \notin \text{-})\ [51, 51]\ 50)$

notation (*ASCII*)
member $(op\ :)$ **and**
member $((\text{-}/ : \text{-})\ [51, 51]\ 50)$ **and**
not-member $(op\ {}^{\sim}:)$ **and**
not-member $((\text{-}/ {}^{\sim}: \text{-})\ [51, 51]\ 50)$

Set comprehensions

syntax
-Coll :: $pttrn \Rightarrow bool \Rightarrow 'a\ set$ $((1\{\text{-}./\ \text{-}\}))$
translations
$\{x.\ P\} \rightleftharpoons CONST\ Collect\ (\lambda x.\ P)$

syntax (*ASCII*)
-Collect :: $pttrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a\ set$ $((1\{\text{-}\ :/\ \text{-}./\ \text{-}\}))$
syntax
-Collect :: $pttrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a\ set$ $((1\{\text{-}\ \in/\ \text{-}./\ \text{-}\}))$
translations
$\{p{:}A.\ P\} \rightharpoonup CONST\ Collect\ (\lambda p.\ p \in A \wedge P)$

lemma *CollectI*: $P\ a \Longrightarrow a \in \{x.\ P\ x\}$
⟨*proof*⟩

lemma *CollectD*: $a \in \{x.\ P\ x\} \Longrightarrow P\ a$
⟨*proof*⟩

lemma *Collect-cong*: $(\bigwedge x.\ P\ x = Q\ x) \Longrightarrow \{x.\ P\ x\} = \{x.\ Q\ x\}$
⟨*proof*⟩

Simproc for pulling $x = t$ in $\{x.\ \ldots \wedge x = t \wedge \ldots\}$ to the front (and similarly for $t = x$):

⟨*ML*⟩

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

lemma *set-eqI*:
assumes $\bigwedge x.\ x \in A \longleftrightarrow x \in B$
shows $A = B$
⟨*proof*⟩

lemma *set-eq-iff*: $A = B \longleftrightarrow (\forall x.\ x \in A \longleftrightarrow x \in B)$
⟨*proof*⟩

lemma *Collect-eqI*:
assumes $\bigwedge x.\ P\ x = Q\ x$
shows *Collect* P = *Collect* Q
⟨*proof*⟩

Lifting of predicate class instances

instantiation *set* :: (*type*) *boolean-algebra*
begin

definition *less-eq-set*
where $A \le B \longleftrightarrow (\lambda x.\ member\ x\ A) \le (\lambda x.\ member\ x\ B)$

definition *less-set*
where $A < B \longleftrightarrow (\lambda x.\ member\ x\ A) < (\lambda x.\ member\ x\ B)$

definition *inf-set*
where $A \sqcap B = Collect\ ((\lambda x.\ member\ x\ A) \sqcap (\lambda x.\ member\ x\ B))$

definition *sup-set*
where $A \sqcup B = Collect\ ((\lambda x.\ member\ x\ A) \sqcup (\lambda x.\ member\ x\ B))$

definition *bot-set*
where $\bot = Collect\ \bot$

definition *top-set*
where $\top = Collect\ \top$

definition *uminus-set*
where $-\ A = Collect\ (-\ (\lambda x.\ member\ x\ A))$

definition *minus-set*
where $A - B = Collect\ ((\lambda x.\ member\ x\ A) - (\lambda x.\ member\ x\ B))$

instance
⟨*proof*⟩

end

Set enumerations

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

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

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

7.2 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
  subset (*op* ⊂) **and**
  subset ((-/ ⊂ -) [*51*, *51*] *50*) **and**
  subset-eq (*op* ⊆) **and**
  subset-eq ((-/ ⊆ -) [*51*, *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
  supset (*op* ⊃) **and**
  supset ((-/ ⊃ -) [*51*, *51*] *50*) **and**
  supset-eq (*op* ⊇) **and**
  supset-eq ((-/ ⊇ -) [*51*, *51*] *50*)

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

definition $Ball :: {}'a\ set \Rightarrow ({}'a \Rightarrow bool) \Rightarrow bool$
where $Ball\ A\ P \longleftrightarrow (\forall x.\ x \in A \longrightarrow P\ x)$ — bounded universal quantifiers

definition $Bex :: {}'a\ set \Rightarrow ({}'a \Rightarrow bool) \Rightarrow bool$
where $Bex\ A\ P \longleftrightarrow (\exists x.\ x \in A \land P\ x)$ — bounded existential quantifiers

syntax (*ASCII*)
-Ball :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3ALL* -:-./ -) [*0*, *0*, *10*] *10*)
-Bex :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3EX* -:-./ -) [*0*, *0*, *10*] *10*)
-Bex1 :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3EX*! -:-./ -) [*0*, *0*, *10*] *10*)
-Bleast :: $id \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow {}'a$ ((*3LEAST* -:-./ -) [*0*, *0*, *10*] *10*)

syntax (*input*)
-Ball :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3*! -:-./ -) [*0*, *0*, *10*] *10*)
-Bex :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3?* -:-./ -) [*0*, *0*, *10*] *10*)
-Bex1 :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3?*! -:-./ -) [*0*, *0*, *10*] *10*)

syntax
-Ball :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3*∀ -∈-./ -) [*0*, *0*, *10*] *10*)
-Bex :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3*∃ -∈-./ -) [*0*, *0*, *10*] *10*)
-Bex1 :: $pttrn \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow bool$ ((*3*∃!-∈-./ -) [*0*, *0*, *10*] *10*)
-Bleast :: $id \Rightarrow {}'a\ set \Rightarrow bool \Rightarrow {}'a$ ((*3LEAST*-∈-./ -) [*0*, *0*, *10*] *10*)

translations
$\forall x{\in}A.\ P \rightleftharpoons CONST\ Ball\ A\ (\lambda x.\ P)$
$\exists x{\in}A.\ P \rightleftharpoons CONST\ Bex\ A\ (\lambda x.\ P)$
$\exists!x{\in}A.\ P \rightharpoonup \exists!x.\ x \in A \land P$
$LEAST\ x{:}A.\ P \rightharpoonup LEAST\ x.\ x \in A \land P$

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

syntax
-setlessAll :: $[idt, {}'a, bool] \Rightarrow bool$ ((*3*∀ -⊂-./ -) [*0*, *0*, *10*] *10*)
-setlessEx :: $[idt, {}'a, bool] \Rightarrow bool$ ((*3*∃ -⊂-./ -) [*0*, *0*, *10*] *10*)
-setleAll :: $[idt, {}'a, bool] \Rightarrow bool$ ((*3*∀ -⊆-./ -) [*0*, *0*, *10*] *10*)
-setleEx :: $[idt, {}'a, bool] \Rightarrow bool$ ((*3*∃ -⊆-./ -) [*0*, *0*, *10*] *10*)
-setleEx1 :: $[idt, {}'a, bool] \Rightarrow bool$ ((*3*∃!-⊆-./ -) [*0*, *0*, *10*] *10*)

translations
$\forall A{\subset}B.\ P \rightharpoonup \forall A.\ A \subset B \longrightarrow P$
$\exists A{\subset}B.\ P \rightharpoonup \exists A.\ A \subset B \land P$
$\forall A{\subseteq}B.\ P \rightharpoonup \forall A.\ A \subseteq B \longrightarrow P$
$\exists A{\subseteq}B.\ P \rightharpoonup \exists A.\ A \subseteq B \land P$
$\exists!A{\subseteq}B.\ P \rightharpoonup \exists!A.\ A \subseteq B \land P$

⟨*ML*⟩

Translate between $\{e \mid x1\ldots xn.\ P\}$ and $\{u.\ \exists\, x1\ldots xn.\ u = e \wedge P\}$; $\{y.\ \exists\, x1\ldots xn.\ y = e \wedge P\}$ is only translated if $[0..n] \subseteq bvs\ e$.

syntax
-Setcompr :: $'a \Rightarrow idts \Rightarrow bool \Rightarrow 'a\ set$ ((*1*{- |/-./ -}))

⟨*ML*⟩

lemma *ballI* [*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall\, x{\in}A.\ P\ x$
⟨*proof*⟩

lemmas *strip* = *impI allI ballI*

lemma *bspec* [*dest?*]: $\forall\, x{\in}A.\ P\ x \Longrightarrow x \in A \Longrightarrow P\ x$
⟨*proof*⟩

Gives better instantiation for bound:

⟨*ML*⟩

lemma *ballE* [*elim*]: $\forall\, x{\in}A.\ P\ x \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *bexI* [*intro*]: $P\ x \Longrightarrow x \in A \Longrightarrow \exists\, x{\in}A.\ P\ x$
— Normally the best argument order: $P\ x$ constrains the choice of $x \in A$.
⟨*proof*⟩

lemma *rev-bexI* [*intro?*]: $x \in A \Longrightarrow P\ x \Longrightarrow \exists\, x{\in}A.\ P\ x$
— The best argument order when there is only one $x \in A$.
⟨*proof*⟩

lemma *bexCI*: $(\forall\, x{\in}A.\ \neg\ P\ x \Longrightarrow P\ a) \Longrightarrow a \in A \Longrightarrow \exists\, x{\in}A.\ P\ x$
⟨*proof*⟩

lemma *bexE* [*elim*!]: $\exists\, x{\in}A.\ P\ x \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow P\ x \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *ball-triv* [*simp*]: $(\forall\, x{\in}A.\ P) \longleftrightarrow ((\exists\, x.\ x \in A) \longrightarrow P)$
— Trival rewrite rule.
⟨*proof*⟩

lemma *bex-triv* [*simp*]: $(\exists\, x{\in}A.\ P) \longleftrightarrow ((\exists\, x.\ x \in A) \wedge P)$
— Dual form for existentials.
⟨*proof*⟩

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

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

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

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

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

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

lemma *ball-conj-distrib*: $(\forall x{\in}A.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall x{\in}A.\ P\ x) \wedge (\forall x{\in}A.\ Q\ x)$
⟨*proof*⟩

lemma *bex-disj-distrib*: $(\exists x{\in}A.\ P\ x \vee Q\ x) \longleftrightarrow (\exists x{\in}A.\ P\ x) \vee (\exists x{\in}A.\ Q\ x)$
⟨*proof*⟩

Congruence rules

lemma *ball-cong*:
$A = B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow P\ x \longleftrightarrow Q\ x) \Longrightarrow$
$(\forall x{\in}A.\ P\ x) \longleftrightarrow (\forall x{\in}B.\ Q\ x)$
⟨*proof*⟩

lemma *strong-ball-cong* [*cong*]:
$A = B \Longrightarrow (\bigwedge x.\ x \in B =simp=> P\ x \longleftrightarrow Q\ x) \Longrightarrow$
$(\forall x{\in}A.\ P\ x) \longleftrightarrow (\forall x{\in}B.\ Q\ x)$
⟨*proof*⟩

lemma *bex-cong*:
$A = B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow P\ x \longleftrightarrow Q\ x) \Longrightarrow$
$(\exists x{\in}A.\ P\ x) \longleftrightarrow (\exists x{\in}B.\ Q\ x)$
⟨*proof*⟩

lemma *strong-bex-cong* [*cong*]:
$A = B \Longrightarrow (\bigwedge x.\ x \in B =simp=> P\ x \longleftrightarrow Q\ x) \Longrightarrow$
$(\exists x{\in}A.\ P\ x) \longleftrightarrow (\exists x{\in}B.\ Q\ x)$
⟨*proof*⟩

lemma *bex1-def*: $(\exists!x{\in}X.\ P\ x) \longleftrightarrow (\exists x{\in}X.\ P\ x) \wedge (\forall x{\in}X.\ \forall y{\in}X.\ P\ x \longrightarrow P\ y \longrightarrow x = y)$
⟨*proof*⟩

7.3 Basic operations

7.3.1 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?*]: $A \subseteq B \Longrightarrow c \in A \Longrightarrow c \in B$
⟨*proof*⟩

lemma *rev-subsetD* [*intro?*]: $c \in A \Longrightarrow A \subseteq B \Longrightarrow c \in B$
— The same, with reversed premises for use with *erule* – cf. $[\![?P;\ ?P \longrightarrow ?Q]\!] \Longrightarrow ?Q$.
⟨*proof*⟩

lemma *subsetCE* [*elim*]: $A \subseteq B \Longrightarrow (c \notin A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$
— Classical elimination rule.
⟨*proof*⟩

lemma *subset-eq*: $A \subseteq B \longleftrightarrow (\forall x{\in}A.\ x \in B)$
⟨*proof*⟩

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

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

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

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

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

lemma *subset-not-subset-eq* [*code*]: $A \subset B \longleftrightarrow A \subseteq B \land \neg B \subseteq A$
⟨*proof*⟩

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

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

7.3.2 Equality

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

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

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

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

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

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

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

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

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

7.3.3 The empty set

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

lemma *empty-iff* [*simp*]: $c \in \{\} \longleftrightarrow \mathit{False}$
⟨*proof*⟩

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

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

lemma *equals0I*: $(\bigwedge y.\ y \in A \Longrightarrow \mathit{False}) \Longrightarrow A = \{\}$
⟨*proof*⟩

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

⟨*proof*⟩

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

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

7.3.4 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?*]: ∃ *x*. *x* ∈ *UNIV*
⟨*proof*⟩

lemma *subset-UNIV*: *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*⟩

lemma *UNIV-not-empty* [*iff*]: *UNIV* ≠ {}
⟨*proof*⟩

lemma *empty-not-UNIV*[*simp*]: {} ≠ *UNIV*
⟨*proof*⟩

7.3.5 The Powerset operator – Pow

definition *Pow* :: *'a set* ⇒ *'a set set*
where *Pow-def*: *Pow A* = {*B*. *B* ⊆ *A*}

lemma *Pow-iff* [*iff*]: $A \in Pow\ B \longleftrightarrow A \subseteq B$
⟨*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*⟩

lemma *Pow-not-empty*: $Pow\ A \neq \{\}$
⟨*proof*⟩

7.3.6 Set complement

lemma *Compl-iff* [*simp*]: $c \in -\ A \longleftrightarrow c \notin A$
⟨*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*!]: $c \in -\ A \Longrightarrow c \notin A$
⟨*proof*⟩

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

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

7.3.7 Binary intersection

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

notation (*ASCII*)
inter (**infixl** *Int 70*)

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

lemma *Int-iff* [*simp*]: $c \in A \cap B \longleftrightarrow c \in A \land c \in B$

⟨*proof*⟩

lemma *IntI* [*intro*!]: $c \in A \Longrightarrow c \in B \Longrightarrow c \in A \cap B$
⟨*proof*⟩

lemma *IntD1*: $c \in A \cap B \Longrightarrow c \in A$
⟨*proof*⟩

lemma *IntD2*: $c \in A \cap B \Longrightarrow c \in B$
⟨*proof*⟩

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

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

7.3.8 Binary union

abbreviation *union* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ set$ (**infixl** $\cup$ *65*)
where $union \equiv sup$

notation (*ASCII*)
union (**infixl** *Un 65*)

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

lemma *Un-iff* [*simp*]: $c \in A \cup B \longleftrightarrow c \in A \lor c \in B$
⟨*proof*⟩

lemma *UnI1* [*elim?*]: $c \in A \Longrightarrow c \in A \cup B$
⟨*proof*⟩

lemma *UnI2* [*elim?*]: $c \in B \Longrightarrow c \in A \cup B$
⟨*proof*⟩

Classical introduction rule: no commitment to A vs. B.

lemma *UnCI* [*intro*!]: $(c \notin B \Longrightarrow c \in A) \Longrightarrow c \in A \cup B$
⟨*proof*⟩

lemma *UnE* [*elim*!]: $c \in A \cup B \Longrightarrow (c \in A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$
⟨*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*⟩

7.3.9 Set difference

lemma *Diff-iff* [*simp*]: $c \in A - B \longleftrightarrow c \in A \wedge c \notin B$
⟨*proof*⟩

lemma *DiffI* [*intro*!]: $c \in A \Longrightarrow c \notin B \Longrightarrow c \in A - B$
⟨*proof*⟩

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

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

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

lemma *set-diff-eq*: $A - B = \{x.\ x \in A \wedge x \notin B\}$
⟨*proof*⟩

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

7.3.10 Augmenting a set − *insert*

lemma *insert-iff* [*simp*]: $a \in insert\ b\ A \longleftrightarrow a = b \vee a \in A$
⟨*proof*⟩

lemma *insertI1*: $a \in insert\ a\ B$
⟨*proof*⟩

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

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

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

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

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

lemma *insert-ident*: $x \notin A \Longrightarrow x \notin B \Longrightarrow insert\ x\ A = insert\ x\ B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *insert-eq-iff*:
assumes $a \notin A\ b \notin B$
shows $insert\ a\ A = insert\ b\ B \longleftrightarrow$
$(if\ a = b\ then\ A = B\ else\ \exists\, C.\ A = insert\ b\ C \wedge b \notin C \wedge B = insert\ a\ C \wedge a \notin C)$
(**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *insert-UNIV*: $insert\ x\ UNIV = UNIV$
⟨*proof*⟩

7.3.11 Singletons, using insert

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

lemma *singletonD* [*dest*!]: $b \in \{a\} \Longrightarrow b = a$
⟨*proof*⟩

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

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

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

lemma *singleton-insert-inj-eq* [*iff*]: $\{b\} = insert\ a\ A \longleftrightarrow a = b \wedge A \subseteq \{b\}$
⟨*proof*⟩

lemma *singleton-insert-inj-eq'* [*iff*]: $insert\ a\ A = \{b\} \longleftrightarrow a = b \wedge A \subseteq \{b\}$
⟨*proof*⟩

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

lemma *subset-singleton-iff*: $X \subseteq \{a\} \longleftrightarrow X = \{\} \vee X = \{a\}$
⟨*proof*⟩

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

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

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

lemma *subset-Diff-insert*: $A \subseteq B - insert\ x\ C \longleftrightarrow A \subseteq B - C \wedge x \notin A$
⟨*proof*⟩

lemma *doubleton-eq-iff*: $\{a, b\} = \{c, d\} \longleftrightarrow a = c \wedge b = d \vee a = d \ \&\ b = c$
⟨*proof*⟩

lemma *Un-singleton-iff*: $A \cup B = \{x\} \longleftrightarrow A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
⟨*proof*⟩

lemma *singleton-Un-iff*: $\{x\} = A \cup B \longleftrightarrow A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
⟨*proof*⟩

7.3.12 Image of a set under a function

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

definition *image* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set$ (**infixr** *‘ 90*)
where $f\ ‘\ A = \{y.\ \exists x \in A.\ y = f\ x\}$

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

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

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

lemma *imageE* [*elim*!]:
assumes $b \in (\lambda x.\ f\ x)\ ‘\ A$ — The eta-expansion gives variable-name preservation.

obtains x **where** $b = f\ x$ **and** $x \in A$
⟨*proof*⟩

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

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

lemma *image-iff*: $z \in f\ ‘\ A \longleftrightarrow (\exists x \in A.\ z = f\ x)$
⟨*proof*⟩

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

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

lemma *subset-imageE*:
assumes $B \subseteq f\ `\ A$
obtains C **where** $C \subseteq A$ **and** $B = f\ `\ C$
⟨*proof*⟩

lemma *subset-image-iff*: $B \subseteq f\ `\ A \longleftrightarrow (\exists\,AA{\subseteq}A.\ B = f\ `\ AA)$
⟨*proof*⟩

lemma *image-ident* [*simp*]: $(\lambda x.\ x)\ `\ Y = Y$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *image-Collect*: $f\ `\ \{x.\ P\ x\} = \{f\,x \mid x.\ P\ x\}$
— NOT suitable as a default simp rule: 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*: $M = N \Longrightarrow (\bigwedge x.\ x \in N \Longrightarrow f\ x = g\ x) \Longrightarrow f$ ` $M = g$ ` N
⟨*proof*⟩

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

lemma *image-diff-subset*: f ` $A - f$ ` $B \subseteq f$ ` $(A - B)$
⟨*proof*⟩

lemma *Setcompr-eq-image*: $\{f\ x\ |x.\ x \in A\} = f$ ` A
⟨*proof*⟩

lemma *setcompr-eq-image*: $\{f\ x\ |x.\ P\ x\} = f$ ` $\{x.\ P\ x\}$
⟨*proof*⟩

lemma *ball-imageD*: $\forall\, x \in f$ ` $A.\ P\ x \Longrightarrow \forall\, x \in A.\ P\ (f\ x)$
⟨*proof*⟩

lemma *bex-imageD*: $\exists\, x \in f$ ` $A.\ P\ x \Longrightarrow \exists\, x \in A.\ P\ (f\ x)$
⟨*proof*⟩

lemma *image-add-0* [*simp*]: $op + (0{::}'a{::}\mathit{comm\text{-}monoid\text{-}add})$ ` $S = S$
⟨*proof*⟩

Range of a function – just an abbreviation for image!

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

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

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

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

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

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

lemma *range-eq-singletonD*: $\mathit{range}\ f = \{a\} \Longrightarrow f\ x = a$

⟨*proof*⟩

7.3.13 Some rules with *if*

Elimination of $\{x.\ \dots\ \wedge\ x = t\ \wedge\ \dots\}$.

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

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

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

lemma *if-split-eq1*: $(\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) = b \longleftrightarrow (Q \longrightarrow x = b) \wedge (\neg\ Q \longrightarrow y = b)$
⟨*proof*⟩

lemma *if-split-eq2*: $a = (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) \longleftrightarrow (Q \longrightarrow a = x) \wedge (\neg\ Q \longrightarrow a = y)$
⟨*proof*⟩

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

lemma *if-split-mem1*: $(\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) \in b \longleftrightarrow (Q \longrightarrow x \in b) \wedge (\neg\ Q \longrightarrow y \in b)$
⟨*proof*⟩

lemma *if-split-mem2*: $(a \in (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y)) \longleftrightarrow (Q \longrightarrow a \in x) \wedge (\neg\ Q \longrightarrow a \in y)$
⟨*proof*⟩

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

7.4 Further operations and lemmas

7.4.1 The "proper subset" relation

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

lemma *psubsetE* [*elim*!]: $A \subset B \Longrightarrow (A \subseteq B \Longrightarrow \neg\ B \subseteq A \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

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

lemma *psubset-eq*: $A \subset B \longleftrightarrow A \subseteq B \wedge A \neq B$
⟨*proof*⟩

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

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

lemma *psubsetD*: $A \subset B \Longrightarrow c \in A \Longrightarrow c \in B$
⟨*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*: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \equiv \mathit{Trueprop}\ (\forall\, x{\in}A.\ P\ x)$
⟨*proof*⟩

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

lemma *image-Pow-mono*: $f\ `\ A \subseteq B \Longrightarrow \mathit{image}\ f\ `\ \mathit{Pow}\ A \subseteq \mathit{Pow}\ B$
⟨*proof*⟩

lemma *image-Pow-surj*: $f\ `\ A = B \Longrightarrow \mathit{image}\ f\ `\ \mathit{Pow}\ A = \mathit{Pow}\ B$
⟨*proof*⟩

7.4.2 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*: $x \notin A \Longrightarrow A \subseteq \mathit{insert}\ x\ B \longleftrightarrow A \subseteq B$
⟨*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*: $A - B \subseteq C \longleftrightarrow A \subseteq B \cup C$
⟨*proof*⟩

7.4.3 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*]: $A \subseteq \{\} \longleftrightarrow A = \{\}$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *Collect-mono-iff*: $Collect\ P \subseteq Collect\ Q \longleftrightarrow (\forall x.\ P\ x \longrightarrow Q\ x)$
⟨*proof*⟩

insert.

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

lemma *insert-not-empty* [*simp*]: $insert\ a\ A \neq \{\}$
and *empty-not-insert* [*simp*]: $\{\} \neq insert\ a\ A$
⟨*proof*⟩

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

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

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

lemma *insert-subset* [*simp*]: $insert\ x\ A \subseteq B \longleftrightarrow x \in B \wedge A \subseteq B$
⟨*proof*⟩

lemma *mk-disjoint-insert*: $a \in A \Longrightarrow \exists B.\ A = insert\ a\ B \wedge a \notin B$
— 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*]:
$insert\ a\ A \cap B = \{\} \longleftrightarrow a \notin B \wedge A \cap B = \{\}$
$\{\} = insert\ a\ A \cap B \longleftrightarrow a \notin B \wedge \{\} = A \cap B$
⟨*proof*⟩

lemma *disjoint-insert* [*simp*]:
$B \cap insert\ a\ A = \{\} \longleftrightarrow a \notin B \wedge B \cap A = \{\}$

$\{\} = A \cap \mathit{insert}\ b\ B \longleftrightarrow b \notin A \wedge \{\} = A \cap B$
⟨*proof*⟩

Int

lemma *Int-absorb*: $A \cap A = A$
⟨*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*: $\{\} \cap B = \{\}$
⟨*proof*⟩

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

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

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

lemma *Int-UNIV-left*: $\mathit{UNIV} \cap B = B$
⟨*proof*⟩

lemma *Int-UNIV-right*: $A \cap \mathit{UNIV} = A$
⟨*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*]: $A \cap B = UNIV \longleftrightarrow A = UNIV \wedge B = UNIV$
⟨*proof*⟩

lemma *Int-subset-iff* [*simp*]: $C \subseteq A \cap B \longleftrightarrow C \subseteq A \wedge C \subseteq B$
⟨*proof*⟩

lemma *Int-Collect*: $x \in A \cap \{x.\ P\ x\} \longleftrightarrow x \in A \wedge P\ x$
⟨*proof*⟩

Un.

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

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

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

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

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

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

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

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

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

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

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

lemma *Un-UNIV-right*: $A \cup UNIV = UNIV$
⟨*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*: $(insert\ a\ B) \cap C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
⟨*proof*⟩

lemma *Int-insert-left-if0* [*simp*]: $a \notin C \Longrightarrow (insert\ a\ B) \cap C = B \cap C$
⟨*proof*⟩

lemma *Int-insert-left-if1* [*simp*]: $a \in C \Longrightarrow (insert\ a\ B) \cap C = insert\ a\ (B \cap C)$
⟨*proof*⟩

lemma *Int-insert-right*: $A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
⟨*proof*⟩

lemma *Int-insert-right-if0* [*simp*]: $a \notin A \Longrightarrow A \cap (insert\ a\ B) = A \cap B$
⟨*proof*⟩

lemma *Int-insert-right-if1* [*simp*]: $a \in A \Longrightarrow A \cap (insert\ a\ B) = insert\ a\ (A \cap B)$
⟨*proof*⟩

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

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

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

lemma *subset-Un-eq*: $A \subseteq B \longleftrightarrow A \cup B = B$
⟨*proof*⟩

lemma *Un-empty* [*iff*]: $A \cup B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
⟨*proof*⟩

lemma *Un-subset-iff* [*simp*]: $A \cup B \subseteq C \longleftrightarrow A \subseteq C \wedge B \subseteq C$
⟨*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*: $- (-A) = A$ **for** $A :: 'a\ set$
⟨*proof*⟩

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

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

lemma *subset-Compl-self-eq*: $A \subseteq - A \longleftrightarrow A = \{\}$
⟨*proof*⟩

lemma *Un-Int-assoc-eq*: $(A \cap B) \cup C = A \cap (B \cup C) \longleftrightarrow C \subseteq A$
— Halmos, Naive Set Theory, page 16.
⟨*proof*⟩

lemma *Compl-UNIV-eq*: $- UNIV = \{\}$
⟨*proof*⟩

lemma *Compl-empty-eq*: $- \{\} = UNIV$
⟨*proof*⟩

lemma *Compl-subset-Compl-iff* [*iff*]: $- A \subseteq - B \longleftrightarrow B \subseteq A$
⟨*proof*⟩

lemma *Compl-eq-Compl-iff* [*iff*]: $- A = - B \longleftrightarrow A = B$
for $A\ B :: 'a\ set$
⟨*proof*⟩

lemma *Compl-insert*: $- insert\ x\ A = (- A) - \{x\}$
⟨*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*: $(\forall x \in A \cup B.\ P\ x) \longleftrightarrow (\forall x{\in}A.\ P\ x) \wedge (\forall x{\in}B.\ P\ x)$
⟨*proof*⟩

lemma *bex-Un*: $(\exists x \in A \cup B.\ P\ x) \longleftrightarrow (\exists x{\in}A.\ P\ x) \vee (\exists x{\in}B.\ P\ x)$
⟨*proof*⟩

Set difference.

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

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

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

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - B$
for $A\ B$ :: *'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*]: $A - \mathit{UNIV} = \{\}$
⟨*proof*⟩

lemma *Diff-insert0* [*simp*]: $x \notin A \Longrightarrow A - \mathit{insert}\ x\ B = A - B$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *double-diff*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow B - (C - A) = A$
⟨*proof*⟩

lemma *Un-Diff-cancel* [*simp*]: $A \cup (B - A) = A \cup B$
⟨*proof*⟩

lemma *Un-Diff-cancel2* [*simp*]: $(B - A) \cup A = B \cup A$
⟨*proof*⟩

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

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

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

lemma *subset-Compl-singleton* [*simp*]: $A \subseteq - \{b\} \longleftrightarrow b \notin A$
⟨*proof*⟩

Quantification over type *bool*.

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

lemma *all-bool-eq*: $(\forall b.\ P\ b) \longleftrightarrow P\ True \wedge P\ False$
⟨*proof*⟩

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

lemma *ex-bool-eq*: $(\exists b.\ P\ b) \longleftrightarrow P\ True \vee P\ False$
⟨*proof*⟩

lemma *UNIV-bool*: $UNIV = \{False,\ True\}$
⟨*proof*⟩

Pow

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

lemma *Pow-singleton-iff* [*simp*]: $Pow\ X = \{Y\} \longleftrightarrow X = \{\} \wedge Y = \{\}$
⟨*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*: $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
$\langle proof \rangle$

lemma *subset-iff*: $A \subseteq B \longleftrightarrow (\forall t.\ t \in A \longrightarrow t \in B)$
$\langle proof \rangle$

lemma *subset-iff-psubset-eq*: $A \subseteq B \longleftrightarrow A \subset B \vee A = B$
$\langle proof \rangle$

lemma *all-not-in-conv* [*simp*]: $(\forall x.\ x \notin A) \longleftrightarrow A = \{\}$
$\langle proof \rangle$

lemma *ex-in-conv*: $(\exists x.\ x \in A) \longleftrightarrow A \neq \{\}$
$\langle proof \rangle$

lemma *ball-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P\ x \vee Q) \longleftrightarrow ((\forall x{\in}A.\ P\ x) \vee Q)$
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P \vee Q\ x) \longleftrightarrow (P \vee (\forall x{\in}A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P \longrightarrow Q\ x) \longleftrightarrow (P \longrightarrow (\forall x{\in}A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P\ x \longrightarrow Q) \longleftrightarrow ((\exists x{\in}A.\ P\ x) \longrightarrow Q)$
$\bigwedge P.\ (\forall x{\in}\{\}.\ P\ x) \longleftrightarrow \mathit{True}$
$\bigwedge P.\ (\forall x{\in}\mathit{UNIV}.\ P\ x) \longleftrightarrow (\forall x.\ P\ x)$
$\bigwedge a\ B\ P.\ (\forall x{\in}\mathit{insert}\ a\ B.\ P\ x) \longleftrightarrow (P\ a \wedge (\forall x{\in}B.\ P\ x))$
$\bigwedge P\ Q.\ (\forall x{\in}\mathit{Collect}\ Q.\ P\ x) \longleftrightarrow (\forall x.\ Q\ x \longrightarrow P\ x)$
$\bigwedge A\ P\ f.\ (\forall x{\in}f`A.\ P\ x) \longleftrightarrow (\forall x{\in}A.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg\ (\forall x{\in}A.\ P\ x)) \longleftrightarrow (\exists x{\in}A.\ \neg\ P\ x)$
$\langle proof \rangle$

lemma *bex-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ (\exists x{\in}A.\ P\ x \wedge Q) \longleftrightarrow ((\exists x{\in}A.\ P\ x) \wedge Q)$
$\bigwedge A\ P\ Q.\ (\exists x{\in}A.\ P \wedge Q\ x) \longleftrightarrow (P \wedge (\exists x{\in}A.\ Q\ x))$
$\bigwedge P.\ (\exists x{\in}\{\}.\ P\ x) \longleftrightarrow \mathit{False}$
$\bigwedge P.\ (\exists x{\in}\mathit{UNIV}.\ P\ x) \longleftrightarrow (\exists x.\ P\ x)$
$\bigwedge a\ B\ P.\ (\exists x{\in}\mathit{insert}\ a\ B.\ P\ x) \longleftrightarrow (P\ a \mid (\exists x{\in}B.\ P\ x))$
$\bigwedge P\ Q.\ (\exists x{\in}\mathit{Collect}\ Q.\ P\ x) \longleftrightarrow (\exists x.\ Q\ x \wedge P\ x)$
$\bigwedge A\ P\ f.\ (\exists x{\in}f`A.\ P\ x) \longleftrightarrow (\exists x{\in}A.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg(\exists x{\in}A.\ P\ x)) \longleftrightarrow (\forall x{\in}A.\ \neg\ P\ x)$
$\langle proof \rangle$

7.4.4 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \Longrightarrow f\ `\ A \subseteq f\ `\ B$
$\langle proof \rangle$

lemma *Pow-mono*: $A \subseteq B \Longrightarrow \mathit{Pow}\ A \subseteq \mathit{Pow}\ B$
$\langle proof \rangle$

lemma *insert-mono*: $C \subseteq D \Longrightarrow \mathit{insert}\ a\ C \subseteq \mathit{insert}\ a\ D$
$\langle proof \rangle$

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*: $P1 \longrightarrow Q1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1 \wedge P2) \longrightarrow (Q1 \wedge Q2)$
  ⟨*proof*⟩

lemma *disj-mono*: $P1 \longrightarrow Q1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1 \vee P2) \longrightarrow (Q1 \vee Q2)$
  ⟨*proof*⟩

lemma *imp-mono*: $Q1 \longrightarrow P1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2)$
  ⟨*proof*⟩

lemma *imp-refl*: $P \longrightarrow P$ ⟨*proof*⟩

lemma *not-mono*: $Q \longrightarrow P \Longrightarrow \neg P \longrightarrow \neg Q$
  ⟨*proof*⟩

lemma *ex-mono*: $(\bigwedge x.\ P\ x \longrightarrow Q\ x) \Longrightarrow (\exists x.\ P\ x) \longrightarrow (\exists x.\ Q\ x)$
  ⟨*proof*⟩

lemma *all-mono*: $(\bigwedge x.\ P\ x \longrightarrow Q\ x) \Longrightarrow (\forall x.\ P\ x) \longrightarrow (\forall x.\ Q\ x)$
  ⟨*proof*⟩

lemma *Collect-mono*: $(\bigwedge x.\ P\ x \longrightarrow Q\ x) \Longrightarrow \mathit{Collect}\ P \subseteq \mathit{Collect}\ Q$
  ⟨*proof*⟩

lemma *Int-Collect-mono*: $A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow P\ x \longrightarrow Q\ x) \Longrightarrow A \cap \mathit{Collect}\ P \subseteq B \cap \mathit{Collect}\ Q$
  ⟨*proof*⟩

lemmas *basic-monos* =
  subset-refl imp-refl disj-mono conj-mono ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b \Longrightarrow c = d \Longrightarrow b \longrightarrow d \Longrightarrow a \longrightarrow c$

⟨*proof*⟩

7.4.5 Inverse image of a function

definition *vimage* :: $('a \Rightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a\ set$ (**infixr** $-`$ *90*)
where $f\ -`\ B \equiv \{x.\ f\ x \in B\}$

lemma *vimage-eq* [*simp*]: $a \in f\ -`\ B \longleftrightarrow f\ a \in B$
⟨*proof*⟩

lemma *vimage-singleton-eq*: $a \in f\ -`\ \{b\} \longleftrightarrow f\ a = b$
⟨*proof*⟩

lemma *vimageI* [*intro*]: $f\ a = b \Longrightarrow b \in B \Longrightarrow a \in f\ -`\ B$
⟨*proof*⟩

lemma *vimageI2*: $f\ a \in A \Longrightarrow a \in f\ -`\ A$
⟨*proof*⟩

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

lemma *vimageD*: $a \in f\ -`\ A \Longrightarrow f\ a \in A$
⟨*proof*⟩

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

lemma *vimage-Compl*: $f\ -`\ (-\ A) = -\ (f\ -`\ A)$
⟨*proof*⟩

lemma *vimage-Un* [*simp*]: $f\ -`\ (A \cup B) = (f\ -`\ A) \cup (f\ -`\ B)$
⟨*proof*⟩

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

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

lemma *vimage-Collect*: $(\bigwedge x.\ P\ (f\ x) = Q\ x) \Longrightarrow f\ -`\ (Collect\ P) = Collect\ Q$
⟨*proof*⟩

lemma *vimage-insert*: $f\ -`\ (insert\ a\ B) = (f\ -`\ \{a\}) \cup (f\ -`\ B)$
— 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*]: $f -` \mathit{UNIV} = \mathit{UNIV}$
  ⟨*proof*⟩

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

lemma *vimage-image-eq*: $f -` (f ` A) = \{y. \exists x{\in}A. f\, x = f\, y\}$
  ⟨*proof*⟩

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

lemma *image-vimage-eq* [*simp*]: $f ` (f -` A) = A \cap \mathit{range}\, f$
  ⟨*proof*⟩

lemma *image-subset-iff-subset-vimage*: $f ` A \subseteq B \longleftrightarrow A \subseteq f -` B$
  ⟨*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*]: $((\lambda x.\ \mathit{if}\ x \in B\ \mathit{then}\ c\ \mathit{else}\ d) -` A) =$
  $(\mathit{if}\ c \in A\ \mathit{then}\ (\mathit{if}\ d \in A\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ B)$
  $\mathit{else}\ \mathit{if}\ d \in A\ \mathit{then}\ -\ B\ \mathit{else}\ \{\})$
  ⟨*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 *vimage-ident* [*simp*]: $(\lambda x.\ x) -` Y = Y$
  ⟨*proof*⟩

7.4.6 Singleton sets

definition *is-singleton* :: $'a\ \mathit{set} \Rightarrow \mathit{bool}$
  where *is-singleton* $A \longleftrightarrow (\exists x.\ A = \{x\})$

lemma *is-singletonI* [*simp*, *intro*!]: *is-singleton* $\{x\}$
  ⟨*proof*⟩

lemma *is-singletonI'*: $A \neq \{\} \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x = y) \Longrightarrow$ *is-singleton* A
  ⟨*proof*⟩

lemma *is-singletonE*: *is-singleton* $A \Longrightarrow (\bigwedge x.\ A = \{x\} \Longrightarrow P) \Longrightarrow P$
  ⟨*proof*⟩

7.4.7 Getting the contents of a singleton set

definition *the-elem* :: $'a\ set \Rightarrow 'a$
where *the-elem* $X = (THE\ x.\ X = \{x\})$

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

lemma *is-singleton-the-elem*: *is-singleton* $A \longleftrightarrow A = \{$*the-elem* $A\}$
⟨*proof*⟩

lemma *the-elem-image-unique*:
assumes $A \neq \{\}$
and $*$: $\bigwedge y.\ y \in A \Longrightarrow f\ y = f\ x$
shows *the-elem* $(f\ \text{`}\ A) = f\ x$
⟨*proof*⟩

7.4.8 Least value operator

lemma *Least-mono*: *mono* $f \Longrightarrow \exists\, x{\in}S.\ \forall\, y{\in}S.\ x \leq y \Longrightarrow (LEAST\ y.\ y \in f\ \text{`}\ S) = f\ (LEAST\ x.\ x \in S)$
for f :: $'a{::}order \Rightarrow 'b{::}order$
— Courtesy of Stephan Merz
⟨*proof*⟩

7.4.9 Monad operation

definition *bind* :: $'a\ set \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'b\ set$
where *bind* $A\ f = \{x.\ \exists\, B \in f\text{`}A.\ x \in B\}$

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

lemma *bind-bind*: $Set.bind\ (Set.bind\ A\ B)\ C = Set.bind\ A\ (\lambda x.\ Set.bind\ (B\ x)\ C)$
for A :: $'a\ set$
⟨*proof*⟩

lemma *empty-bind* [*simp*]: $Set.bind\ \{\}\ f = \{\}$
⟨*proof*⟩

lemma *nonempty-bind-const*: $A \neq \{\} \Longrightarrow Set.bind\ A\ (\lambda\text{-}.\ B) = B$
⟨*proof*⟩

lemma *bind-const*: $Set.bind\ A\ (\lambda\text{-}.\ B) = (if\ A = \{\}\ then\ \{\}\ else\ B)$
⟨*proof*⟩

lemma *bind-singleton-conv-image*: $Set.bind\ A\ (\lambda x.\ \{f\ x\}) = f\ \text{`}\ A$
⟨*proof*⟩

7.4.10 Operations for execution

definition *is-empty* :: $'a\ set \Rightarrow bool$
where [*code-abbrev*]: *is-empty* $A \longleftrightarrow A = \{\}$

hide-const (**open**) *is-empty*

definition *remove* :: $'a \Rightarrow 'a\ set \Rightarrow 'a\ set$
where [*code-abbrev*]: *remove* $x\ A = A - \{x\}$

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

lemma *member-remove* [*simp*]: $x \in Set.remove\ y\ A \longleftrightarrow x \in A \wedge x \neq y$
⟨*proof*⟩

definition *filter* :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'a\ set$
where [*code-abbrev*]: *filter* $P\ A = \{a \in A.\ P\ a\}$

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

lemma *member-filter* [*simp*]: $x \in Set.filter\ P\ A \longleftrightarrow x \in A \wedge P\ x$
⟨*proof*⟩

instantiation *set* :: (*equal*) *equal*
begin

definition $HOL.equal\ A\ B \longleftrightarrow A \subseteq B \wedge B \subseteq A$

instance ⟨*proof*⟩

end

Misc

definition *pairwise* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$
where *pairwise* $R\ S \longleftrightarrow (\forall x \in S.\ \forall y \in S.\ x \neq y \longrightarrow R\ x\ y)$

lemma *pairwise-subset*: *pairwise* $P\ S \Longrightarrow T \subseteq S \Longrightarrow$ *pairwise* $P\ T$
⟨*proof*⟩

lemma *pairwise-mono*: $[\![pairwise\ P\ A;\ \bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y]\!] \Longrightarrow pairwise\ Q\ A$
⟨*proof*⟩

definition *disjnt* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$
where *disjnt* $A\ B \longleftrightarrow A \cap B = \{\}$

lemma *disjnt-self-iff-empty* [*simp*]: *disjnt* $S\ S \longleftrightarrow S = \{\}$
⟨*proof*⟩

lemma *disjnt-iff*: *disjnt* $A\ B \longleftrightarrow (\forall x.\ \neg\ (x \in A \wedge x \in B))$
⟨*proof*⟩

lemma *disjnt-sym*: $disjnt\ A\ B \Longrightarrow disjnt\ B\ A$
⟨*proof*⟩

lemma *disjnt-empty1* [*simp*]: $disjnt\ \{\}\ A$ **and** *disjnt-empty2* [*simp*]: $disjnt\ A\ \{\}$
⟨*proof*⟩

lemma *disjnt-insert1* [*simp*]: $disjnt\ (insert\ a\ X)\ Y \longleftrightarrow a \notin Y \land disjnt\ X\ Y$
⟨*proof*⟩

lemma *disjnt-insert2* [*simp*]: $disjnt\ Y\ (insert\ a\ X) \longleftrightarrow a \notin Y \land disjnt\ Y\ X$
⟨*proof*⟩

lemma *disjnt-subset1* : $[\![disjnt\ X\ Y;\ Z \subseteq X]\!] \Longrightarrow disjnt\ Z\ Y$
⟨*proof*⟩

lemma *disjnt-subset2* : $[\![disjnt\ X\ Y;\ Z \subseteq Y]\!] \Longrightarrow disjnt\ X\ Z$
⟨*proof*⟩

lemma *pairwise-empty* [*simp*]: $pairwise\ P\ \{\}$
⟨*proof*⟩

lemma *pairwise-singleton* [*simp*]: $pairwise\ P\ \{A\}$
⟨*proof*⟩

lemma *pairwise-insert*:
$pairwise\ r\ (insert\ x\ s) \longleftrightarrow (\forall y.\ y \in s \land y \neq x \longrightarrow r\ x\ y \land r\ y\ x) \land pairwise\ r\ s$
⟨*proof*⟩

lemma *pairwise-image*: $pairwise\ r\ (f\ `\ s) \longleftrightarrow pairwise\ (\lambda x\ y.\ (f\ x \neq f\ y) \longrightarrow r\ (f\ x)\ (f\ y))\ s$
⟨*proof*⟩

lemma *disjoint-image-subset*: $[\![pairwise\ disjnt\ \mathcal{A};\ \bigwedge X.\ X \in \mathcal{A} \Longrightarrow f\ X \subseteq X]\!] \Longrightarrow pairwise\ disjnt\ (f\ `\mathcal{A})$
⟨*proof*⟩

lemma *Int-emptyI*: $(\bigwedge x.\ x \in A \Longrightarrow x \in B \Longrightarrow False) \Longrightarrow A \cap B = \{\}$
⟨*proof*⟩

lemma *in-image-insert-iff* :
assumes $\bigwedge C.\ C \in B \Longrightarrow x \notin C$
shows $A \in insert\ x\ `\ B \longleftrightarrow x \in A \land A - \{x\} \in B$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

hide-const (**open**) *member not-member*

lemmas *equalityI* = *subset-antisym*

⟨*ML*⟩

end

8 HOL type definitions

theory *Typedef*
imports *Set*
keywords
  typedef :: *thy-goal* **and**
  morphisms :: *quasi-command*
begin

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

lemma *Rep-inject*: $Rep\ x = Rep\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *Abs-inject*:
  assumes $x \in A$ **and** $y \in A$
  shows $Abs\ x = Abs\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *Rep-cases* [*cases set*]:
  assumes $y \in A$
    and *hyp*: $\bigwedge x.\ y = Rep\ x \Longrightarrow P$
  shows P
⟨*proof*⟩

lemma *Abs-cases* [*cases type*]:
  assumes *r*: $\bigwedge y.\ x = Abs\ y \Longrightarrow y \in A \Longrightarrow P$
  shows P
⟨*proof*⟩

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

lemma *Abs-induct* [*induct type*]:
  assumes *r*: $\bigwedge y.\ y \in A \Longrightarrow P\ (Abs\ y)$
  shows $P\ x$

⟨*proof*⟩

lemma *Rep-range*: $range\ Rep = A$
⟨*proof*⟩

lemma *Abs-image*: $Abs\ `\ A = UNIV$
⟨*proof*⟩

end

⟨*ML*⟩

end

9 Notions about functions

theory *Fun*
imports *Set*
keywords *functor* :: *thy-goal*
begin

lemma *apply-inverse*: $f\,x = u \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow g\ (f\,x) = x) \Longrightarrow P\ x \Longrightarrow x = g\ u$
⟨*proof*⟩

Uniqueness, so NOT the axiom of choice.

lemma *uniq-choice*: $\forall\,x.\ \exists!y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall\,x.\ Q\ x\ (f\,x)$
⟨*proof*⟩

lemma *b-uniq-choice*: $\forall\,x{\in}S.\ \exists!y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall\,x{\in}S.\ Q\ x\ (f\,x)$
⟨*proof*⟩

9.1 The Identity Function *id*

definition *id* :: $'a \Rightarrow 'a$
where $id = (\lambda x.\ x)$

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

lemma *image-id* [*simp*]: $image\ id = id$
⟨*proof*⟩

lemma *vimage-id* [*simp*]: $vimage\ id = id$
⟨*proof*⟩

lemma *eq-id-iff*: $(\forall\,x.\ f\,x = x) \longleftrightarrow f = id$
⟨*proof*⟩

code-printing
  constant $id \rightharpoonup$ (*Haskell*) *id*

9.2 The Composition Operator $f \circ g$

definition $comp :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** $\circ$ *55*)
  where $f \circ g = (\lambda x.\ f\ (g\ x))$

notation (*ASCII*)
  comp (**infixl** *o 55*)

lemma *comp-apply* [*simp*]: $(f \circ g)\ x = f\ (g\ x)$
  ⟨*proof*⟩

lemma *comp-assoc*: $(f \circ g) \circ h = f \circ (g \circ h)$
  ⟨*proof*⟩

lemma *id-comp* [*simp*]: $id \circ g = g$
  ⟨*proof*⟩

lemma *comp-id* [*simp*]: $f \circ id = f$
  ⟨*proof*⟩

lemma *comp-eq-dest*: $a \circ b = c \circ d \Longrightarrow a\ (b\ v) = c\ (d\ v)$
  ⟨*proof*⟩

lemma *comp-eq-elim*: $a \circ b = c \circ d \Longrightarrow ((\bigwedge v.\ a\ (b\ v) = c\ (d\ v)) \Longrightarrow R) \Longrightarrow R$
  ⟨*proof*⟩

lemma *comp-eq-dest-lhs*: $a \circ b = c \Longrightarrow a\ (b\ v) = c\ v$
  ⟨*proof*⟩

lemma *comp-eq-id-dest*: $a \circ b = id \circ c \Longrightarrow a\ (b\ v) = c\ v$
  ⟨*proof*⟩

lemma *image-comp*: $f\ `\ (g\ `\ r) = (f \circ g)\ `\ r$
  ⟨*proof*⟩

lemma *vimage-comp*: $f\ -`\ (g\ -`\ x) = (g \circ f)\ -`\ x$
  ⟨*proof*⟩

lemma *image-eq-imp-comp*: $f\ `\ A = g\ `\ B \Longrightarrow (h \circ f)\ `\ A = (h \circ g)\ `\ B$
  ⟨*proof*⟩

lemma *image-bind*: $f\ `\ (Set.bind\ A\ g) = Set.bind\ A\ (op\ `\ f \circ g)$
  ⟨*proof*⟩

lemma *bind-image*: $Set.bind\ (f\ `\ A)\ g = Set.bind\ A\ (g \circ f)$
  ⟨*proof*⟩

lemma (**in** *group-add*) *minus-comp-minus* [*simp*]: $uminus \circ uminus = id$
⟨*proof*⟩

lemma (**in** *boolean-algebra*) *minus-comp-minus* [*simp*]: $uminus \circ uminus = id$
⟨*proof*⟩

code-printing
constant *comp* ⇀ (*SML*) **infixl** *5 o* **and** (*Haskell*) **infixr** *9 .*

9.3 The Forward Composition Operator *fcomp*

definition $fcomp :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$ (**infixl** ∘> *60*)
where $f \circ\!> g = (\lambda x.\ g\ (f\ x))$

lemma *fcomp-apply* [*simp*]: $(f \circ\!> g)\ x = g\ (f\ x)$
⟨*proof*⟩

lemma *fcomp-assoc*: $(f \circ\!> g) \circ\!> h = f \circ\!> (g \circ\!> h)$
⟨*proof*⟩

lemma *id-fcomp* [*simp*]: $id \circ\!> g = g$
⟨*proof*⟩

lemma *fcomp-id* [*simp*]: $f \circ\!> id = f$
⟨*proof*⟩

lemma *fcomp-comp*: $fcomp\ f\ g = comp\ g\ f$
⟨*proof*⟩

code-printing
constant *fcomp* ⇀ (*Eval*) **infixl** *1 #>*

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

9.4 Mapping functions

definition $map\text{-}fun :: ('c \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'd$
where $map\text{-}fun\ f\ g\ h = g \circ h \circ f$

lemma *map-fun-apply* [*simp*]: $map\text{-}fun\ f\ g\ h\ x = g\ (h\ (f\ x))$
⟨*proof*⟩

9.5 Injectivity and Bijectivity

definition $inj\text{-}on :: ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow bool$ — injective
where $inj\text{-}on\ f\ A \longleftrightarrow (\forall x{\in}A.\ \forall y{\in}A.\ f\ x = f\ y \longrightarrow x = y)$

definition $bij\text{-}betw :: ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$ — bijective
where $bij\text{-}betw\ f\ A\ B \longleftrightarrow inj\text{-}on\ f\ A \wedge f\ `\ A = B$

A common special case: functions injective, surjective or bijective over the entire domain type.

abbreviation *inj* :: $('a \Rightarrow 'b) \Rightarrow bool$
where $inj\ f \equiv inj\text{-}on\ f\ UNIV$

abbreviation *surj* :: $('a \Rightarrow 'b) \Rightarrow bool$
where $surj\ f \equiv range\ f = UNIV$

translations — The negated case:
$\neg\ CONST\ surj\ f \leftharpoondown CONST\ range\ f \neq CONST\ UNIV$

abbreviation *bij* :: $('a \Rightarrow 'b) \Rightarrow bool$
where $bij\ f \equiv bij\text{-}betw\ f\ UNIV\ UNIV$

lemma *inj-def*: $inj\ f \longleftrightarrow (\forall\, x\ y.\ f\ x = f\ y \longrightarrow x = y)$
⟨*proof*⟩

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

theorem *range-ex1-eq*: $inj\ f \Longrightarrow b \in range\ f \longleftrightarrow (\exists !x.\ b = f\ x)$
⟨*proof*⟩

lemma *injD*: $inj\ f \Longrightarrow f\ x = f\ y \Longrightarrow x = y$
⟨*proof*⟩

lemma *inj-on-eq-iff*: $inj\text{-}on\ f\ A \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *inj-on-cong*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a) \Longrightarrow inj\text{-}on\ f\ A \longleftrightarrow inj\text{-}on\ g\ A$
⟨*proof*⟩

lemma *inj-on-strict-subset*: $inj\text{-}on\ f\ B \Longrightarrow A \subset B \Longrightarrow f\ `\ A \subset f\ `\ B$
⟨*proof*⟩

lemma *inj-comp*: $inj\ f \Longrightarrow inj\ g \Longrightarrow inj\ (f \circ g)$
⟨*proof*⟩

lemma *inj-fun*: $inj\ f \Longrightarrow inj\ (\lambda x\ y.\ f\ x)$
⟨*proof*⟩

lemma *inj-eq*: $inj\ f \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *inj-on-id*[*simp*]: $inj\text{-}on\ id\ A$
⟨*proof*⟩

lemma *inj-on-id2*[*simp*]: $inj\text{-}on\ (\lambda x.\ x)\ A$
⟨*proof*⟩

lemma *inj-on-Int*: $inj\text{-}on\ f\ A \lor inj\text{-}on\ f\ B \Longrightarrow inj\text{-}on\ f\ (A \cap B)$
⟨*proof*⟩

lemma *surj-id*: *surj id*
⟨*proof*⟩

lemma *bij-id*[*simp*]: *bij id*
⟨*proof*⟩

lemma *bij-uminus*: $bij\ (uminus :: 'a \Rightarrow 'a{::}ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *inj-onI* [*intro?*]: $(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow x = y) \Longrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

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

lemma *inj-onD*: $inj\text{-}on\ f\ A \Longrightarrow f\ x = f\ y \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow x = y$
⟨*proof*⟩

lemma *inj-on-subset*:
assumes *inj-on f A*
and $B \subseteq A$
shows *inj-on f B*
⟨*proof*⟩

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

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

lemma *inj-on-image-iff*:
$\forall x \in A.\ \forall y \in A.\ g\ (f\ x) = g\ (f\ y) \longleftrightarrow g\ x = g\ y \Longrightarrow inj\text{-}on\ f\ A \Longrightarrow inj\text{-}on\ g\ (f\ `\ A) \longleftrightarrow inj\text{-}on\ g\ A$
⟨*proof*⟩

lemma *inj-on-contraD*: $inj\text{-}on\ f\ A \Longrightarrow x \neq y \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow f\ x \neq f\ y$
⟨*proof*⟩

lemma *inj-singleton* [*simp*]: $inj\text{-}on\ (\lambda x.\ \{x\})\ A$
⟨*proof*⟩

lemma *inj-on-empty*[*iff*]: $inj\text{-}on\ f\ \{\}$
⟨*proof*⟩

lemma *subset-inj-on*: $inj\text{-}on\ f\ B \Longrightarrow A \subseteq B \Longrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *inj-on-Un*: $inj\text{-}on\ f\ (A \cup B) \longleftrightarrow inj\text{-}on\ f\ A \wedge inj\text{-}on\ f\ B \wedge f\ `\ (A - B) \cap f\ `\ (B - A) = \{\}$
⟨*proof*⟩

lemma *inj-on-insert* [*iff*]: $inj\text{-}on\ f\ (insert\ a\ A) \longleftrightarrow inj\text{-}on\ f\ A \wedge f\ a \notin f\ `\ (A - \{a\})$
⟨*proof*⟩

lemma *inj-on-diff*: $inj\text{-}on\ f\ A \Longrightarrow inj\text{-}on\ f\ (A - B)$
⟨*proof*⟩

lemma *comp-inj-on-iff*: $inj\text{-}on\ f\ A \Longrightarrow inj\text{-}on\ f'\ (f\ `\ A) \longleftrightarrow inj\text{-}on\ (f' \circ f)\ A$
⟨*proof*⟩

lemma *inj-on-imageI2*: $inj\text{-}on\ (f' \circ f)\ A \Longrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *inj-img-insertE*:
assumes $inj\text{-}on\ f\ A$
assumes $x \notin B$
and $insert\ x\ B = f\ `\ A$
obtains $x'\ A'$ **where** $x' \notin A'$ **and** $A = insert\ x'\ A'$ **and** $x = f\ x'$ **and** $B = f\ `\ A'$
⟨*proof*⟩

lemma *linorder-injI*:
assumes $\bigwedge x\ y{::}'a{::}linorder.\ x < y \Longrightarrow f\ x \neq f\ y$
shows $inj\ f$
— Courtesy of Stephan Merz
⟨*proof*⟩

lemma *surj-def*: $surj\ f \longleftrightarrow (\forall y.\ \exists x.\ y = f\ x)$
⟨*proof*⟩

lemma *surjI*:
assumes $\bigwedge x.\ g\ (f\ x) = x$
shows $surj\ g$
⟨*proof*⟩

lemma *surjD*: $surj\ f \Longrightarrow \exists x.\ y = f\ x$
⟨*proof*⟩

lemma *surjE*: $surj\ f \Longrightarrow (\bigwedge x.\ y = f\ x \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

lemma *comp-surj*: $surj\ f \Longrightarrow surj\ g \Longrightarrow surj\ (g \circ f)$

⟨proof⟩

lemma *bij-betw-imageI*: *inj-on f A* $\Longrightarrow$ *f* ‘ *A* = *B* $\Longrightarrow$ *bij-betw f A B*
⟨proof⟩

lemma *bij-betw-imp-surj-on*: *bij-betw f A B* $\Longrightarrow$ *f* ‘ *A* = *B*
⟨proof⟩

lemma *bij-betw-imp-surj*: *bij-betw f A UNIV* $\Longrightarrow$ *surj f*
⟨proof⟩

lemma *bij-betw-empty1*: *bij-betw f* {} *A* $\Longrightarrow$ *A* = {}
⟨proof⟩

lemma *bij-betw-empty2*: *bij-betw f A* {} $\Longrightarrow$ *A* = {}
⟨proof⟩

lemma *inj-on-imp-bij-betw*: *inj-on f A* $\Longrightarrow$ *bij-betw f A* (*f* ‘ *A*)
⟨proof⟩

lemma *bij-def*: *bij f* $\longleftrightarrow$ *inj f* $\wedge$ *surj f*
⟨proof⟩

lemma *bijI*: *inj f* $\Longrightarrow$ *surj f* $\Longrightarrow$ *bij f*
⟨proof⟩

lemma *bij-is-inj*: *bij f* $\Longrightarrow$ *inj f*
⟨proof⟩

lemma *bij-is-surj*: *bij f* $\Longrightarrow$ *surj f*
⟨proof⟩

lemma *bij-betw-imp-inj-on*: *bij-betw f A B* $\Longrightarrow$ *inj-on f A*
⟨proof⟩

lemma *bij-betw-trans*: *bij-betw f A B* $\Longrightarrow$ *bij-betw g B C* $\Longrightarrow$ *bij-betw* ($g \circ f$) *A C*
⟨proof⟩

lemma *bij-comp*: *bij f* $\Longrightarrow$ *bij g* $\Longrightarrow$ *bij* ($g \circ f$)
⟨proof⟩

lemma *bij-betw-comp-iff*: *bij-betw f A A'* $\Longrightarrow$ *bij-betw f' A' A''* $\longleftrightarrow$ *bij-betw* ($f' \circ f$) *A A''*
⟨proof⟩

lemma *bij-betw-comp-iff2*:
assumes *bij*: *bij-betw f' A' A''*
and *img*: *f* ‘ *A* $\leq$ *A'*
shows *bij-betw f A A'* $\longleftrightarrow$ *bij-betw* ($f' \circ f$) *A A''*

⟨*proof*⟩

lemma *bij-betw-inv*:
assumes *bij-betw f A B*
shows $\exists g.$ *bij-betw g B A*
⟨*proof*⟩

lemma *bij-betw-cong*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a) \Longrightarrow$ *bij-betw f A A'* $=$ *bij-betw g A A'*
⟨*proof*⟩

lemma *bij-betw-id*[*intro*, *simp*]: *bij-betw id A A*
⟨*proof*⟩

lemma *bij-betw-id-iff*: *bij-betw id A B* $\longleftrightarrow A = B$
⟨*proof*⟩

lemma *bij-betw-combine*:
bij-betw f A B $\Longrightarrow$ *bij-betw f C D* $\Longrightarrow B \cap D = \{\} \Longrightarrow$ *bij-betw f* $(A \cup C)$ $(B \cup D)$
⟨*proof*⟩

lemma *bij-betw-subset*: *bij-betw f A A'* $\Longrightarrow B \subseteq A \Longrightarrow f\ `\ B = B' \Longrightarrow$ *bij-betw f B B'*
⟨*proof*⟩

lemma *bij-pointE*:
assumes *bij f*
obtains x **where** $y = f\ x$ **and** $\bigwedge x'.\ y = f\ x' \Longrightarrow x' = x$
⟨*proof*⟩

lemma *surj-image-vimage-eq*: *surj f* $\Longrightarrow f\ `\ (f\ -`\ A) = A$
⟨*proof*⟩

lemma *surj-vimage-empty*:
assumes *surj f*
shows $f\ -`\ A = \{\} \longleftrightarrow A = \{\}$
⟨*proof*⟩

lemma *inj-vimage-image-eq*: *inj f* $\Longrightarrow f\ -`\ (f\ `\ A) = A$
⟨*proof*⟩

lemma *vimage-subsetD*: *surj f* $\Longrightarrow f\ -`\ B \subseteq A \Longrightarrow B \subseteq f\ `\ A$
⟨*proof*⟩

lemma *vimage-subsetI*: *inj f* $\Longrightarrow B \subseteq f\ `\ A \Longrightarrow f\ -`\ B \subseteq A$
⟨*proof*⟩

lemma *vimage-subset-eq*: *bij f* $\Longrightarrow f\ -`\ B \subseteq A \longleftrightarrow B \subseteq f\ `\ A$

⟨*proof*⟩

lemma *inj-on-image-eq-iff*: $inj\text{-}on\ f\ C \Longrightarrow A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow f\ `\ A = f\ `\ B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *inj-on-Un-image-eq-iff*: $inj\text{-}on\ f\ (A \cup B) \Longrightarrow f\ `\ A = f\ `\ B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *inj-on-image-Int*: $inj\text{-}on\ f\ C \Longrightarrow A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow f\ `\ (A \cap B) = f\ `\ A \cap f\ `\ B$
⟨*proof*⟩

lemma *inj-on-image-set-diff*: $inj\text{-}on\ f\ C \Longrightarrow A - B \subseteq C \Longrightarrow B \subseteq C \Longrightarrow f\ `\ (A - B) = f\ `\ A - f\ `\ B$
⟨*proof*⟩

lemma *image-Int*: $inj\ f \Longrightarrow f\ `\ (A \cap B) = f\ `\ A \cap f\ `\ B$
⟨*proof*⟩

lemma *image-set-diff*: $inj\ f \Longrightarrow f\ `\ (A - B) = f\ `\ A - f\ `\ B$
⟨*proof*⟩

lemma *inj-on-image-mem-iff*: $inj\text{-}on\ f\ B \Longrightarrow a \in B \Longrightarrow A \subseteq B \Longrightarrow f\ a \in f\ `\ A \longleftrightarrow a \in A$
⟨*proof*⟩

lemma *inj-on-image-mem-iff-alt*: $inj\text{-}on\ f\ B \Longrightarrow A \subseteq B \Longrightarrow f\ a \in f\ `\ A \Longrightarrow a \in B \Longrightarrow a \in A$
⟨*proof*⟩

lemma *inj-image-mem-iff*: $inj\ f \Longrightarrow f\ a \in f\ `\ A \longleftrightarrow a \in A$
⟨*proof*⟩

lemma *inj-image-subset-iff*: $inj\ f \Longrightarrow f\ `\ A \subseteq f\ `\ B \longleftrightarrow A \subseteq B$
⟨*proof*⟩

lemma *inj-image-eq-iff*: $inj\ f \Longrightarrow f\ `\ A = f\ `\ B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *surj-Compl-image-subset*: $surj\ f \Longrightarrow -\ (f\ `\ A) \subseteq f\ `\ (-\ A)$
⟨*proof*⟩

lemma *inj-image-Compl-subset*: $inj\ f \Longrightarrow f\ `\ (-\ A) \subseteq -\ (f\ `\ A)$
⟨*proof*⟩

lemma *bij-image-Compl-eq*: $bij\ f \Longrightarrow f\ `\ (-\ A) = -\ (f\ `\ A)$
⟨*proof*⟩

lemma *inj-vimage-singleton*: *inj f* $\Longrightarrow$ *f* $-`$ $\{a\} \subseteq \{$*THE x*. *f x* $= a\}$
— The inverse image of a singleton under an injective function is included in a singleton.
⟨*proof*⟩

lemma *inj-on-vimage-singleton*: *inj-on f A* $\Longrightarrow$ *f* $-`$ $\{a\} \cap A \subseteq \{$*THE x*. $x \in A$ $\wedge$ *f x* $= a\}$
⟨*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* $\Longrightarrow$ *inj-on f A*
⟨*proof*⟩

lemma *bij-betw-byWitness*:
assumes *left*: $\forall a \in A.\ f'\ (f\ a) = a$
and *right*: $\forall a' \in A'.\ f\ (f'\ a') = a'$
and f ` $A \subseteq A'$
and *img2*: f' ` $A' \subseteq A$
shows *bij-betw f A A'*
⟨*proof*⟩

corollary *notIn-Un-bij-betw*:
assumes $b \notin A$
and $f\ b \notin A'$
and *bij-betw f A A'*
shows *bij-betw f* $(A \cup \{b\})\ (A' \cup \{f\ b\})$
⟨*proof*⟩

lemma *notIn-Un-bij-betw3*:
assumes $b \notin A$
and $f\ b \notin A'$
shows *bij-betw f A A'* = *bij-betw f* $(A \cup \{b\})\ (A' \cup \{f\ b\})$
⟨*proof*⟩

9.6 Function Updating

definition *fun-upd* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$
where *fun-upd f a b* $= (\lambda x.$ *if* $x = a$ *then b else f x*$)$

nonterminal *updbinds* **and** *updbind*

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

translations
$-Update\ f\ (-updbinds\ b\ bs) \rightleftharpoons -Update\ (-Update\ f\ b)\ bs$
$f(x{:=}y) \rightleftharpoons CONST\ fun\text{-}upd\ f\ x\ y$

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

lemma *fun-upd-idem*: $f\ x = y \Longrightarrow f(x := y) = f$
⟨*proof*⟩

lemma *fun-upd-triv* [*iff*]: $f(x := f\ x) = f$
⟨*proof*⟩

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*: $z \neq x \Longrightarrow (f(x := y))\ z = f\ z$
⟨*proof*⟩

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

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

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

lemma *fun-upd-eqD*: $f(x := y) = g(x := z) \Longrightarrow y = z$
⟨*proof*⟩

9.7 *override-on*

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

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

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

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

lemma *override-on-insert*: *override-on f g* (*insert x X*) = (*override-on f g X*)(*x*:=*g x*)
⟨*proof*⟩

lemma *override-on-insert'*: *override-on f g* (*insert x X*) = (*override-on* (*f*(*x*:=*g x*)) *g X*)
⟨*proof*⟩

9.8 *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-apply* [*simp*]:
swap a b f a = *f b*
swap a b f b = *f a*
$c \neq a \Longrightarrow c \neq b \Longrightarrow \textit{swap } a\ b\ f\ c = f\ c$
⟨*proof*⟩

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-comp-involutory* [*simp*]: $\textit{swap } a\ b \circ \textit{swap } a\ b = \textit{id}$
⟨*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*: $f \circ \textit{swap } a\ b\ g = \textit{swap } a\ b\ (f \circ g)$
⟨*proof*⟩

lemma *swap-image-eq* [*simp*]:
assumes $a \in A$ $b \in A$
shows *swap* a b f ‘ $A = f$ ‘ A
⟨*proof*⟩

lemma *inj-on-imp-inj-on-swap*: *inj-on* f $A \Longrightarrow a \in A \Longrightarrow b \in A \Longrightarrow$ *inj-on* (*swap* a b f) A
⟨*proof*⟩

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

lemma *surj-imp-surj-swap*: *surj* $f \Longrightarrow$ *surj* (*swap* a b f)
⟨*proof*⟩

lemma *surj-swap-iff* [*simp*]: *surj* (*swap* a b f) $\longleftrightarrow$ *surj* f
⟨*proof*⟩

lemma *bij-betw-swap-iff* [*simp*]: $x \in A \Longrightarrow y \in A \Longrightarrow$ *bij-betw* (*swap* x y f) A B $\longleftrightarrow$ *bij-betw* f A B
⟨*proof*⟩

lemma *bij-swap-iff* [*simp*]: *bij* (*swap* a b f) $\longleftrightarrow$ *bij* f
⟨*proof*⟩

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

9.9 Inversion of injective functions

definition *the-inv-into* :: $'a$ *set* $\Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$
where *the-inv-into* A f = (λx. *THE* y. $y \in A \land f\ y = x$)

lemma *the-inv-into-f-f*: *inj-on* f $A \Longrightarrow x \in A \Longrightarrow$ *the-inv-into* A f $(f\ x) = x$
⟨*proof*⟩

lemma *f-the-inv-into-f*: *inj-on* f $A \Longrightarrow y \in f$ ‘ $A \Longrightarrow f$ (*the-inv-into* A f y) $= y$
⟨*proof*⟩

lemma *the-inv-into-into*: *inj-on* f $A \Longrightarrow x \in f$ ‘ $A \Longrightarrow A \subseteq B \Longrightarrow$ *the-inv-into* A f $x \in B$
⟨*proof*⟩

lemma *the-inv-into-onto* [*simp*]: *inj-on* f $A \Longrightarrow$ *the-inv-into* A f ‘ (f ‘ A) $= A$
⟨*proof*⟩

lemma *the-inv-into-f-eq*: *inj-on* f $A \Longrightarrow f\ x = y \Longrightarrow x \in A \Longrightarrow$ *the-inv-into* A f $y = x$

⟨*proof*⟩

lemma *the-inv-into-comp*:
$inj\text{-}on\ f\ (g\ `\ A) \Longrightarrow inj\text{-}on\ g\ A \Longrightarrow x \in f\ `\ g\ `\ A \Longrightarrow$
$the\text{-}inv\text{-}into\ A\ (f \circ g)\ x = (the\text{-}inv\text{-}into\ A\ g \circ the\text{-}inv\text{-}into\ (g\ `\ A)\ f)\ x$
⟨*proof*⟩

lemma *inj-on-the-inv-into*: $inj\text{-}on\ f\ A \Longrightarrow inj\text{-}on\ (the\text{-}inv\text{-}into\ A\ f)\ (f\ `\ A)$
⟨*proof*⟩

lemma *bij-betw-the-inv-into*: $bij\text{-}betw\ f\ A\ B \Longrightarrow bij\text{-}betw\ (the\text{-}inv\text{-}into\ A\ f)\ B\ A$
⟨*proof*⟩

abbreviation *the-inv* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$
where $the\text{-}inv\ f \equiv the\text{-}inv\text{-}into\ UNIV\ f$

lemma *the-inv-f-f*: $the\text{-}inv\ f\ (f\ x) = x$ **if** *inj f*
⟨*proof*⟩

9.10 Cantor's Paradox

theorem *Cantors-paradox*: $\nexists f.\ f\ `\ A = Pow\ A$
⟨*proof*⟩

9.11 Setup

9.11.1 Proof tools

Simplify terms of the form $f(\ldots,x{:=}y,\ldots,x{:=}z,\ldots)$ to $f(\ldots,x{:=}z,\ldots)$

⟨*ML*⟩

9.11.2 Functorial structure of types

⟨*ML*⟩

functor *map-fun*: *map-fun*
⟨*proof*⟩

functor *vimage*
⟨*proof*⟩

Legacy theorem names

lemmas *o-def* = *comp-def*
lemmas *o-apply* = *comp-apply*
lemmas *o-assoc* = *comp-assoc* [*symmetric*]
lemmas *id-o* = *id-comp*
lemmas *o-id* = *comp-id*
lemmas *o-eq-dest* = *comp-eq-dest*
lemmas *o-eq-elim* = *comp-eq-elim*

lemmas *o-eq-dest-lhs* = *comp-eq-dest-lhs*
lemmas *o-eq-id-dest* = *comp-eq-id-dest*

end

10 Complete lattices

theory *Complete-Lattices*
imports *Fun*
begin

10.1 Syntactic infimum and supremum operations

class *Inf* =
fixes *Inf* :: $'a$ *set* ⇒ $'a$ (⨅- [900] 900)
begin

abbreviation *INFIMUM* :: $'b$ *set* ⇒ ($'b$ ⇒ $'a$) ⇒ $'a$
where *INFIMUM A f* ≡ ⨅(*f* ‘ *A*)

lemma *INF-image* [*simp*]: *INFIMUM* (*f* ‘ *A*) *g* = *INFIMUM A* (*g* ∘ *f*)
⟨*proof*⟩

lemma *INF-identity-eq* [*simp*]: *INFIMUM A* (λ*x*. *x*) = ⨅*A*
⟨*proof*⟩

lemma *INF-id-eq* [*simp*]: *INFIMUM A id* = ⨅*A*
⟨*proof*⟩

lemma *INF-cong*: *A* = *B* ⟹ (⋀*x*. *x* ∈ *B* ⟹ *C x* = *D x*) ⟹ *INFIMUM A C* = *INFIMUM B D*
⟨*proof*⟩

lemma *strong-INF-cong* [*cong*]:
A = *B* ⟹ (⋀*x*. *x* ∈ *B* =*simp*=> *C x* = *D x*) ⟹ *INFIMUM A C* = *INFIMUM B D*
⟨*proof*⟩

end

class *Sup* =
fixes *Sup* :: $'a$ *set* ⇒ $'a$ (⨆- [900] 900)
begin

abbreviation *SUPREMUM* :: $'b$ *set* ⇒ ($'b$ ⇒ $'a$) ⇒ $'a$
where *SUPREMUM A f* ≡ ⨆(*f* ‘ *A*)

lemma *SUP-image* [*simp*]: *SUPREMUM* (*f* ‘ *A*) *g* = *SUPREMUM A* (*g* ∘ *f*)
⟨*proof*⟩

lemma *SUP-identity-eq* [*simp*]: *SUPREMUM A* ($\lambda x.\ x$) = $\bigsqcup A$
⟨*proof*⟩

lemma *SUP-id-eq* [*simp*]: *SUPREMUM A id* = $\bigsqcup A$
⟨*proof*⟩

lemma *SUP-cong*: $A = B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow C\ x = D\ x) \Longrightarrow$ *SUPREMUM A C* = *SUPREMUM B D*
⟨*proof*⟩

lemma *strong-SUP-cong* [*cong*]:
$A = B \Longrightarrow (\bigwedge x.\ x \in B$ =*simp*=> $C\ x = D\ x) \Longrightarrow$ *SUPREMUM A C* = *SUPREMUM B D*
⟨*proof*⟩

end

Note: must use names *INFIMUM* and *SUPREMUM* here instead of *INF* and *SUP* to allow the following syntax coexist with the plain constant names.

syntax (*ASCII*)
-*INF1* :: *pttrns* ⇒ ′*b* ⇒ ′*b* ((*3INF* -./ -) [*0*, *10*] *10*)
-*INF* :: *pttrn* ⇒ ′*a set* ⇒ ′*b* ⇒ ′*b* ((*3INF* -:-./ -) [*0*, *0*, *10*] *10*)
-*SUP1* :: *pttrns* ⇒ ′*b* ⇒ ′*b* ((*3SUP* -./ -) [*0*, *10*] *10*)
-*SUP* :: *pttrn* ⇒ ′*a set* ⇒ ′*b* ⇒ ′*b* ((*3SUP* -:-./ -) [*0*, *0*, *10*] *10*)

syntax (**output**)
-*INF1* :: *pttrns* ⇒ ′*b* ⇒ ′*b* ((*3INF* -./ -) [*0*, *10*] *10*)
-*INF* :: *pttrn* ⇒ ′*a set* ⇒ ′*b* ⇒ ′*b* ((*3INF* -:-./ -) [*0*, *0*, *10*] *10*)
-*SUP1* :: *pttrns* ⇒ ′*b* ⇒ ′*b* ((*3SUP* -./ -) [*0*, *10*] *10*)
-*SUP* :: *pttrn* ⇒ ′*a set* ⇒ ′*b* ⇒ ′*b* ((*3SUP* -:-./ -) [*0*, *0*, *10*] *10*)

syntax
-*INF1* :: *pttrns* ⇒ ′*b* ⇒ ′*b* ((*3*⨅-./ -) [*0*, *10*] *10*)
-*INF* :: *pttrn* ⇒ ′*a set* ⇒ ′*b* ⇒ ′*b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)
-*SUP1* :: *pttrns* ⇒ ′*b* ⇒ ′*b* ((*3*⨆-./ -) [*0*, *10*] *10*)
-*SUP* :: *pttrn* ⇒ ′*a set* ⇒ ′*b* ⇒ ′*b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)

translations
⨅*x y*. *B* ⇌ ⨅*x*. ⨅*y*. *B*
⨅*x*. *B* ⇌ *CONST INFIMUM CONST UNIV* ($\lambda x.\ B$)
⨅*x*. *B* ⇌ ⨅*x* ∈ *CONST UNIV*. *B*
⨅*x*∈*A*. *B* ⇌ *CONST INFIMUM A* ($\lambda x.\ B$)
⨆*x y*. *B* ⇌ ⨆*x*. ⨆*y*. *B*
⨆*x*. *B* ⇌ *CONST SUPREMUM CONST UNIV* ($\lambda x.\ B$)
⨆*x*. *B* ⇌ ⨆*x* ∈ *CONST UNIV*. *B*
⨆*x*∈*A*. *B* ⇌ *CONST SUPREMUM A* ($\lambda x.\ B$)

⟨*ML*⟩

10.2 Abstract complete lattices

A complete lattice always has a bottom and a top, so we include them into the following type class, along with assumptions that define bottom and top in terms of infimum and supremum.

class *complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +
  assumes *Inf-lower*: $x \in A \Longrightarrow \bigsqcap A \leq x$
    and *Inf-greatest*: $(\bigwedge x.\ x \in A \Longrightarrow z \leq x) \Longrightarrow z \leq \bigsqcap A$
    and *Sup-upper*: $x \in A \Longrightarrow x \leq \bigsqcup A$
    and *Sup-least*: $(\bigwedge x.\ x \in A \Longrightarrow x \leq z) \Longrightarrow \bigsqcup A \leq z$
    and *Inf-empty* [*simp*]: $\bigsqcap\{\} = \top$
    and *Sup-empty* [*simp*]: $\bigsqcup\{\} = \bot$
begin

subclass *bounded-lattice*
⟨*proof*⟩

lemma *dual-complete-lattice*: *class.complete-lattice Sup Inf sup* $(op \geq)$ $(op >)$ *inf* $\top$ $\bot$
  ⟨*proof*⟩

end

context *complete-lattice*
begin

lemma *Sup-eqI*:
  $(\bigwedge y.\ y \in A \Longrightarrow y \leq x) \Longrightarrow (\bigwedge y.\ (\bigwedge z.\ z \in A \Longrightarrow z \leq y) \Longrightarrow x \leq y) \Longrightarrow \bigsqcup A = x$
  ⟨*proof*⟩

lemma *Inf-eqI*:
  $(\bigwedge i.\ i \in A \Longrightarrow x \leq i) \Longrightarrow (\bigwedge y.\ (\bigwedge i.\ i \in A \Longrightarrow y \leq i) \Longrightarrow y \leq x) \Longrightarrow \bigsqcap A = x$
  ⟨*proof*⟩

lemma *SUP-eqI*:
  $(\bigwedge i.\ i \in A \Longrightarrow f\ i \leq x) \Longrightarrow (\bigwedge y.\ (\bigwedge i.\ i \in A \Longrightarrow f\ i \leq y) \Longrightarrow x \leq y) \Longrightarrow (\bigsqcup i{\in}A.\ f\ i) = x$
  ⟨*proof*⟩

lemma *INF-eqI*:
  $(\bigwedge i.\ i \in A \Longrightarrow x \leq f\ i) \Longrightarrow (\bigwedge y.\ (\bigwedge i.\ i \in A \Longrightarrow f\ i \geq y) \Longrightarrow x \geq y) \Longrightarrow (\bigsqcap i{\in}A.\ f\ i) = x$
  ⟨*proof*⟩

lemma *INF-lower*: $i \in A \Longrightarrow (\bigsqcap i{\in}A.\ f\ i) \leq f\ i$
  ⟨*proof*⟩

lemma *INF-greatest*: $(\bigwedge i.\ i \in A \Longrightarrow u \leq f\ i) \Longrightarrow u \leq (\bigsqcap i{\in}A.\ f\ i)$

⟨*proof*⟩

lemma *SUP-upper*: $i \in A \Longrightarrow f\ i \leq (\bigsqcup i{\in}A.\ f\ i)$
⟨*proof*⟩

lemma *SUP-least*: $(\bigwedge i.\ i \in A \Longrightarrow f\ i \leq u) \Longrightarrow (\bigsqcup i{\in}A.\ f\ i) \leq u$
⟨*proof*⟩

lemma *Inf-lower2*: $u \in A \Longrightarrow u \leq v \Longrightarrow \bigsqcap A \leq v$
⟨*proof*⟩

lemma *INF-lower2*: $i \in A \Longrightarrow f\ i \leq u \Longrightarrow (\bigsqcap i{\in}A.\ f\ i) \leq u$
⟨*proof*⟩

lemma *Sup-upper2*: $u \in A \Longrightarrow v \leq u \Longrightarrow v \leq \bigsqcup A$
⟨*proof*⟩

lemma *SUP-upper2*: $i \in A \Longrightarrow u \leq f\ i \Longrightarrow u \leq (\bigsqcup i{\in}A.\ f\ i)$
⟨*proof*⟩

lemma *le-Inf-iff*: $b \leq \bigsqcap A \longleftrightarrow (\forall\, a{\in}A.\ b \leq a)$
⟨*proof*⟩

lemma *le-INF-iff*: $u \leq (\bigsqcap i{\in}A.\ f\ i) \longleftrightarrow (\forall\, i{\in}A.\ u \leq f\ i)$
⟨*proof*⟩

lemma *Sup-le-iff*: $\bigsqcup A \leq b \longleftrightarrow (\forall\, a{\in}A.\ a \leq b)$
⟨*proof*⟩

lemma *SUP-le-iff*: $(\bigsqcup i{\in}A.\ f\ i) \leq u \longleftrightarrow (\forall\, i{\in}A.\ f\ i \leq u)$
⟨*proof*⟩

lemma *Inf-insert* [*simp*]: $\bigsqcap \mathit{insert}\ a\ A = a \sqcap \bigsqcap A$
⟨*proof*⟩

lemma *INF-insert* [*simp*]: $(\bigsqcap x{\in}\mathit{insert}\ a\ A.\ f\ x) = f\ a \sqcap \mathit{INFIMUM}\ A\ f$
⟨*proof*⟩

lemma *Sup-insert* [*simp*]: $\bigsqcup \mathit{insert}\ a\ A = a \sqcup \bigsqcup A$
⟨*proof*⟩

lemma *SUP-insert* [*simp*]: $(\bigsqcup x{\in}\mathit{insert}\ a\ A.\ f\ x) = f\ a \sqcup \mathit{SUPREMUM}\ A\ f$
⟨*proof*⟩

lemma *INF-empty* [*simp*]: $(\bigsqcap x{\in}\{\}.\ f\ x) = \top$
⟨*proof*⟩

lemma *SUP-empty* [*simp*]: $(\bigsqcup x{\in}\{\}.\ f\ x) = \bot$
⟨*proof*⟩

lemma *Inf-UNIV* [*simp*]: $\bigsqcap$ *UNIV* $= \bot$
⟨*proof*⟩

lemma *Sup-UNIV* [*simp*]: $\bigsqcup$ *UNIV* $= \top$
⟨*proof*⟩

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

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

lemma *Inf-superset-mono*: $B \subseteq A \Longrightarrow \bigsqcap A \leq \bigsqcap B$
⟨*proof*⟩

lemma *Sup-subset-mono*: $A \subseteq B \Longrightarrow \bigsqcup A \leq \bigsqcup B$
⟨*proof*⟩

lemma *Inf-mono*:
assumes $\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b$
shows $\bigsqcap A \leq \bigsqcap B$
⟨*proof*⟩

lemma *INF-mono*: $(\bigwedge m.\ m \in B \Longrightarrow \exists\, n{\in}A.\ f\ n \leq g\ m) \Longrightarrow (\bigsqcap n{\in}A.\ f\ n) \leq (\bigsqcap n{\in}B.\ g\ n)$
⟨*proof*⟩

lemma *Sup-mono*:
assumes $\bigwedge a.\ a \in A \Longrightarrow \exists\, b{\in}B.\ a \leq b$
shows $\bigsqcup A \leq \bigsqcup B$
⟨*proof*⟩

lemma *SUP-mono*: $(\bigwedge n.\ n \in A \Longrightarrow \exists\, m{\in}B.\ f\ n \leq g\ m) \Longrightarrow (\bigsqcup n{\in}A.\ f\ n) \leq (\bigsqcup n{\in}B.\ g\ n)$
⟨*proof*⟩

lemma *INF-superset-mono*: $B \subseteq A \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\bigsqcap x{\in}A.\ f\ x) \leq (\bigsqcap x{\in}B.\ g\ x)$
— The last inclusion is POSITIVE!
⟨*proof*⟩

lemma *SUP-subset-mono*: $A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\bigsqcup x{\in}A.\ f\ x) \leq (\bigsqcup x{\in}B.\ g\ x)$
⟨*proof*⟩

lemma *Inf-less-eq*:
assumes $\bigwedge v.\ v \in A \Longrightarrow v \leq u$
and $A \neq \{\}$

shows $\bigsqcap A \leq u$
⟨*proof*⟩

lemma *less-eq-Sup*:
assumes $\bigwedge v.\ v \in A \Longrightarrow u \leq v$
and $A \neq \{\}$
shows $u \leq \bigsqcup A$
⟨*proof*⟩

lemma *INF-eq*:
assumes $\bigwedge i.\ i \in A \Longrightarrow \exists j{\in}B.\ f\ i \geq g\ j$
and $\bigwedge j.\ j \in B \Longrightarrow \exists i{\in}A.\ g\ j \geq f\ i$
shows $INFIMUM\ A\ f = INFIMUM\ B\ g$
⟨*proof*⟩

lemma *SUP-eq*:
assumes $\bigwedge i.\ i \in A \Longrightarrow \exists j{\in}B.\ f\ i \leq g\ j$
and $\bigwedge j.\ j \in B \Longrightarrow \exists i{\in}A.\ g\ j \leq f\ i$
shows $SUPREMUM\ A\ f = SUPREMUM\ B\ g$
⟨*proof*⟩

lemma *less-eq-Inf-inter*: $\bigsqcap A \sqcup \bigsqcap B \leq \bigsqcap (A \cap B)$
⟨*proof*⟩

lemma *Sup-inter-less-eq*: $\bigsqcup (A \cap B) \leq \bigsqcup A \sqcap \bigsqcup B$
⟨*proof*⟩

lemma *Inf-union-distrib*: $\bigsqcap (A \cup B) = \bigsqcap A \sqcap \bigsqcap B$
⟨*proof*⟩

lemma *INF-union*: $(\bigsqcap i \in A \cup B.\ M\ i) = (\bigsqcap i \in A.\ M\ i) \sqcap (\bigsqcap i{\in}B.\ M\ i)$
⟨*proof*⟩

lemma *Sup-union-distrib*: $\bigsqcup (A \cup B) = \bigsqcup A \sqcup \bigsqcup B$
⟨*proof*⟩

lemma *SUP-union*: $(\bigsqcup i \in A \cup B.\ M\ i) = (\bigsqcup i \in A.\ M\ i) \sqcup (\bigsqcup i{\in}B.\ M\ i)$
⟨*proof*⟩

lemma *INF-inf-distrib*: $(\bigsqcap a{\in}A.\ f\ a) \sqcap (\bigsqcap a{\in}A.\ g\ a) = (\bigsqcap a{\in}A.\ f\ a \sqcap g\ a)$
⟨*proof*⟩

lemma *SUP-sup-distrib*: $(\bigsqcup a{\in}A.\ f\ a) \sqcup (\bigsqcup a{\in}A.\ g\ a) = (\bigsqcup a{\in}A.\ f\ a \sqcup g\ a)$
(**is** $?L = ?R$)
⟨*proof*⟩

lemma *Inf-top-conv* [*simp*]:
$\bigsqcap A = \top \longleftrightarrow (\forall x{\in}A.\ x = \top)$
$\top = \bigsqcap A \longleftrightarrow (\forall x{\in}A.\ x = \top)$

⟨*proof*⟩

lemma *INF-top-conv* [*simp*]:
$(\bigsqcap x\in A.\ B\ x) = \top \longleftrightarrow (\forall x\in A.\ B\ x = \top)$
$\top = (\bigsqcap x\in A.\ B\ x) \longleftrightarrow (\forall x\in A.\ B\ x = \top)$
⟨*proof*⟩

lemma *Sup-bot-conv* [*simp*]:
$\bigsqcup A = \bot \longleftrightarrow (\forall x\in A.\ x = \bot)$
$\bot = \bigsqcup A \longleftrightarrow (\forall x\in A.\ x = \bot)$
⟨*proof*⟩

lemma *SUP-bot-conv* [*simp*]:
$(\bigsqcup x\in A.\ B\ x) = \bot \longleftrightarrow (\forall x\in A.\ B\ x = \bot)$
$\bot = (\bigsqcup x\in A.\ B\ x) \longleftrightarrow (\forall x\in A.\ B\ x = \bot)$
⟨*proof*⟩

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

lemma *SUP-const* [*simp*]: $A \neq \{\} \Longrightarrow (\bigsqcup i\in A.\ f) = f$
⟨*proof*⟩

lemma *INF-top* [*simp*]: $(\bigsqcap x\in A.\ \top) = \top$
⟨*proof*⟩

lemma *SUP-bot* [*simp*]: $(\bigsqcup x\in A.\ \bot) = \bot$
⟨*proof*⟩

lemma *INF-commute*: $(\bigsqcap i\in A.\ \bigsqcap j\in B.\ f\ i\ j) = (\bigsqcap j\in B.\ \bigsqcap i\in A.\ f\ i\ j)$
⟨*proof*⟩

lemma *SUP-commute*: $(\bigsqcup i\in A.\ \bigsqcup j\in B.\ f\ i\ j) = (\bigsqcup j\in B.\ \bigsqcup i\in A.\ f\ i\ j)$
⟨*proof*⟩

lemma *INF-absorb*:
assumes $k \in I$
shows $A\ k \sqcap (\bigsqcap i\in I.\ A\ i) = (\bigsqcap i\in I.\ A\ i)$
⟨*proof*⟩

lemma *SUP-absorb*:
assumes $k \in I$
shows $A\ k \sqcup (\bigsqcup i\in I.\ A\ i) = (\bigsqcup i\in I.\ A\ i)$
⟨*proof*⟩

lemma *INF-inf-const1*: $I \neq \{\} \Longrightarrow (\mathit{INF}\ i{:}I.\ \mathit{inf}\ x\ (f\ i)) = \mathit{inf}\ x\ (\mathit{INF}\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *INF-inf-const2*: $I \neq \{\} \Longrightarrow (\mathit{INF}\ i{:}I.\ \mathit{inf}\ (f\ i)\ x) = \mathit{inf}\ (\mathit{INF}\ i{:}I.\ f\ i)\ x$

⟨*proof*⟩

lemma *INF-constant*: $(\bigsqcap y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \top\ \mathit{else}\ c)$
⟨*proof*⟩

lemma *SUP-constant*: $(\bigsqcup y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \bot\ \mathit{else}\ c)$
⟨*proof*⟩

lemma *less-INF-D*:
assumes $y < (\bigsqcap i{\in}A.\ f\ i)\ i \in A$
shows $y < f\ i$
⟨*proof*⟩

lemma *SUP-lessD*:
assumes $(\bigsqcup i{\in}A.\ f\ i) < y\ i \in A$
shows $f\ i < y$
⟨*proof*⟩

lemma *INF-UNIV-bool-expand*: $(\bigsqcap b.\ A\ b) = A\ \mathit{True} \sqcap A\ \mathit{False}$
⟨*proof*⟩

lemma *SUP-UNIV-bool-expand*: $(\bigsqcup b.\ A\ b) = A\ \mathit{True} \sqcup A\ \mathit{False}$
⟨*proof*⟩

lemma *Inf-le-Sup*: $A \neq \{\} \Longrightarrow \mathit{Inf}\ A \leq \mathit{Sup}\ A$
⟨*proof*⟩

lemma *INF-le-SUP*: $A \neq \{\} \Longrightarrow \mathit{INFIMUM}\ A\ f \leq \mathit{SUPREMUM}\ A\ f$
⟨*proof*⟩

lemma *INF-eq-const*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i = x) \Longrightarrow \mathit{INFIMUM}\ I\ f = x$
⟨*proof*⟩

lemma *SUP-eq-const*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i = x) \Longrightarrow \mathit{SUPREMUM}\ I\ f = x$
⟨*proof*⟩

lemma *INF-eq-iff*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i \leq c) \Longrightarrow \mathit{INFIMUM}\ I\ f = c \longleftrightarrow (\forall\, i{\in}I.\ f\ i = c)$
⟨*proof*⟩

lemma *SUP-eq-iff*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow c \leq f\ i) \Longrightarrow \mathit{SUPREMUM}\ I\ f = c \longleftrightarrow (\forall\, i{\in}I.\ f\ i = c)$
⟨*proof*⟩

end

class *complete-distrib-lattice* = *complete-lattice* +
assumes *sup-Inf*: $a \sqcup \bigsqcap B = (\bigsqcap b{\in}B.\ a \sqcup b)$

and *inf-Sup*: $a \sqcap \bigsqcup B = (\bigsqcup b{\in}B.\ a \sqcap b)$
begin

lemma *sup-INF*: $a \sqcup (\bigsqcap b{\in}B.\ f\ b) = (\bigsqcap b{\in}B.\ a \sqcup f\ b)$
⟨*proof*⟩

lemma *inf-SUP*: $a \sqcap (\bigsqcup b{\in}B.\ f\ b) = (\bigsqcup b{\in}B.\ a \sqcap f\ b)$
⟨*proof*⟩

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

subclass *distrib-lattice*
⟨*proof*⟩

lemma *Inf-sup*: $\bigsqcap B \sqcup a = (\bigsqcap b{\in}B.\ b \sqcup a)$
⟨*proof*⟩

lemma *Sup-inf*: $\bigsqcup B \sqcap a = (\bigsqcup b{\in}B.\ b \sqcap a)$
⟨*proof*⟩

lemma *INF-sup*: $(\bigsqcap b{\in}B.\ f\ b) \sqcup a = (\bigsqcap b{\in}B.\ f\ b \sqcup a)$
⟨*proof*⟩

lemma *SUP-inf*: $(\bigsqcup b{\in}B.\ f\ b) \sqcap a = (\bigsqcup b{\in}B.\ f\ b \sqcap a)$
⟨*proof*⟩

lemma *Inf-sup-eq-top-iff*: $(\bigsqcap B \sqcup a = \top) \longleftrightarrow (\forall\, b{\in}B.\ b \sqcup a = \top)$
⟨*proof*⟩

lemma *Sup-inf-eq-bot-iff*: $(\bigsqcup B \sqcap a = \bot) \longleftrightarrow (\forall\, b{\in}B.\ b \sqcap a = \bot)$
⟨*proof*⟩

lemma *INF-sup-distrib2*: $(\bigsqcap a{\in}A.\ f\ a) \sqcup (\bigsqcap b{\in}B.\ g\ b) = (\bigsqcap a{\in}A.\ \bigsqcap b{\in}B.\ f\ a \sqcup g\ b)$
⟨*proof*⟩

lemma *SUP-inf-distrib2*: $(\bigsqcup a{\in}A.\ f\ a) \sqcap (\bigsqcup b{\in}B.\ g\ b) = (\bigsqcup a{\in}A.\ \bigsqcup b{\in}B.\ f\ a \sqcap g\ b)$
⟨*proof*⟩

context
fixes f :: $'a \Rightarrow {'b}$::*complete-lattice*
assumes *mono f*
begin

lemma *mono-Inf*: $f\ (\bigsqcap A) \leq (\bigsqcap x{\in}A.\ f\ x)$
⟨*proof*⟩

lemma *mono-Sup*: $(\bigsqcup x \in A.\ f\ x) \leq f\ (\bigsqcup A)$
⟨*proof*⟩

lemma *mono-INF*: $f\ (INF\ i : I.\ A\ i) \leq (INF\ x : I.\ f\ (A\ x))$
⟨*proof*⟩

lemma *mono-SUP*: $(SUP\ x : I.\ f\ (A\ x)) \leq f\ (SUP\ i : I.\ A\ i)$
⟨*proof*⟩

end

end

class *complete-boolean-algebra* = *boolean-algebra* + *complete-distrib-lattice*
begin

lemma *dual-complete-boolean-algebra*:
class.complete-boolean-algebra Sup Inf sup $(op \geq)$ $(op >)$ *inf* $\top$ $\bot$ $(\lambda x\ y.\ x \sqcup -\ y)$ *uminus*
⟨*proof*⟩

lemma *uminus-Inf*: $-\ (\bigsqcap A) = \bigsqcup(uminus\ `\ A)$
⟨*proof*⟩

lemma *uminus-INF*: $-\ (\bigsqcap x \in A.\ B\ x) = (\bigsqcup x \in A.\ -\ B\ x)$
⟨*proof*⟩

lemma *uminus-Sup*: $-\ (\bigsqcup A) = \bigsqcap(uminus\ `\ A)$
⟨*proof*⟩

lemma *uminus-SUP*: $-\ (\bigsqcup x \in A.\ B\ x) = (\bigsqcap x \in A.\ -\ B\ x)$
⟨*proof*⟩

end

class *complete-linorder* = *linorder* + *complete-lattice*
begin

lemma *dual-complete-linorder*:
class.complete-linorder Sup Inf sup $(op \geq)$ $(op >)$ *inf* $\top$ $\bot$
⟨*proof*⟩

lemma *complete-linorder-inf-min*: *inf* = *min*
⟨*proof*⟩

lemma *complete-linorder-sup-max*: *sup* = *max*
⟨*proof*⟩

lemma *Inf-less-iff*: $\bigsqcap S < a \longleftrightarrow (\exists x \in S.\ x < a)$
⟨*proof*⟩

lemma *INF-less-iff*: $(\bigsqcap i \in A.\ f\ i) < a \longleftrightarrow (\exists x \in A.\ f\ x < a)$
⟨*proof*⟩

lemma *less-Sup-iff*: $a < \bigsqcup S \longleftrightarrow (\exists x \in S.\ a < x)$
⟨*proof*⟩

lemma *less-SUP-iff*: $a < (\bigsqcup i \in A.\ f\ i) \longleftrightarrow (\exists x \in A.\ a < f\ x)$
⟨*proof*⟩

lemma *Sup-eq-top-iff* [*simp*]: $\bigsqcup A = \top \longleftrightarrow (\forall x < \top.\ \exists i \in A.\ x < i)$
⟨*proof*⟩

lemma *SUP-eq-top-iff* [*simp*]: $(\bigsqcup i \in A.\ f\ i) = \top \longleftrightarrow (\forall x < \top.\ \exists i \in A.\ x < f\ i)$
⟨*proof*⟩

lemma *Inf-eq-bot-iff* [*simp*]: $\bigsqcap A = \bot \longleftrightarrow (\forall x > \bot.\ \exists i \in A.\ i < x)$
⟨*proof*⟩

lemma *INF-eq-bot-iff* [*simp*]: $(\bigsqcap i \in A.\ f\ i) = \bot \longleftrightarrow (\forall x > \bot.\ \exists i \in A.\ f\ i < x)$
⟨*proof*⟩

lemma *Inf-le-iff*: $\bigsqcap A \leq x \longleftrightarrow (\forall y > x.\ \exists a \in A.\ y > a)$
⟨*proof*⟩

lemma *INF-le-iff*: $\mathit{INFIMUM}\ A\ f \leq x \longleftrightarrow (\forall y > x.\ \exists i \in A.\ y > f\ i)$
⟨*proof*⟩

lemma *le-Sup-iff*: $x \leq \bigsqcup A \longleftrightarrow (\forall y < x.\ \exists a \in A.\ y < a)$
⟨*proof*⟩

lemma *le-SUP-iff*: $x \leq \mathit{SUPREMUM}\ A\ f \longleftrightarrow (\forall y < x.\ \exists i \in A.\ y < f\ i)$
⟨*proof*⟩

subclass *complete-distrib-lattice*
⟨*proof*⟩

end

10.3 Complete lattice on *bool*

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

definition [*simp*, *code*]: $\bigsqcap A \longleftrightarrow \mathit{False} \notin A$

definition [*simp*, *code*]: $\bigsqcup A \longleftrightarrow \mathit{True} \in A$

instance
⟨*proof*⟩

end

lemma *not-False-in-image-Ball* [*simp*]: $False \notin P$ ‘ $A \longleftrightarrow Ball\ A\ P$
⟨*proof*⟩

lemma *True-in-image-Bex* [*simp*]: $True \in P$ ‘ $A \longleftrightarrow Bex\ A\ P$
⟨*proof*⟩

lemma *INF-bool-eq* [*simp*]: $INFIMUM = Ball$
⟨*proof*⟩

lemma *SUP-bool-eq* [*simp*]: $SUPREMUM = Bex$
⟨*proof*⟩

instance *bool* :: *complete-boolean-algebra*
⟨*proof*⟩

10.4 Complete lattice on - ⇒ -

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

definition $\bigsqcap A = (\lambda x. \bigsqcap f{\in}A.\ f\ x)$

lemma *Inf-apply* [*simp*, *code*]: $(\bigsqcap A)\ x = (\bigsqcap f{\in}A.\ f\ x)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

definition $\bigsqcup A = (\lambda x. \bigsqcup f{\in}A.\ f\ x)$

lemma *Sup-apply* [*simp*, *code*]: $(\bigsqcup A)\ x = (\bigsqcup f{\in}A.\ f\ x)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

instance
⟨*proof*⟩

end

lemma *INF-apply* [*simp*]: $(\prod y{\in}A.\ f\ y)\ x = (\prod y{\in}A.\ f\ y\ x)$
⟨*proof*⟩

lemma *SUP-apply* [*simp*]: $(\bigsqcup y{\in}A.\ f\ y)\ x = (\bigsqcup y{\in}A.\ f\ y\ x)$
⟨*proof*⟩

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

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

10.5 Complete lattice on unary and binary predicates

lemma *Inf1-I*: $(\bigwedge P.\ P \in A \Longrightarrow P\ a) \Longrightarrow (\prod A)\ a$
⟨*proof*⟩

lemma *INF1-I*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x\ b) \Longrightarrow (\prod x{\in}A.\ B\ x)\ b$
⟨*proof*⟩

lemma *INF2-I*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x\ b\ c) \Longrightarrow (\prod x{\in}A.\ B\ x)\ b\ c$
⟨*proof*⟩

lemma *Inf2-I*: $(\bigwedge r.\ r \in A \Longrightarrow r\ a\ b) \Longrightarrow (\prod A)\ a\ b$
⟨*proof*⟩

lemma *Inf1-D*: $(\prod A)\ a \Longrightarrow P \in A \Longrightarrow P\ a$
⟨*proof*⟩

lemma *INF1-D*: $(\prod x{\in}A.\ B\ x)\ b \Longrightarrow a \in A \Longrightarrow B\ a\ b$
⟨*proof*⟩

lemma *Inf2-D*: $(\prod A)\ a\ b \Longrightarrow r \in A \Longrightarrow r\ a\ b$
⟨*proof*⟩

lemma *INF2-D*: $(\prod x{\in}A.\ B\ x)\ b\ c \Longrightarrow a \in A \Longrightarrow B\ a\ b\ c$
⟨*proof*⟩

lemma *Inf1-E*:
assumes $(\prod A)\ a$
obtains $P\ a \mid P \notin A$
⟨*proof*⟩

lemma *INF1-E*:

assumes $(\bigsqcap x{\in}A.\ B\ x)\ b$
obtains $B\ a\ b \mid a \notin A$
⟨*proof*⟩

lemma *Inf2-E*:
assumes $(\bigsqcap A)\ a\ b$
obtains $r\ a\ b \mid r \notin A$
⟨*proof*⟩

lemma *INF2-E*:
assumes $(\bigsqcap x{\in}A.\ B\ x)\ b\ c$
obtains $B\ a\ b\ c \mid a \notin A$
⟨*proof*⟩

lemma *Sup1-I*: $P \in A \Longrightarrow P\ a \Longrightarrow (\bigsqcup A)\ a$
⟨*proof*⟩

lemma *SUP1-I*: $a \in A \Longrightarrow B\ a\ b \Longrightarrow (\bigsqcup x{\in}A.\ B\ x)\ b$
⟨*proof*⟩

lemma *Sup2-I*: $r \in A \Longrightarrow r\ a\ b \Longrightarrow (\bigsqcup A)\ a\ b$
⟨*proof*⟩

lemma *SUP2-I*: $a \in A \Longrightarrow B\ a\ b\ c \Longrightarrow (\bigsqcup x{\in}A.\ B\ x)\ b\ c$
⟨*proof*⟩

lemma *Sup1-E*:
assumes $(\bigsqcup A)\ a$
obtains P **where** $P \in A$ **and** $P\ a$
⟨*proof*⟩

lemma *SUP1-E*:
assumes $(\bigsqcup x{\in}A.\ B\ x)\ b$
obtains x **where** $x \in A$ **and** $B\ x\ b$
⟨*proof*⟩

lemma *Sup2-E*:
assumes $(\bigsqcup A)\ a\ b$
obtains r **where** $r \in A\ r\ a\ b$
⟨*proof*⟩

lemma *SUP2-E*:
assumes $(\bigsqcup x{\in}A.\ B\ x)\ b\ c$
obtains x **where** $x \in A\ B\ x\ b\ c$
⟨*proof*⟩

10.6 Complete lattice on - *set*

instantiation *set* :: (*type*) *complete-lattice*

begin

definition $\bigsqcap A = \{x.\ \bigsqcap((\lambda B.\ x \in B)\ `\ A)\}$

definition $\bigsqcup A = \{x.\ \bigsqcup((\lambda B.\ x \in B)\ `\ A)\}$

instance
⟨*proof*⟩

end

instance *set* :: (*type*) *complete-boolean-algebra*
⟨*proof*⟩

10.6.1 Inter

abbreviation *Inter* :: $'a\ set\ set \Rightarrow\ 'a\ set$ ($\bigcap$ - [900] 900)
where $\bigcap S \equiv \bigsqcap S$

lemma *Inter-eq*: $\bigcap A = \{x.\ \forall B \in A.\ x \in B\}$
⟨*proof*⟩

lemma *Inter-iff* [*simp*]: $A \in \bigcap C \longleftrightarrow (\forall X{\in}C.\ A \in X)$
⟨*proof*⟩

lemma *InterI* [*intro*!]: $(\bigwedge X.\ X \in C \Longrightarrow A \in X) \Longrightarrow A \in \bigcap C$
⟨*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*, *Pure.elim*]: $A \in \bigcap C \Longrightarrow X \in C \Longrightarrow A \in X$
⟨*proof*⟩

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

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

lemma *Inter-subset*: $(\bigwedge X.\ X \in A \Longrightarrow X \subseteq B) \Longrightarrow A \neq \{\} \Longrightarrow \bigcap A \subseteq B$
⟨*proof*⟩

lemma *Inter-greatest*: $(\bigwedge X.\ X \in A \Longrightarrow C \subseteq X) \Longrightarrow C \subseteq \bigcap A$
⟨*proof*⟩

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

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

lemma *Inter-insert*: $\bigcap$(*insert a B*) = *a* ∩ $\bigcap$ *B*
⟨*proof*⟩

lemma *Inter-Un-subset*: $\bigcap$ *A* ∪ $\bigcap$ *B* ⊆ $\bigcap$(*A* ∩ *B*)
⟨*proof*⟩

lemma *Inter-Un-distrib*: $\bigcap$(*A* ∪ *B*) = $\bigcap$ *A* ∩ $\bigcap$ *B*
⟨*proof*⟩

lemma *Inter-UNIV-conv* [*simp*]:
$\bigcap$ *A* = *UNIV* ⟷ (∀ *x*∈*A*. *x* = *UNIV*)
UNIV = $\bigcap$ *A* ⟷ (∀ *x*∈*A*. *x* = *UNIV*)
⟨*proof*⟩

lemma *Inter-anti-mono*: *B* ⊆ *A* ⟹ $\bigcap$ *A* ⊆ $\bigcap$ *B*
⟨*proof*⟩

10.6.2 Intersections of families

abbreviation *INTER* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *'b set*
where *INTER* ≡ *INFIMUM*

Note: must use name *INTER* here instead of *INT* to allow the following syntax coexist with the plain constant name.

syntax (*ASCII*)
-INTER1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* ((*3INT* -./ -) [*0*, *10*] *10*)
-INTER :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* ((*3INT* -:-./ -) [*0*, *0*, *10*] *10*)

syntax (*latex* **output**)
-INTER1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* ((*3*$\bigcap$(‹*unbreakable*›-)/ -) [*0*, *10*] *10*)
-INTER :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* ((*3*$\bigcap$(‹*unbreakable*›-∈-)/ -) [*0*, *0*, *10*] *10*)

syntax
-INTER1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* ((*3*$\bigcap$-./ -) [*0*, *10*] *10*)
-INTER :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* ((*3*$\bigcap$-∈-./ -) [*0*, *0*, *10*] *10*)

translations
$\bigcap$ *x y*. *B* ⇌ $\bigcap$ *x*. $\bigcap$ *y*. *B*
$\bigcap$ *x*. *B* ⇌ *CONST INTER CONST UNIV* (λ*x*. *B*)
$\bigcap$ *x*. *B* ⇌ $\bigcap$ *x* ∈ *CONST UNIV*. *B*
$\bigcap$ *x*∈*A*. *B* ⇌ *CONST INTER A* (λ*x*. *B*)

⟨*ML*⟩

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

lemma *INT-iff* [*simp*]: $b \in (\bigcap x\in A.\ B\ x) \longleftrightarrow (\forall x\in A.\ b \in B\ x)$
⟨*proof*⟩

lemma *INT-I* [*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow b \in B\ x) \Longrightarrow b \in (\bigcap x\in A.\ B\ x)$
⟨*proof*⟩

lemma *INT-D* [*elim*, *Pure.elim*]: $b \in (\bigcap x\in A.\ B\ x) \Longrightarrow a \in A \Longrightarrow b \in B\ a$
⟨*proof*⟩

lemma *INT-E* [*elim*]: $b \in (\bigcap x\in A.\ B\ x) \Longrightarrow (b \in B\ a \Longrightarrow R) \Longrightarrow (a \notin A \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination – by the Excluded Middle on $a \in A$.
⟨*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*: $(\bigwedge x.\ x \in A \Longrightarrow C \subseteq B\ x) \Longrightarrow C \subseteq (\bigcap x\in A.\ B\ x)$
⟨*proof*⟩

lemma *INT-empty*: $(\bigcap x\in\{\}.\ B\ x) = UNIV$
⟨*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*: $B \subseteq (\bigcap i\in I.\ A\ i) \longleftrightarrow (\forall i\in I.\ B \subseteq A\ i)$
⟨*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 *INTER-UNIV-conv*:
$(UNIV = (\bigcap x{\in}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = UNIV)$
$((\bigcap x{\in}A.\ B\ x) = UNIV) = (\forall x{\in}A.\ B\ x = UNIV)$
⟨*proof*⟩

lemma *INT-bool-eq*: $(\bigcap b.\ A\ b) = A\ True \cap A\ False$
⟨*proof*⟩

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

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

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

10.6.3 Union

abbreviation *Union* :: $'a\ set\ set \Rightarrow 'a\ set$ $(\bigcup$- $[900]\ 900)$
where $\bigcup S \equiv \bigsqcup S$

lemma *Union-eq*: $\bigcup A = \{x.\ \exists B \in A.\ x \in B\}$
⟨*proof*⟩

lemma *Union-iff* [*simp*]: $A \in \bigcup C \longleftrightarrow (\exists X{\in}C.\ A{\in}X)$
⟨*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*: $B \in A \Longrightarrow B \subseteq \bigcup A$
⟨*proof*⟩

lemma *Union-least*: $(\bigwedge X.\ X \in A \Longrightarrow X \subseteq C) \Longrightarrow \bigcup A \subseteq C$
⟨*proof*⟩

lemma *Union-empty*: $\bigcup \{\} = \{\}$
⟨*proof*⟩

lemma *Union-UNIV*: $\bigcup UNIV = UNIV$

⟨*proof*⟩

lemma *Union-insert*: $\bigcup insert\ a\ B = a \cup \bigcup B$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *Union-subsetI*: $(\bigwedge x.\ x \in A \Longrightarrow \exists y.\ y \in B \wedge x \subseteq y) \Longrightarrow \bigcup A \subseteq \bigcup B$
⟨*proof*⟩

lemma *disjnt-inj-on-iff*:
$[\![inj\text{-}on\ f\ (\bigcup \mathcal{A});\ X \in \mathcal{A};\ Y \in \mathcal{A}]\!] \Longrightarrow disjnt\ (f\ `\ X)\ (f\ `\ Y) \longleftrightarrow disjnt\ X\ Y$
⟨*proof*⟩

10.6.4 Unions of families

abbreviation *UNION* :: $'a\ set \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'b\ set$
where *UNION* ≡ *SUPREMUM*

Note: must use name *UNION* here instead of *UN* to allow the following syntax coexist with the plain constant name.

syntax (*ASCII*)
-UNION1 :: *pttrns => 'b set => 'b set* ((*3UN -./ -*) [*0*, *10*] *10*)
-UNION :: *pttrn => 'a set => 'b set => 'b set* ((*3UN -:-./ -*) [*0*, *0*, *10*] *10*)

syntax (*latex* **output**)
-UNION1 :: *pttrns => 'b set => 'b set* ((*3*$\bigcup$(‹*unbreakable*›$_-$)/ -) [*0*, *10*] *10*)
-UNION :: *pttrn => 'a set => 'b set => 'b set* ((*3*$\bigcup$(‹*unbreakable*›$_{-\in-}$)/ -)

$[0, 0, 10]$ $10)$

syntax
-UNION1 :: *pttrns => 'b set => 'b set* $((3\bigcup$ -./ -) $[0, 10]$ $10)$
-UNION :: *pttrn => 'a set => 'b set => 'b set* $((3\bigcup$ -∈-./ -) $[0, 0, 10]$ $10)$

translations
$\bigcup x\ y.\ B \rightleftharpoons \bigcup x.\ \bigcup y.\ B$
$\bigcup x.\ B \rightleftharpoons CONST\ UNION\ CONST\ UNIV\ (\lambda x.\ B)$
$\bigcup x.\ B \rightleftharpoons \bigcup x \in CONST\ UNIV.\ B$
$\bigcup x{\in}A.\ B \rightleftharpoons CONST\ UNION\ A\ (\lambda x.\ B)$

Note the difference between ordinary syntax of indexed unions and intersections (e.g. $\bigcup a_1{\in}A_1.\ B$) and their LaTeX rendition: $\bigcup_{a_1 \in A_1} B$.

⟨*ML*⟩

lemma *UNION-eq*: $(\bigcup x{\in}A.\ B\ x) = \{y.\ \exists x{\in}A.\ y \in B\ x\}$
⟨*proof*⟩

lemma *bind-UNION* [*code*]: *Set.bind A f = UNION A f*
⟨*proof*⟩

lemma *member-bind* [*simp*]: $x \in Set.bind\ P\ f \longleftrightarrow x \in UNION\ P\ f$
⟨*proof*⟩

lemma *Union-SetCompr-eq*: $\bigcup \{f\ x|\ x.\ P\ x\} = \{a.\ \exists x.\ P\ x \wedge a \in f\ x\}$
⟨*proof*⟩

lemma *UN-iff* [*simp*]: $b \in (\bigcup x{\in}A.\ B\ x) \longleftrightarrow (\exists x{\in}A.\ b \in B\ x)$
⟨*proof*⟩

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

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

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

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

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

lemma *UN-empty2*: $(\bigcup x\in A.\ \{\}) = \{\}$
⟨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 *UN-constant* [*simp*]: $(\bigcup y\in A.\ c) = (if\ A = \{\}\ then\ \{\}\ else\ c)$
⟨proof⟩

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

lemma *UNION-empty-conv*:
$\{\} = (\bigcup x\in A.\ B\ x) \longleftrightarrow (\forall\, x\in A.\ B\ x = \{\})$
$(\bigcup x\in A.\ B\ x) = \{\} \longleftrightarrow (\forall\, x\in A.\ B\ x = \{\})$
⟨proof⟩

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

lemma *ball-UN*: $(\forall\, z \in UNION\ A\ B.\ P\ z) \longleftrightarrow (\forall\, x\in A.\ \forall\, z \in B\ x.\ P\ z)$
⟨proof⟩

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

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

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

lemma *UN-Pow-subset*: $(\bigcup x{\in}A.\ Pow\ (B\ x)) \subseteq Pow\ (\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

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

lemma *vimage-Union*: $f\ -`\ (\bigcup A) = (\bigcup X{\in}A.\ f\ -`\ X)$
⟨*proof*⟩

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

lemma *vimage-eq-UN*: $f\ -`\ B = (\bigcup y{\in}B.\ f\ -`\ \{y\})$
— NOT suitable for rewriting
⟨*proof*⟩

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

lemma *UN-singleton* [*simp*]: $(\bigcup x{\in}A.\ \{x\}) = A$
⟨*proof*⟩

lemma *inj-on-image*: $inj\text{-}on\ f\ (\bigcup A) \Longrightarrow inj\text{-}on\ (op\ `\ f)\ A$
⟨*proof*⟩

10.6.5 Distributive laws

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

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap C{\in}B.\ A \cup C)$
⟨*proof*⟩

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C{\in}B.\ C \cap A)$
⟨*proof*⟩

lemma *INT-Int-distrib*: $(\bigcap i{\in}I.\ A\ i \cap B\ i) = (\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i)$
⟨*proof*⟩

lemma *UN-Un-distrib*: $(\bigcup i{\in}I.\ A\ i \cup B\ i) = (\bigcup i{\in}I.\ A\ i) \cup (\bigcup i{\in}I.\ B\ i)$
⟨*proof*⟩

lemma *Int-Inter-image*: $(\bigcap x{\in}C.\ A\ x \cap B\ x) = \bigcap (A\ `\ C) \cap \bigcap (B\ `\ C)$
⟨*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-INT-distrib*: $B \cup (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ B \cup A\ i)$
⟨*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 *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*⟩

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

lemma *SUP-UNION*: $(SUP\ x{:}(UN\ y{:}A.\ g\ y).\ f\ x) = (SUP\ y{:}A.\ SUP\ x{:}g\ y.\ f\ x ::$ - :: *complete-lattice*)
⟨*proof*⟩

10.7 Injections and bijections

lemma *inj-on-Inter*: $S \neq \{\} \Longrightarrow (\bigwedge A.\ A \in S \Longrightarrow inj\text{-}on\ f\ A) \Longrightarrow inj\text{-}on\ f\ (\bigcap S)$
⟨*proof*⟩

lemma *inj-on-INTER*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow inj\text{-}on\ f\ (A\ i)) \Longrightarrow inj\text{-}on\ f\ (\bigcap i \in I.\ A\ i)$
⟨*proof*⟩

lemma *inj-on-UNION-chain*:
assumes *chain*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow A\ i \leq A\ j \vee A\ j \leq A\ i$
and *inj*: $\bigwedge i.\ i \in I \Longrightarrow inj\text{-}on\ f\ (A\ i)$
shows $inj\text{-}on\ f\ (\bigcup i \in I.\ A\ i)$
⟨*proof*⟩

lemma *bij-betw-UNION-chain*:
assumes *chain*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow A\ i \leq A\ j \vee A\ j \leq A\ i$
and *bij*: $\bigwedge i.\ i \in I \Longrightarrow bij\text{-}betw\ f\ (A\ i)\ (A'\ i)$
shows $bij\text{-}betw\ f\ (\bigcup i \in I.\ A\ i)\ (\bigcup i \in I.\ A'\ i)$
⟨*proof*⟩

lemma *image-INT*: $inj\text{-}on\ f\ C \Longrightarrow \forall x{\in}A.\ B\ x \subseteq C \Longrightarrow j \in A \Longrightarrow f$ ‘ (*INTER*

$A\ B) = (INT\ x{:}A.\ f\ `\ B\ x)$
⟨proof⟩

lemma *bij-image-INT*: $bij\ f \Longrightarrow f\ `\ (INTER\ A\ B) = (INT\ x{:}A.\ f\ `\ B\ x)$
⟨proof⟩

lemma *UNION-fun-upd*: $UNION\ J\ (A(i := B)) = UNION\ (J - \{i\})\ A \cup (if\ i \in J\ then\ B\ else\ \{\})$
⟨proof⟩

lemma *bij-betw-Pow*:
assumes *bij-betw f A B*
shows *bij-betw (image f) (Pow A) (Pow B)*
⟨proof⟩

10.7.1 Complement

lemma *Compl-INT* [*simp*]: $- (\bigcap x{\in}A.\ B\ x) = (\bigcup x{\in}A.\ -B\ x)$
⟨proof⟩

lemma *Compl-UN* [*simp*]: $- (\bigcup x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ -B\ x)$
⟨proof⟩

10.7.2 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps* [*simp*]:
$\bigwedge a\ B\ C.\ (\bigcup x{\in}C.\ insert\ a\ (B\ x)) = (if\ C{=}\{\}\ then\ \{\}\ else\ insert\ a\ (\bigcup x{\in}C.\ B\ x))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x \cup B) = ((if\ C{=}\{\}\ then\ \{\}\ else\ (\bigcup x{\in}C.\ A\ x) \cup B))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A \cup B\ x) = ((if\ C{=}\{\}\ then\ \{\}\ else\ A \cup (\bigcup x{\in}C.\ B\ x)))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x \cap B) = ((\bigcup x{\in}C.\ A\ x) \cap B)$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A \cap B\ x) = (A \cap (\bigcup x{\in}C.\ B\ x))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x - B) = ((\bigcup x{\in}C.\ A\ x) - B)$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A - B\ x) = (A - (\bigcap x{\in}C.\ B\ x))$
$\bigwedge A\ B.\ (\bigcup x{\in}\bigcup A.\ B\ x) = (\bigcup y{\in}A.\ \bigcup x{\in}y.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcup z{\in}UNION\ A\ B.\ C\ z) = (\bigcup x{\in}A.\ \bigcup z{\in}B\ x.\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcup x{\in}f`A.\ B\ x) = (\bigcup a{\in}A.\ B\ (f\ a))$
⟨proof⟩

lemma *INT-simps* [*simp*]:
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x \cap B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (\bigcap x{\in}C.\ A\ x) \cap B)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A \cap B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A \cap (\bigcap x{\in}C.\ B\ x))$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x - B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (\bigcap x{\in}C.\ A\ x) - B)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A - B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A - (\bigcup x{\in}C.\ B\ x))$
$\bigwedge a\ B\ C.\ (\bigcap x{\in}C.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcap x{\in}C.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x \cup B) = ((\bigcap x{\in}C.\ A\ x) \cup B)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A \cup B\ x) = (A \cup (\bigcap x{\in}C.\ B\ x))$
$\bigwedge A\ B.\ (\bigcap x{\in}\bigcup A.\ B\ x) = (\bigcap y{\in}A.\ \bigcap x{\in}y.\ B\ x)$

$\bigwedge A\ B\ C.\ (\bigcap z{\in}UNION\ A\ B.\ C\ z) = (\bigcap x{\in}A.\ \bigcap z{\in}B\ x.\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcap x{\in}f`A.\ B\ x) = (\bigcap a{\in}A.\ B\ (f\ a))$
⟨*proof*⟩

lemma *UN-ball-bex-simps* [*simp*]:
$\bigwedge A\ P.\ (\forall x{\in}\bigcup A.\ P\ x) \longleftrightarrow (\forall y{\in}A.\ \forall x{\in}y.\ P\ x)$
$\bigwedge A\ B\ P.\ (\forall x{\in}UNION\ A\ B.\ P\ x) = (\forall a{\in}A.\ \forall x{\in}\ B\ a.\ P\ x)$
$\bigwedge A\ P.\ (\exists x{\in}\bigcup A.\ P\ x) \longleftrightarrow (\exists y{\in}A.\ \exists x{\in}y.\ P\ x)$
$\bigwedge A\ B\ P.\ (\exists x{\in}UNION\ A\ B.\ P\ x) \longleftrightarrow (\exists a{\in}A.\ \exists x{\in}B\ a.\ P\ x)$
⟨*proof*⟩

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:
$\bigwedge a\ B\ C.\ insert\ a\ (\bigcup x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ \{a\}\ else\ (\bigcup x{\in}C.\ insert\ a\ (B\ x)))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x) \cup B = (if\ C{=}\{\}\ then\ B\ else\ (\bigcup x{\in}C.\ A\ x \cup B))$
$\bigwedge A\ B\ C.\ A \cup (\bigcup x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (\bigcup x{\in}C.\ A \cup B\ x))$
$\bigwedge A\ B\ C.\ ((\bigcup x{\in}C.\ A\ x) \cap B) = (\bigcup x{\in}C.\ A\ x \cap B)$
$\bigwedge A\ B\ C.\ (A \cap (\bigcup x{\in}C.\ B\ x)) = (\bigcup x{\in}C.\ A \cap B\ x)$
$\bigwedge A\ B\ C.\ ((\bigcup x{\in}C.\ A\ x) - B) = (\bigcup x{\in}C.\ A\ x - B)$
$\bigwedge A\ B\ C.\ (A - (\bigcap x{\in}C.\ B\ x)) = (\bigcup x{\in}C.\ A - B\ x)$
$\bigwedge A\ B.\ (\bigcup y{\in}A.\ \bigcup x{\in}y.\ B\ x) = (\bigcup x{\in}\bigcup A.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}A.\ \bigcup z{\in}B\ x.\ C\ z) = (\bigcup z{\in}UNION\ A\ B.\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcup a{\in}A.\ B\ (f\ a)) = (\bigcup x{\in}f`A.\ B\ x)$
⟨*proof*⟩

lemma *INT-extend-simps*:
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x) \cap B = (if\ C{=}\{\}\ then\ B\ else\ (\bigcap x{\in}C.\ A\ x \cap B))$
$\bigwedge A\ B\ C.\ A \cap (\bigcap x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (\bigcap x{\in}C.\ A \cap B\ x))$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x) - B = (if\ C{=}\{\}\ then\ UNIV - B\ else\ (\bigcap x{\in}C.\ A\ x - B))$
$\bigwedge A\ B\ C.\ A - (\bigcup x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (\bigcap x{\in}C.\ A - B\ x))$
$\bigwedge a\ B\ C.\ insert\ a\ (\bigcap x{\in}C.\ B\ x) = (\bigcap x{\in}C.\ insert\ a\ (B\ x))$
$\bigwedge A\ B\ C.\ ((\bigcap x{\in}C.\ A\ x) \cup B) = (\bigcap x{\in}C.\ A\ x \cup B)$
$\bigwedge A\ B\ C.\ A \cup (\bigcap x{\in}C.\ B\ x) = (\bigcap x{\in}C.\ A \cup B\ x)$
$\bigwedge A\ B.\ (\bigcap y{\in}A.\ \bigcap x{\in}y.\ B\ x) = (\bigcap x{\in}\bigcup A.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}A.\ \bigcap z{\in}B\ x.\ C\ z) = (\bigcap z{\in}UNION\ A\ B.\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcap a{\in}A.\ B\ (f\ a)) = (\bigcap x{\in}f`A.\ B\ x)$
⟨*proof*⟩

Finally

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

11 Wrapping Existing Freely Generated Type's Constructors

theory *Ctr-Sugar*
imports *HOL*
keywords
print-case-translations :: *diag* **and**
free-constructors :: *thy-goal*
begin

consts
$case\text{-}guard :: bool \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$
$case\text{-}nil :: 'a \Rightarrow 'b$
$case\text{-}cons :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
$case\text{-}elem :: 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b$
$case\text{-}abs :: ('c \Rightarrow 'b) \Rightarrow 'b$

declare [[*coercion-args case-guard* − + −]]
declare [[*coercion-args case-cons* − −]]
declare [[*coercion-args case-abs* −]]
declare [[*coercion-args case-elem* − +]]

⟨*ML*⟩

lemma *iffI-np*: $[\![x \Longrightarrow \neg\, y;\ \neg\, x \Longrightarrow y]\!] \Longrightarrow \neg\, x \longleftrightarrow y$
⟨*proof*⟩

lemma *iff-contradict*:
$\neg\, P \Longrightarrow P \longleftrightarrow Q \Longrightarrow Q \Longrightarrow R$
$\neg\, Q \Longrightarrow P \longleftrightarrow Q \Longrightarrow P \Longrightarrow R$
⟨*proof*⟩

⟨*ML*⟩

Coinduction method that avoids some boilerplate compared with coinduct.

⟨*ML*⟩

end

12 Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Complete-Lattices Ctr-Sugar*
keywords
inductive coinductive inductive-cases inductive-simps :: *thy-decl* **and**
monos **and**
print-inductives :: *diag* **and**

old-rep-datatype :: *thy-goal* **and**
primrec :: *thy-decl*
begin

12.1 Least fixed points

context *complete-lattice*
begin

definition $lfp :: ('a \Rightarrow 'a) \Rightarrow 'a$
where $lfp\ f = Inf\ \{u.\ f\ u \leq u\}$

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

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

end

lemma *lfp-fixpoint*:
assumes *mono f*
shows $f\ (lfp\ f) = lfp\ f$
⟨*proof*⟩

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

lemma *lfp-const*: $lfp\ (\lambda x.\ t) = t$
⟨*proof*⟩

lemma *lfp-eqI*: $mono\ F \Longrightarrow F\ x = x \Longrightarrow (\bigwedge z.\ F\ z = z \Longrightarrow x \leq z) \Longrightarrow lfp\ F = x$
⟨*proof*⟩

12.2 General induction rules for least fixed points

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

theorem *lfp-induct*:
assumes *mono*: *mono f*
and *ind*: $f\ (inf\ (lfp\ f)\ P) \leq P$
shows $lfp\ f \leq P$
⟨*proof*⟩

lemma *lfp-induct-set*:
assumes *lfp*: $a \in lfp\ f$
and *mono*: *mono f*
and *hyp*: $\bigwedge x.\ x \in f\ (lfp\ f \cap \{x.\ P\ x\}) \Longrightarrow P\ x$
shows $P\ a$
⟨*proof*⟩

lemma *lfp-ordinal-induct-set*:
assumes *mono*: *mono f*
and *P-f*: $\bigwedge S.\ P\ S \Longrightarrow P\ (f\ S)$
and *P-Union*: $\bigwedge M.\ \forall S{\in}M.\ P\ S \Longrightarrow P\ (\bigcup M)$
shows $P\ (lfp\ f)$
⟨*proof*⟩

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding.

lemma *def-lfp-unfold*: $h \equiv lfp\ f \Longrightarrow mono\ f \Longrightarrow h = f\ h$
⟨*proof*⟩

lemma *def-lfp-induct*: $A \equiv lfp\ f \Longrightarrow mono\ f \Longrightarrow f\ (inf\ A\ P) \leq P \Longrightarrow A \leq P$
⟨*proof*⟩

lemma *def-lfp-induct-set*:
$A \equiv lfp\ f \Longrightarrow mono\ f \Longrightarrow a \in A \Longrightarrow (\bigwedge x.\ x \in f\ (A \cap \{x.\ P\ x\}) \Longrightarrow P\ x) \Longrightarrow P\ a$
⟨*proof*⟩

Monotonicity of *lfp*!

lemma *lfp-mono*: $(\bigwedge Z.\ f\ Z \leq g\ Z) \Longrightarrow lfp\ f \leq lfp\ g$
⟨*proof*⟩

12.3 Greatest fixed points

context *complete-lattice*
begin

definition $gfp :: ('a \Rightarrow 'a) \Rightarrow 'a$
where $gfp\ f = Sup\ \{u.\ u \leq f\ u\}$

lemma *gfp-upperbound*: $X \leq f\ X \Longrightarrow X \leq gfp\ f$
⟨*proof*⟩

lemma *gfp-least*: $(\bigwedge u.\ u \leq f\ u \Longrightarrow u \leq X) \Longrightarrow gfp\ f \leq X$
⟨*proof*⟩

end

lemma *lfp-le-gfp*: $mono\ f \Longrightarrow lfp\ f \leq gfp\ f$
⟨*proof*⟩

lemma *gfp-fixpoint*:
  assumes *mono f*
  shows $f\ (gfp\ f) = gfp\ f$
  ⟨*proof*⟩

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

lemma *gfp-const*: $gfp\ (\lambda x.\ t) = t$
  ⟨*proof*⟩

lemma *gfp-eqI*: $mono\ F \Longrightarrow F\ x = x \Longrightarrow (\bigwedge z.\ F\ z = z \Longrightarrow z \le x) \Longrightarrow gfp\ F = x$
  ⟨*proof*⟩

12.4 Coinduction rules for greatest fixed points

Weak version.

lemma *weak-coinduct*: $a \in X \Longrightarrow X \subseteq f\ X \Longrightarrow a \in gfp\ f$
  ⟨*proof*⟩

lemma *weak-coinduct-image*: $a \in X \Longrightarrow g`X \subseteq f\ (g`X) \Longrightarrow g\ a \in gfp\ f$
  ⟨*proof*⟩

lemma *coinduct-lemma*: $X \le f\ (sup\ X\ (gfp\ f)) \Longrightarrow mono\ f \Longrightarrow sup\ X\ (gfp\ f) \le f\ (sup\ X\ (gfp\ f))$
  ⟨*proof*⟩

Strong version, thanks to Coen and Frost.

lemma *coinduct-set*: $mono\ f \Longrightarrow a \in X \Longrightarrow X \subseteq f\ (X \cup gfp\ f) \Longrightarrow a \in gfp\ f$
  ⟨*proof*⟩

lemma *gfp-fun-UnI2*: $mono\ f \Longrightarrow a \in gfp\ f \Longrightarrow a \in f\ (X \cup gfp\ f)$
  ⟨*proof*⟩

lemma *gfp-ordinal-induct*[*case-names mono step union*]:
  fixes $f :: 'a{::}complete\text{-}lattice \Rightarrow 'a$
  assumes *mono*: *mono f*
    and *P-f*: $\bigwedge S.\ P\ S \Longrightarrow gfp\ f \le S \Longrightarrow P\ (f\ S)$
    and *P-Union*: $\bigwedge M.\ \forall S{\in}M.\ P\ S \Longrightarrow P\ (Inf\ M)$
  shows $P\ (gfp\ f)$
⟨*proof*⟩

lemma *coinduct*:
  assumes *mono*: *mono f*
    and *ind*: $X \le f\ (sup\ X\ (gfp\ f))$
  shows $X \le gfp\ f$
⟨*proof*⟩

12.5 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition $X \subseteq f\ X$ to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: $mono\ f \Longrightarrow mono\ (\lambda x.\ f\ x \cup X \cup B)$
⟨*proof*⟩

lemma *coinduct3-lemma*:
$X \subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup gfp\ f)) \Longrightarrow mono\ f \Longrightarrow$
$lfp\ (\lambda x.\ f\ x \cup X \cup gfp\ f) \subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup gfp\ f))$
⟨*proof*⟩

lemma *coinduct3*: $mono\ f \Longrightarrow a \in X \Longrightarrow X \subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup gfp\ f)) \Longrightarrow a \in gfp\ f$
⟨*proof*⟩

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding.

lemma *def-gfp-unfold*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow A = f\ A$
⟨*proof*⟩

lemma *def-coinduct*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow X \leq f\ (sup\ X\ A) \Longrightarrow X \leq A$
⟨*proof*⟩

lemma *def-coinduct-set*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow a \in X \Longrightarrow X \subseteq f\ (X \cup A) \Longrightarrow a \in A$
⟨*proof*⟩

lemma *def-Collect-coinduct*:
$A \equiv gfp\ (\lambda w.\ Collect\ (P\ w)) \Longrightarrow mono\ (\lambda w.\ Collect\ (P\ w)) \Longrightarrow a \in X \Longrightarrow$
$(\bigwedge z.\ z \in X \Longrightarrow P\ (X \cup A)\ z) \Longrightarrow a \in A$
⟨*proof*⟩

lemma *def-coinduct3*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow a \in X \Longrightarrow X \subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup A)) \Longrightarrow a \in A$
⟨*proof*⟩

Monotonicity of *gfp*!

lemma *gfp-mono*: $(\bigwedge Z.\ f\ Z \leq g\ Z) \Longrightarrow gfp\ f \leq gfp\ g$
⟨*proof*⟩

12.6 Rules for fixed point calculus

lemma *lfp-rolling*:
assumes $mono\ g\ mono\ f$
shows $g\ (lfp\ (\lambda x.\ f\ (g\ x))) = lfp\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *lfp-lfp*:
assumes f: $\bigwedge x\ y\ w\ z.\ x \leq y \Longrightarrow w \leq z \Longrightarrow f\ x\ w \leq f\ y\ z$
shows $lfp\ (\lambda x.\ lfp\ (f\ x)) = lfp\ (\lambda x.\ f\ x\ x)$

⟨*proof*⟩

lemma *gfp-rolling*:
  assumes *mono g mono f*
  shows $g\ (gfp\ (\lambda x.\ f\ (g\ x))) = gfp\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *gfp-gfp*:
  assumes f: $\bigwedge x\ y\ w\ z.\ x \le y \Longrightarrow w \le z \Longrightarrow f\ x\ w \le f\ y\ z$
  shows $gfp\ (\lambda x.\ gfp\ (f\ x)) = gfp\ (\lambda x.\ f\ x\ x)$
⟨*proof*⟩

12.7 Inductive predicates and sets

Package setup.

lemmas *basic-monos* =
  subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  Collect-mono in-mono vimage-mono

lemma *le-rel-bool-arg-iff*: $X \le Y \longleftrightarrow X\ False \le Y\ False \wedge X\ True \le Y\ True$
  ⟨*proof*⟩

lemma *imp-conj-iff*: $((P \longrightarrow Q) \wedge P) = (P \wedge Q)$
  ⟨*proof*⟩

lemma *meta-fun-cong*: $P \equiv Q \Longrightarrow P\ a \equiv Q\ a$
  ⟨*proof*⟩

⟨*ML*⟩

lemmas [*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

12.8 The Schroeder-Bernstein Theorem

See also:

- `$ISABELLE_HOME/src/HOL/ex/Set_Theory.thy`
- http://planetmath.org/proofofschroederbernsteintheoremusingtarskiknastertheorem
- Springer LNCS 828 (cover page)

theorem *Schroeder-Bernstein*:
  fixes $f :: 'a \Rightarrow 'b$ **and** $g :: 'b \Rightarrow 'a$

and *A* :: *'a set* **and** *B* :: *'b set*
assumes *inj1*: *inj-on f A* **and** *sub1*: *f ' A* ⊆ *B*
and *inj2*: *inj-on g B* **and** *sub2*: *g ' B* ⊆ *A*
shows ∃ *h*. *bij-betw h A B*
⟨*proof*⟩

12.9 Inductive datatypes and primitive recursion

Package setup.

⟨*ML*⟩

Lambda-abstractions with pattern matching:

syntax (*ASCII*)
-lam-pats-syntax :: *cases-syn* ⇒ *'a* ⇒ *'b* ((%-) *10*)
syntax
-lam-pats-syntax :: *cases-syn* ⇒ *'a* ⇒ *'b* ((λ-) *10*)
⟨*ML*⟩

end

13 Cartesian products

theory *Product-Type*
imports *Typedef Inductive Fun*
keywords *inductive-set coinductive-set* :: *thy-decl*
begin

13.1 *bool* is a datatype

free-constructors (*discs-sels*) *case-bool* **for** *True* | *False*
⟨*proof*⟩

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

⟨*ML*⟩

old-rep-datatype *True False* ⟨*proof*⟩

⟨*ML*⟩

But erase the prefix for properties that are not generated by *free-constructors*.

⟨*ML*⟩

lemmas *induct* = *old.bool.induct*
lemmas *inducts* = *old.bool.inducts*
lemmas *rec* = *old.bool.rec*
lemmas *simps* = *bool.distinct bool.case bool.rec*

⟨*ML*⟩

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

lemma [*code*]: $HOL.equal\ False\ P \longleftrightarrow \neg\ P$
and [*code*]: $HOL.equal\ True\ P \longleftrightarrow P$
and [*code*]: $HOL.equal\ P\ False \longleftrightarrow \neg\ P$
and [*code*]: $HOL.equal\ P\ True \longleftrightarrow P$
and [*code nbe*]: $HOL.equal\ P\ P \longleftrightarrow True$
⟨*proof*⟩

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

⟨*ML*⟩

code-printing
constant $HOL.equal :: bool \Rightarrow bool \Rightarrow bool$ ⇀ (*Haskell*) **infix** *4* ==
| **class-instance** *bool* :: *equal* ⇀ (*Haskell*) −

13.2 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*⟩

free-constructors *case-unit* **for** ()
⟨*proof*⟩

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

⟨*ML*⟩

old-rep-datatype () ⟨*proof*⟩

⟨*ML*⟩

But erase the prefix for properties that are not generated by *free-constructors*.

⟨*ML*⟩

lemmas *induct* = *old.unit.induct*
lemmas *inducts* = *old.unit.inducts*
lemmas *rec* = *old.unit.rec*
lemmas *simps* = *unit.case unit.rec*

⟨*ML*⟩

lemma *unit-all-eq1*: ($\bigwedge x$::*unit*. *PROP P x*) ≡ *PROP P* ()
⟨*proof*⟩

lemma *unit-all-eq2*: ($\bigwedge x$::*unit*. *PROP P*) ≡ *PROP P*
⟨*proof*⟩

This rewrite counters the effect of simproc *unit-eq* on λu::*unit*. *f u*, replacing it by *f* rather than by λu. *f* ().

lemma *unit-abs-eta-conv* [*simp*]: (λu::*unit*. *f* ()) = *f*
⟨*proof*⟩

lemma *UNIV-unit*: *UNIV* = {()}
⟨*proof*⟩

instantiation *unit* :: *default*
begin

definition *default* = ()

instance ⟨*proof*⟩

end

instantiation *unit* :: {*complete-boolean-algebra*,*complete-linorder*,*wellorder*}
begin

definition *less-eq-unit* :: *unit* ⇒ *unit* ⇒ *bool*
where (-::*unit*) ≤ - ⟷ *True*

lemma *less-eq-unit* [*iff*]: *u* ≤ *v* **for** *u v* :: *unit*
⟨*proof*⟩

definition *less-unit* :: *unit* ⇒ *unit* ⇒ *bool*
where (-::*unit*) < - ⟷ *False*

lemma *less-unit* [*iff*]: ¬ *u* < *v* **for** *u v* :: *unit*
⟨*proof*⟩

definition *bot-unit* :: *unit*
where [*code-unfold*]: ⊥ = ()

definition *top-unit* :: *unit*
  where [*code-unfold*]: $\top$ = ()

definition *inf-unit* :: *unit* $\Rightarrow$ *unit* $\Rightarrow$ *unit*
  where [*simp*]: - $\sqcap$ - = ()

definition *sup-unit* :: *unit* $\Rightarrow$ *unit* $\Rightarrow$ *unit*
  where [*simp*]: - $\sqcup$ - = ()

definition *Inf-unit* :: *unit set* $\Rightarrow$ *unit*
  where [*simp*]: $\bigsqcap$ - = ()

definition *Sup-unit* :: *unit set* $\Rightarrow$ *unit*
  where [*simp*]: $\bigsqcup$ - = ()

definition *uminus-unit* :: *unit* $\Rightarrow$ *unit*
  where [*simp*]: $-$ - = ()

declare *less-eq-unit-def* [*abs-def*, *code-unfold*]
  less-unit-def [*abs-def*, *code-unfold*]
  inf-unit-def [*abs-def*, *code-unfold*]
  sup-unit-def [*abs-def*, *code-unfold*]
  Inf-unit-def [*abs-def*, *code-unfold*]
  Sup-unit-def [*abs-def*, *code-unfold*]
  uminus-unit-def [*abs-def*, *code-unfold*]

instance
  ⟨*proof*⟩

end

lemma [*code*]: *HOL.equal u v* $\longleftrightarrow$ *True* **for** *u v* :: *unit*
  ⟨*proof*⟩

code-printing
  type-constructor *unit* $\rightharpoonup$
    (*SML*) *unit*
    and (*OCaml*) *unit*
    and (*Haskell*) ()
    and (*Scala*) *Unit*
| **constant** *Unity* $\rightharpoonup$
    (*SML*) ()
    and (*OCaml*) ()
    and (*Haskell*) ()
    and (*Scala*) ()
| **class-instance** *unit* :: *equal* $\rightharpoonup$
    (*Haskell*) $-$
| **constant** *HOL.equal* :: *unit* $\Rightarrow$ *unit* $\Rightarrow$ *bool* $\rightharpoonup$
    (*Haskell*) **infix** *4* ==

code-reserved *SML*
unit

code-reserved *OCaml*
unit

code-reserved *Scala*
Unit

13.3 The product type

13.3.1 Type definition

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

definition $prod = \{f.\ \exists\ a\ b.\ f = Pair\text{-}Rep\ (a::'a)\ (b::'b)\}$

typedef $('a,\ 'b)\ prod\ ((\text{-} \times/\ \text{-})\ [21,\ 20]\ 20) = prod :: ('a \Rightarrow 'b \Rightarrow bool)\ set$
⟨*proof*⟩

type-notation (*ASCII*)
prod (**infixr** $*$ *20*)

definition $Pair :: 'a \Rightarrow 'b \Rightarrow 'a \times 'b$
where $Pair\ a\ b = Abs\text{-}prod\ (Pair\text{-}Rep\ a\ b)$

lemma *prod-cases*: $(\bigwedge a\ b.\ P\ (Pair\ a\ b)) \Longrightarrow P\ p$
⟨*proof*⟩

free-constructors *case-prod* **for** *Pair fst snd*
⟨*proof*⟩

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

⟨*ML*⟩

old-rep-datatype *Pair*
⟨*proof*⟩

⟨*ML*⟩

But erase the prefix for properties that are not generated by *free-constructors*.

⟨*ML*⟩

declare *old.prod.inject* [*iff del*]

lemmas *induct* = *old.prod.induct*
lemmas *inducts* = *old.prod.inducts*

lemmas *rec* = *old.prod.rec*
lemmas *simps* = *prod.inject prod.case prod.rec*

⟨*ML*⟩

declare *prod.case* [*nitpick-simp del*]
declare *old.prod.case-cong-weak* [*cong del*]
declare *prod.case-eq-if* [*mono*]
declare *prod.split* [*no-atp*]
declare *prod.split-asm* [*no-atp*]

prod.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.

13.3.2 Tuple syntax

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

nonterminal *tuple-args* **and** *patterns*
syntax
  -tuple :: $'a \Rightarrow$ *tuple-args* $\Rightarrow 'a \times 'b$ ((*1*′(-,/ -′)))
  -tuple-arg :: $'a \Rightarrow$ *tuple-args* (-)
  -tuple-args :: $'a \Rightarrow$ *tuple-args* $\Rightarrow$ *tuple-args* (-,/ -)
  -pattern :: *pttrn* $\Rightarrow$ *patterns* $\Rightarrow$ *pttrn* (′(-,/ -′))
  :: *pttrn* $\Rightarrow$ *patterns* (-)
  -patterns :: *pttrn* $\Rightarrow$ *patterns* $\Rightarrow$ *patterns* (-,/ -)
  -unit :: *pttrn* (′(′))
translations
  $(x, y) \rightleftharpoons$ *CONST Pair x y*
  -pattern x y $\rightleftharpoons$ *CONST Pair x y*
  -patterns x y $\rightleftharpoons$ *CONST Pair x y*
  -tuple x (*-tuple-args y z*) $\rightleftharpoons$ *-tuple x* (*-tuple-arg* (*-tuple y z*))
  $\lambda(x, y, zs).\ b \rightleftharpoons$ *CONST case-prod* $(\lambda x\ (y, zs).\ b)$
  $\lambda(x, y).\ b \rightleftharpoons$ *CONST case-prod* $(\lambda x\ y.\ b)$
  -abs (*CONST Pair x y*) $t \rightharpoonup \lambda(x, y).\ t$
  — This rule accommodates tuples in *case C* ... (x, y) ... $\Rightarrow$ ...: The (x, y) is parsed as *Pair x y* because it is *logic*, not *pttrn*.
  $\lambda().\ b \rightleftharpoons$ *CONST case-unit b*
  -abs (*CONST Unity*) $t \rightharpoonup \lambda().\ t$

print *case-prod f* as *case-prod f* and *case-prod f* as *case-prod f*

⟨*ML*⟩

Reconstruct pattern from (nested) *case-prod*s, avoiding eta-contraction of body; required for enclosing "let", if "let" does not avoid eta-contraction, which has been observed to occur.

⟨*ML*⟩

13.3.3 Code generator setup

code-printing
type-constructor *prod* $\rightharpoonup$
(*SML*) **infix** *2* $*$
and (*OCaml*) **infix** *2* $*$
and (*Haskell*) !((-),/ (-))
and (*Scala*) ((-),/ (-))
| **constant** *Pair* $\rightharpoonup$
(*SML*) !((-),/ (-))
and (*OCaml*) !((-),/ (-))
and (*Haskell*) !((-),/ (-))
and (*Scala*) !((-),/ (-))
| **class-instance** *prod* :: *equal* $\rightharpoonup$
(*Haskell*) $-$
| **constant** *HOL.equal* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool$ $\rightharpoonup$
(*Haskell*) **infix** *4* ==
| **constant** *fst* $\rightharpoonup$ (*Haskell*) *fst*
| **constant** *snd* $\rightharpoonup$ (*Haskell*) *snd*

13.3.4 Fundamental operations and properties

lemma *Pair-inject*: $(a, b) = (a', b') \Longrightarrow (a = a' \Longrightarrow b = b' \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

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

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

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

lemma *case-prod-unfold* [*nitpick-unfold*]: $case\text{-}prod = (\lambda c\ p.\ c\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

lemma *case-prod-conv* [*simp*, *code*]: $(case\ (a, b)\ of\ (c, d) \Rightarrow f\ c\ d) = f\ a\ b$
⟨*proof*⟩

lemmas *surjective-pairing* = *prod.collapse* [*symmetric*]

lemma *prod-eq-iff*: $s = t \longleftrightarrow fst\ s = fst\ t \wedge snd\ s = snd\ t$
⟨*proof*⟩

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

lemma *case-prodI*: $f\ a\ b \Longrightarrow case\ (a, b)\ of\ (c, d) \Rightarrow f\ c\ d$
⟨*proof*⟩

lemma *case-prodD*: $(case\ (a,\ b)\ of\ (c,\ d) \Rightarrow f\ c\ d) \Longrightarrow f\ a\ b$
⟨*proof*⟩

lemma *case-prod-Pair* [*simp*]: $case\text{-}prod\ Pair = id$
⟨*proof*⟩

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

lemma *case-prod-comp*: $(case\ x\ of\ (a,\ b) \Rightarrow (f \circ g)\ a\ b) = f\ (g\ (fst\ x))\ (snd\ x)$
⟨*proof*⟩

lemma *The-case-prod*: $The\ (case\text{-}prod\ P) = (THE\ xy.\ P\ (fst\ xy)\ (snd\ xy))$
⟨*proof*⟩

lemma *cond-case-prod-eta*: $(\bigwedge x\ y.\ f\ x\ y = g\ (x,\ y)) \Longrightarrow (\lambda(x,\ y).\ f\ x\ y) = g$
⟨*proof*⟩

lemma *split-paired-all* [*no-atp*]: $(\bigwedge x.\ PROP\ P\ x) \equiv (\bigwedge a\ b.\ PROP\ P\ (a,\ b))$
⟨*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 $\bigwedge a\ b.\ \ldots = ?P(a,\ b)$ which cannot be solved by reflexivity.

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

⟨*ML*⟩

lemma *split-paired-All* [*simp*, *no-atp*]: $(\forall x.\ P\ x) \longleftrightarrow (\forall a\ b.\ P\ (a,\ b))$
— [*iff*] is not a good idea because it makes *blast* loop
⟨*proof*⟩

lemma *split-paired-Ex* [*simp*, *no-atp*]: $(\exists x.\ P\ x) \longleftrightarrow (\exists a\ b.\ P\ (a,\ b))$
⟨*proof*⟩

lemma *split-paired-The* [*no-atp*]: $(THE\ x.\ P\ x) = (THE\ (a,\ b).\ P\ (a,\ b))$
— Can’t be added to simpset: loops!
⟨*proof*⟩

Simplification procedure for *cond-case-prod-eta*. Using *case-prod-eta* as a rewrite rule is not general enough, and using *cond-case-prod-eta* directly would render some existing proofs very inefficient; similarly for *prod.case-eq-if*.

⟨*ML*⟩

lemma *case-prod-beta'*: $(\lambda(x,y).\ f\ x\ y) = (\lambda x.\ f\ (fst\ x)\ (snd\ x))$
⟨*proof*⟩

case-prod used as a logical connective or set former.

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

lemma *case-prodI2*:
$\bigwedge p.\ (\bigwedge a\ b.\ p = (a,\ b) \Longrightarrow c\ a\ b) \Longrightarrow case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b$
⟨*proof*⟩

lemma *case-prodI2'*:
$\bigwedge p.\ (\bigwedge a\ b.\ (a,\ b) = p \Longrightarrow c\ a\ b\ x) \Longrightarrow (case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b)\ x$
⟨*proof*⟩

lemma *case-prodE* [*elim*!]:
$(case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b) \Longrightarrow (\bigwedge x\ y.\ p = (x,\ y) \Longrightarrow c\ x\ y \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *case-prodE'* [*elim*!]:
$(case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b)\ z \Longrightarrow (\bigwedge x\ y.\ p = (x,\ y) \Longrightarrow c\ x\ y\ z \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *case-prodE2*:
assumes *q*: $Q\ (case\ z\ of\ (a,\ b) \Rightarrow P\ a\ b)$
and *r*: $\bigwedge x\ y.\ z = (x,\ y) \Longrightarrow Q\ (P\ x\ y) \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *case-prodD'*: $(case\ (a,\ b)\ of\ (c,\ d) \Rightarrow R\ c\ d)\ c \Longrightarrow R\ a\ b\ c$
⟨*proof*⟩

lemma *mem-case-prodI*: $z \in c\ a\ b \Longrightarrow z \in (case\ (a,\ b)\ of\ (d,\ e) \Rightarrow c\ d\ e)$
⟨*proof*⟩

lemma *mem-case-prodI2* [*intro*!]:
$\bigwedge p.\ (\bigwedge a\ b.\ p = (a,\ b) \Longrightarrow z \in c\ a\ b) \Longrightarrow z \in (case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b)$
⟨*proof*⟩

declare *mem-case-prodI* [*intro*!] — postponed to maintain traditional declaration order!
declare *case-prodI2'* [*intro*!] — postponed to maintain traditional declaration order!
declare *case-prodI2* [*intro*!] — postponed to maintain traditional declaration order!

declare *case-prodI* [*intro*!] — postponed to maintain traditional declaration order!

lemma *mem-case-prodE* [*elim*!]:
assumes $z \in case\text{-}prod\ c\ p$
obtains $x\ y$ **where** $p = (x,\ y)$ **and** $z \in c\ x\ y$
⟨*proof*⟩

⟨*ML*⟩

lemma *split-eta-SetCompr* [*simp*, *no-atp*]: $(\lambda u.\ \exists\, x\ y.\ u = (x,\ y) \wedge P\ (x,\ y)) = P$
⟨*proof*⟩

lemma *split-eta-SetCompr2* [*simp*, *no-atp*]: $(\lambda u.\ \exists\, x\ y.\ u = (x,\ y) \wedge P\ x\ y) =$ *case-prod* P
⟨*proof*⟩

lemma *split-part* [*simp*]: $(\lambda(a,b).\ P \wedge Q\ a\ b) = (\lambda ab.\ P \wedge$ *case-prod* $Q\ ab)$
— Allows simplifications of nested splits in case of independent predicates.
⟨*proof*⟩

lemma *split-comp-eq*:
fixes f :: $'a \Rightarrow\ 'b \Rightarrow\ 'c$
and g :: $'d \Rightarrow\ 'a$
shows $(\lambda u.\ f\ (g\ (fst\ u))\ (snd\ u)) =$ *case-prod* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma *pair-imageI* [*intro*]: $(a,\ b) \in A \Longrightarrow f\ a\ b \in (\lambda(a,\ b).\ f\ a\ b)$ ‘ A
⟨*proof*⟩

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

lemma *case-prod-beta*: *case-prod* $f\ p = f\ (fst\ p)\ (snd\ p)$
⟨*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*]:
$(\bigwedge a\ b\ c.\ P\ (a,\ b,\ c)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d.\ P\ (a,\ b,\ c,\ d)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e.\ P\ (a,\ b,\ c,\ d,\ e)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f.\ P\ (a,\ b,\ c,\ d,\ e,\ f)) \Longrightarrow P\ x$
⟨*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*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g.\ P\ (a,\ b,\ c,\ d,\ e,\ f,\ g)) \Longrightarrow P\ x$
⟨*proof*⟩

definition *internal-case-prod* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
where *internal-case-prod* $\equiv$ *case-prod*

lemma *internal-case-prod-conv*: *internal-case-prod* $c\ (a,\ b) = c\ a\ b$
⟨*proof*⟩

⟨*ML*⟩

hide-const *internal-case-prod*

13.3.5 Derived operations

definition *curry* :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$
where *curry* $= (\lambda c\ x\ y.\ c\ (x,\ y))$

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

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

lemma *curry-K*: *curry* ($\lambda x.\ c$) = ($\lambda x\ y.\ c$)
⟨*proof*⟩

The composition-uncurry combinator.

notation *fcomp* (**infixl** ∘> *60*)

definition *scomp* :: ($'a \Rightarrow 'b \times 'c$) ⇒ ($'b \Rightarrow 'c \Rightarrow 'd$) ⇒ $'a \Rightarrow 'd$ (**infixl** ∘→ *60*)
where *f* ∘→ *g* = ($\lambda x.$ *case-prod g* (*f x*))

lemma *scomp-unfold*: *scomp* = ($\lambda f\ g\ x.\ g$ (*fst* (*f x*)) (*snd* (*f x*)))
⟨*proof*⟩

lemma *scomp-apply* [*simp*]: (*f* ∘→ *g*) *x* = *case-prod g* (*f x*)
⟨*proof*⟩

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

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

lemma *scomp-scomp*: (*f* ∘→ *g*) ∘→ *h* = *f* ∘→ ($\lambda x.$ *g x* ∘→ *h*)
⟨*proof*⟩

lemma *scomp-fcomp*: (*f* ∘→ *g*) ∘> *h* = *f* ∘→ ($\lambda x.$ *g x* ∘> *h*)
⟨*proof*⟩

lemma *fcomp-scomp*: (*f* ∘> *g*) ∘→ *h* = *f* ∘> (*g* ∘→ *h*)
⟨*proof*⟩

code-printing
constant *scomp* ⇀ (*Eval*) **infixl** *3* #−>

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

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

definition *map-prod* :: ($'a \Rightarrow 'c$) ⇒ ($'b \Rightarrow 'd$) ⇒ $'a \times 'b \Rightarrow 'c \times 'd$
where *map-prod f g* = ($\lambda(x,\ y).\ (f\ x,\ g\ y)$)

lemma *map-prod-simp* [*simp*, *code*]: *map-prod f g* (*a*, *b*) = (*f a*, *g b*)
⟨*proof*⟩

functor *map-prod*: *map-prod*
⟨*proof*⟩

lemma *fst-map-prod* [*simp*]: $fst\ (map\text{-}prod\ f\ g\ x) = f\ (fst\ x)$
⟨*proof*⟩

lemma *snd-map-prod* [*simp*]: $snd\ (map\text{-}prod\ f\ g\ x) = g\ (snd\ x)$
⟨*proof*⟩

lemma *fst-comp-map-prod* [*simp*]: $fst \circ map\text{-}prod\ f\ g = f \circ fst$
⟨*proof*⟩

lemma *snd-comp-map-prod* [*simp*]: $snd \circ map\text{-}prod\ f\ g = g \circ snd$
⟨*proof*⟩

lemma *map-prod-compose*: $map\text{-}prod\ (f1 \circ f2)\ (g1 \circ g2) = (map\text{-}prod\ f1\ g1 \circ map\text{-}prod\ f2\ g2)$
⟨*proof*⟩

lemma *map-prod-ident* [*simp*]: $map\text{-}prod\ (\lambda x.\ x)\ (\lambda y.\ y) = (\lambda z.\ z)$
⟨*proof*⟩

lemma *map-prod-imageI* [*intro*]: $(a,\ b) \in R \Longrightarrow (f\ a,\ g\ b) \in map\text{-}prod\ f\ g\ `\ R$
⟨*proof*⟩

lemma *prod-fun-imageE* [*elim*!]:
assumes *major*: $c \in map\text{-}prod\ f\ g\ `\ R$
and *cases*: $\bigwedge x\ y.\ c = (f\ x,\ g\ y) \Longrightarrow (x,\ y) \in R \Longrightarrow P$
shows P
⟨*proof*⟩

definition *apfst* :: $('a \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'b$
where $apfst\ f = map\text{-}prod\ f\ id$

definition *apsnd* :: $('b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'c$
where $apsnd\ f = map\text{-}prod\ id\ f$

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-comp-apfst* [*simp*]: $fst \circ apfst\ f = f \circ fst$
⟨*proof*⟩

lemma *fst-apsnd* [*simp*]: $fst\ (apsnd\ f\ x) = fst\ x$
⟨*proof*⟩

lemma *fst-comp-apsnd* [*simp*]: $fst \circ apsnd\ f = fst$
⟨*proof*⟩

lemma *snd-apfst* [*simp*]: $snd\ (apfst\ f\ x) = snd\ x$
⟨*proof*⟩

lemma *snd-comp-apfst* [*simp*]: $snd \circ apfst\ f = snd$
⟨*proof*⟩

lemma *snd-apsnd* [*simp*]: $snd\ (apsnd\ f\ x) = f\ (snd\ x)$
⟨*proof*⟩

lemma *snd-comp-apsnd* [*simp*]: $snd \circ apsnd\ f = f \circ snd$
⟨*proof*⟩

lemma *apfst-compose*: $apfst\ f\ (apfst\ g\ x) = apfst\ (f \circ g)\ x$
⟨*proof*⟩

lemma *apsnd-compose*: $apsnd\ f\ (apsnd\ g\ x) = apsnd\ (f \circ g)\ x$
⟨*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*]: $apfst\ f\ x = apfst\ g\ x \longleftrightarrow f\ (fst\ x) = g\ (fst\ x)$
⟨*proof*⟩

lemma *apsnd-eq-conv* [*simp*]: $apsnd\ f\ x = apsnd\ g\ x \longleftrightarrow f\ (snd\ x) = g\ (snd\ x)$
⟨*proof*⟩

lemma *apsnd-apfst-commute*: $apsnd\ f\ (apfst\ g\ p) = apfst\ g\ (apsnd\ f\ p)$
⟨*proof*⟩

context
begin

⟨ML⟩

definition *swap* :: $'a \times 'b \Rightarrow 'b \times 'a$
  where *swap p* = (*snd p*, *fst p*)

end

lemma *swap-simp* [*simp*]: *prod.swap* $(x, y) = (y, x)$
  ⟨*proof*⟩

lemma *swap-swap* [*simp*]: *prod.swap* (*prod.swap p*) = *p*
  ⟨*proof*⟩

lemma *swap-comp-swap* [*simp*]: *prod.swap* ∘ *prod.swap* = *id*
  ⟨*proof*⟩

lemma *pair-in-swap-image* [*simp*]: $(y, x) \in$ *prod.swap* ‘ $A \longleftrightarrow (x, y) \in A$
  ⟨*proof*⟩

lemma *inj-swap* [*simp*]: *inj-on prod.swap A*
  ⟨*proof*⟩

lemma *swap-inj-on*: *inj-on* $(\lambda(i, j). (j, i))$ A
  ⟨*proof*⟩

lemma *surj-swap* [*simp*]: *surj prod.swap*
  ⟨*proof*⟩

lemma *bij-swap* [*simp*]: *bij prod.swap*
  ⟨*proof*⟩

lemma *case-swap* [*simp*]: (*case prod.swap p of* $(y, x) \Rightarrow f\ x\ y$) = (*case p of* $(x, y) \Rightarrow f\ x\ y$)
  ⟨*proof*⟩

lemma *fst-swap* [*simp*]: *fst* (*prod.swap x*) = *snd x*
  ⟨*proof*⟩

lemma *snd-swap* [*simp*]: *snd* (*prod.swap x*) = *fst x*
  ⟨*proof*⟩

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: $'a$ *set* $\Rightarrow ('a \Rightarrow 'b$ *set*$) \Rightarrow ('a \times 'b)$ *set*
  where *Sigma A B* $\equiv \bigcup x{\in}A. \bigcup y{\in}B\ x. \{Pair\ x\ y\}$

abbreviation *Times* :: $'a$ *set* $\Rightarrow 'b$ *set* $\Rightarrow ('a \times 'b)$ *set* (**infixr** × *80*)
  where $A \times B \equiv$ *Sigma A* $(\lambda\text{-}. B)$

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

syntax
-Sigma :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ (*'a* × *'b*) *set* ((*3SIGMA -:-./ -*) [*0*, *0*, *10*] *10*)
translations
SIGMA x:*A*. *B* ⇌ *CONST Sigma A* (λ*x*. *B*)

lemma *SigmaI* [*intro*!]: $a \in A \Longrightarrow b \in B\ a \Longrightarrow (a, b) \in Sigma\ A\ B$
⟨*proof*⟩

lemma *SigmaE* [*elim*!]: $c \in Sigma\ A\ B \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in B\ x \Longrightarrow c = (x, y) \Longrightarrow P) \Longrightarrow P$
— The general elimination rule.
⟨*proof*⟩

Elimination of $(a, b) \in A \times B$ – introduces no eigenvariables.

lemma *SigmaD1*: $(a, b) \in Sigma\ A\ B \Longrightarrow a \in A$
⟨*proof*⟩

lemma *SigmaD2*: $(a, b) \in Sigma\ A\ B \Longrightarrow b \in B\ a$
⟨*proof*⟩

lemma *SigmaE2*: $(a, b) \in Sigma\ A\ B \Longrightarrow (a \in A \Longrightarrow b \in B\ a \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Sigma-cong*: $A = B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow C\ x = D\ x) \Longrightarrow (SIGMA\ x{:}A.\ C\ x) = (SIGMA\ x{:}B.\ D\ x)$
⟨*proof*⟩

lemma *Sigma-mono*: $A \subseteq C \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq D\ x) \Longrightarrow Sigma\ A\ B \subseteq Sigma\ C\ D$
⟨*proof*⟩

lemma *Sigma-empty1* [*simp*]: $Sigma\ \{\}\ B = \{\}$
⟨*proof*⟩

lemma *Sigma-empty2* [*simp*]: $A \times \{\} = \{\}$
⟨*proof*⟩

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

lemma *Compl-Times-UNIV1* [*simp*]: $-(UNIV \times A) = UNIV \times (-A)$
⟨*proof*⟩

lemma *Compl-Times-UNIV2* [*simp*]: $-(A \times UNIV) = (-A) \times UNIV$
⟨*proof*⟩

lemma *mem-Sigma-iff* [*iff*]: $(a, b) \in Sigma\ A\ B \longleftrightarrow a \in A \wedge b \in B\ a$

$\langle proof \rangle$

lemma *mem-Times-iff*: $x \in A \times B \longleftrightarrow fst\ x \in A \wedge snd\ x \in B$
$\langle proof \rangle$

lemma *Sigma-empty-iff*: $(SIGMA\ i{:}I.\ X\ i) = \{\} \longleftrightarrow (\forall i{\in}I.\ X\ i = \{\})$
$\langle proof \rangle$

lemma *Times-subset-cancel2*: $x \in C \Longrightarrow A \times C \subseteq B \times C \longleftrightarrow A \subseteq B$
$\langle proof \rangle$

lemma *Times-eq-cancel2*: $x \in C \Longrightarrow A \times C = B \times C \longleftrightarrow A = B$
$\langle proof \rangle$

lemma *Collect-case-prod-Sigma*: $\{(x, y).\ P\ x \wedge Q\ x\ y\} = (SIGMA\ x{:}Collect\ P.\ Collect\ (Q\ x))$
$\langle proof \rangle$

lemma *Collect-case-prod* [*simp*]: $\{(a, b).\ P\ a \wedge Q\ b\} = Collect\ P \times Collect\ Q$
$\langle proof \rangle$

lemma *Collect-case-prodD*: $x \in Collect\ (case\text{-}prod\ A) \Longrightarrow A\ (fst\ x)\ (snd\ x)$
$\langle proof \rangle$

lemma *Collect-case-prod-mono*: $A \leq B \Longrightarrow Collect\ (case\text{-}prod\ A) \subseteq Collect\ (case\text{-}prod\ B)$
$\langle proof \rangle$

lemma *Collect-split-mono-strong*:
$X = fst\ `\ A \Longrightarrow Y = snd\ `\ A \Longrightarrow \forall a{\in}X.\ \forall b \in Y.\ P\ a\ b \longrightarrow Q\ a\ b$
$\Longrightarrow A \subseteq Collect\ (case\text{-}prod\ P) \Longrightarrow A \subseteq Collect\ (case\text{-}prod\ Q)$
$\langle proof \rangle$

lemma *UN-Times-distrib*: $(\bigcup(a, b){\in}A \times B.\ E\ a \times F\ b) = UNION\ A\ E \times UNION\ B\ F$
— Suggested by Pierre Chartier
$\langle proof \rangle$

lemma *split-paired-Ball-Sigma* [*simp*, *no-atp*]: $(\forall z{\in}Sigma\ A\ B.\ P\ z) \longleftrightarrow (\forall x{\in}A.\ \forall y{\in}B\ x.\ P\ (x, y))$
$\langle proof \rangle$

lemma *split-paired-Bex-Sigma* [*simp*, *no-atp*]: $(\exists z{\in}Sigma\ A\ B.\ P\ z) \longleftrightarrow (\exists x{\in}A.\ \exists y{\in}B\ x.\ P\ (x, y))$
$\langle proof \rangle$

lemma *Sigma-Un-distrib1*: $Sigma\ (I \cup J)\ C = Sigma\ I\ C \cup Sigma\ J\ C$
$\langle proof \rangle$

lemma *Sigma-Un-distrib2*: $(SIGMA\ i{:}I.\ A\ i \cup B\ i) = Sigma\ I\ A \cup Sigma\ I\ B$
⟨*proof*⟩

lemma *Sigma-Int-distrib1*: $Sigma\ (I \cap J)\ C = Sigma\ I\ C \cap Sigma\ J\ C$
⟨*proof*⟩

lemma *Sigma-Int-distrib2*: $(SIGMA\ i{:}I.\ A\ i \cap B\ i) = Sigma\ I\ A \cap Sigma\ I\ B$
⟨*proof*⟩

lemma *Sigma-Diff-distrib1*: $Sigma\ (I - J)\ C = Sigma\ I\ C - Sigma\ J\ C$
⟨*proof*⟩

lemma *Sigma-Diff-distrib2*: $(SIGMA\ i{:}I.\ A\ i - B\ i) = Sigma\ I\ A - Sigma\ I\ B$
⟨*proof*⟩

lemma *Sigma-Union*: $Sigma\ (\bigcup X)\ B = (\bigcup A{\in}X.\ Sigma\ A\ B)$
⟨*proof*⟩

lemma *Pair-vimage-Sigma*: $Pair\ x\ -`\ Sigma\ A\ f = (if\ x \in A\ then\ f\ x\ else\ \{\})$
⟨*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*: $(A \cup B) \times C = A \times C \cup B \times C$
⟨*proof*⟩

lemma *Times-Int-distrib1*: $(A \cap B) \times C = A \times C \cap B \times C$
⟨*proof*⟩

lemma *Times-Diff-distrib1*: $(A - B) \times C = A \times C - B \times C$
⟨*proof*⟩

lemma *Times-empty* [*simp*]: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
⟨*proof*⟩

lemma *times-eq-iff*: $A \times B = C \times D \longleftrightarrow A = C \wedge B = D \vee (A = \{\} \vee B = \{\}) \wedge (C = \{\} \vee D = \{\})$
⟨*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 *fst-image-Sigma*: $fst\ `\ (Sigma\ A\ B) = \{x \in A.\ B(x) \neq \{\}\}$
⟨*proof*⟩

lemma *snd-image-Sigma*: $snd\ `\ (Sigma\ A\ B) = (\bigcup\ x \in A.\ B\ x)$

⟨*proof*⟩

lemma *vimage-fst*: $fst\ -`\ A = A \times UNIV$
⟨*proof*⟩

lemma *vimage-snd*: $snd\ -`\ A = UNIV \times A$
⟨*proof*⟩

lemma *insert-times-insert* [*simp*]:
$insert\ a\ A \times insert\ b\ B = insert\ (a,b)\ (A \times insert\ b\ B \cup insert\ a\ A \times B)$
⟨*proof*⟩

lemma *vimage-Times*: $f\ -`\ (A \times B) = (fst \circ f)\ -`\ A \cap (snd \circ f)\ -`\ B$
⟨*proof*⟩

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

lemma *product-swap*: $prod.swap\ `\ (A \times B) = B \times A$
⟨*proof*⟩

lemma *swap-product*: $(\lambda(i, j).\ (j, i))\ `\ (A \times B) = B \times A$
⟨*proof*⟩

lemma *image-split-eq-Sigma*: $(\lambda x.\ (f\ x,\ g\ x))\ `\ A = Sigma\ (f\ `\ A)\ (\lambda x.\ g\ `\ (f\ -`\ \{x\} \cap A))$
⟨*proof*⟩

lemma *subset-fst-snd*: $A \subseteq (fst\ `\ A \times snd\ `\ A)$
⟨*proof*⟩

lemma *inj-on-apfst* [*simp*]: $inj\text{-}on\ (apfst\ f)\ (A \times UNIV) \longleftrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *inj-apfst* [*simp*]: $inj\ (apfst\ f) \longleftrightarrow inj\ f$
⟨*proof*⟩

lemma *inj-on-apsnd* [*simp*]: $inj\text{-}on\ (apsnd\ f)\ (UNIV \times A) \longleftrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *inj-apsnd* [*simp*]: $inj\ (apsnd\ f) \longleftrightarrow inj\ f$
⟨*proof*⟩

context
begin

qualified definition $product :: 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \times 'b)\ set$
where [*code-abbrev*]: $product\ A\ B = A \times B$

lemma *member-product*: $x \in$ *Product-Type.product* $A\ B \longleftrightarrow x \in A \times B$
⟨*proof*⟩

end

The following *map-prod* lemmas are due to Joachim Breitner:

lemma *map-prod-inj-on*:
assumes *inj-on* $f\ A$
and *inj-on* $g\ B$
shows *inj-on* (*map-prod* $f\ g$) ($A \times B$)
⟨*proof*⟩

lemma *map-prod-surj*:
fixes $f :: 'a \Rightarrow 'b$
and $g :: 'c \Rightarrow 'd$
assumes *surj* f **and** *surj* g
shows *surj* (*map-prod* $f\ g$)
⟨*proof*⟩

lemma *map-prod-surj-on*:
assumes f ‘ $A = A'$ **and** g ‘ $B = B'$
shows *map-prod* $f\ g$ ‘ $(A \times B) = A' \times B'$
⟨*proof*⟩

13.4 Simproc for rewriting a set comprehension into a pointfree expression

⟨*ML*⟩

13.5 Inductively defined sets

⟨*ML*⟩

13.6 Legacy theorem bindings and duplicates

lemmas *fst-conv* = *prod.sel*(*1*)
lemmas *snd-conv* = *prod.sel*(*2*)
lemmas *split-def* = *case-prod-unfold*
lemmas *split-beta′* = *case-prod-beta′*
lemmas *split-beta* = *prod.case-eq-if*
lemmas *split-conv* = *case-prod-conv*
lemmas *split* = *case-prod-conv*

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

end

14 The Disjoint Sum of Two Types

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

14.1 Construction of the sum type and its basic abstract operations

definition $\mathit{Inl\text{-}Rep} :: {'a} \Rightarrow {'a} \Rightarrow {'b} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$
where $\mathit{Inl\text{-}Rep}\ a\ x\ y\ p \longleftrightarrow x = a \wedge p$

definition $\mathit{Inr\text{-}Rep} :: {'b} \Rightarrow {'a} \Rightarrow {'b} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$
where $\mathit{Inr\text{-}Rep}\ b\ x\ y\ p \longleftrightarrow y = b \wedge \neg\ p$

definition $\mathit{sum} = \{f.\ (\exists\, a.\ f = \mathit{Inl\text{-}Rep}\ (a{::}{'a})) \vee (\exists\, b.\ f = \mathit{Inr\text{-}Rep}\ (b{::}{'b}))\}$

typedef $({'a}, {'b})\ \mathit{sum}$ (**infixr** $+$ *10*) $= \mathit{sum} :: ({'a} \Rightarrow {'b} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool})\ \mathit{set}$
⟨*proof*⟩

lemma *Inl-RepI*: $\mathit{Inl\text{-}Rep}\ a \in \mathit{sum}$
⟨*proof*⟩

lemma *Inr-RepI*: $\mathit{Inr\text{-}Rep}\ b \in \mathit{sum}$
⟨*proof*⟩

lemma *inj-on-Abs-sum*: $A \subseteq \mathit{sum} \Longrightarrow \mathit{inj\text{-}on}\ \mathit{Abs\text{-}sum}\ A$
⟨*proof*⟩

lemma *Inl-Rep-inject*: $\mathit{inj\text{-}on}\ \mathit{Inl\text{-}Rep}\ A$
⟨*proof*⟩

lemma *Inr-Rep-inject*: $\mathit{inj\text{-}on}\ \mathit{Inr\text{-}Rep}\ A$
⟨*proof*⟩

lemma *Inl-Rep-not-Inr-Rep*: $\mathit{Inl\text{-}Rep}\ a \neq \mathit{Inr\text{-}Rep}\ b$
⟨*proof*⟩

definition $\mathit{Inl} :: {'a} \Rightarrow {'a} + {'b}$
where $\mathit{Inl} = \mathit{Abs\text{-}sum} \circ \mathit{Inl\text{-}Rep}$

definition $\mathit{Inr} :: {'b} \Rightarrow {'a} + {'b}$
where $\mathit{Inr} = \mathit{Abs\text{-}sum} \circ \mathit{Inr\text{-}Rep}$

lemma *inj-Inl* [*simp*]: $\mathit{inj\text{-}on}\ \mathit{Inl}\ A$
⟨*proof*⟩

lemma *Inl-inject*: $\mathit{Inl}\ x = \mathit{Inl}\ y \Longrightarrow x = y$
⟨*proof*⟩

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
⟨*proof*⟩

lemma *Inr-inject*: *Inr* x = *Inr* $y \Longrightarrow x = y$
⟨*proof*⟩

lemma *Inl-not-Inr*: *Inl* $a \neq$ *Inr* b
⟨*proof*⟩

lemma *Inr-not-Inl*: *Inr* $b \neq$ *Inl* a
⟨*proof*⟩

lemma *sumE*:
assumes $\bigwedge x::'a.\ s =$ *Inl* $x \Longrightarrow P$
and $\bigwedge y::'b.\ s =$ *Inr* $y \Longrightarrow P$
shows P
⟨*proof*⟩

free-constructors *case-sum* **for**
isl: *Inl projl*
| *Inr projr*
⟨*proof*⟩

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

⟨*ML*⟩

old-rep-datatype *Inl Inr*
⟨*proof*⟩

⟨*ML*⟩

But erase the prefix for properties that are not generated by *free-constructors*.

⟨*ML*⟩

declare
old.sum.inject[*iff del*]
old.sum.distinct(*1*)[*simp del*, *induct-simp del*]

lemmas *induct* = *old.sum.induct*
lemmas *inducts* = *old.sum.inducts*
lemmas *rec* = *old.sum.rec*
lemmas *simps* = *sum.inject sum.distinct sum.case sum.rec*

⟨*ML*⟩

primrec *map-sum* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a + 'b \Rightarrow 'c + 'd$
where
map-sum f1 f2 (*Inl* a) = *Inl* (*f1* a)

| *map-sum f1 f2* (*Inr a*) = *Inr* (*f2 a*)

functor *map-sum*: *map-sum*
⟨*proof*⟩

lemma *split-sum-all*: $(\forall x.\ P\ x) \longleftrightarrow (\forall x.\ P\ (Inl\ x)) \wedge (\forall x.\ P\ (Inr\ x))$
⟨*proof*⟩

lemma *split-sum-ex*: $(\exists x.\ P\ x) \longleftrightarrow (\exists x.\ P\ (Inl\ x)) \vee (\exists x.\ P\ (Inr\ x))$
⟨*proof*⟩

14.2 Projections

lemma *case-sum-KK* [*simp*]: *case-sum* $(\lambda x.\ a)\ (\lambda x.\ a) = (\lambda x.\ a)$
⟨*proof*⟩

lemma *surjective-sum*: *case-sum* $(\lambda x::'a.\ f\ (Inl\ x))\ (\lambda y::'b.\ f\ (Inr\ y)) = f$
⟨*proof*⟩

lemma *case-sum-inject*:
assumes *a*: *case-sum f1 f2* = *case-sum g1 g2*
and *r*: $f1 = g1 \Longrightarrow f2 = g2 \Longrightarrow P$
shows *P*
⟨*proof*⟩

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*⟩

14.3 The Disjoint Sum of Sets

definition *Plus* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a + 'b)\ set$ (**infixr** <+> *65*)
where $A <\!+\!> B = Inl\ `\ A \cup Inr\ `\ B$

hide-const (**open**) *Plus* — Valuable identifier

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*]: $A <+> B = \{\} \longleftrightarrow A = \{\} \land B = \{\}$
⟨*proof*⟩

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

lemma *UNIV-sum*: $UNIV = Inl\ `\ UNIV \cup Inr\ `\ UNIV$
⟨*proof*⟩

hide-const (**open**) *Suml Sumr sum*

end

15 Rings

theory *Rings*
imports *Groups Set*
begin

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
assumes *distrib-right*[*algebra-simps*]: $(a + b) * c = a * c + b * c$
assumes *distrib-left*[*algebra-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*]: $a * 0 = 0$
begin

lemma *mult-not-zero*: $a * b \neq 0 \Longrightarrow a \neq 0 \land b \neq 0$
⟨*proof*⟩

end

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*]: $0 \neq 1$
begin

lemma *one-neq-zero* [*simp*]: $1 \neq 0$
  ⟨*proof*⟩

definition *of-bool* :: *bool* ⇒ *'a*
  where *of-bool p* = (*if p then 1 else 0*)

lemma *of-bool-eq* [*simp*, *code*]:
  of-bool False = *0*

of-bool True = *1*
⟨*proof*⟩

lemma *of-bool-eq-iff*: *of-bool p* = *of-bool q* ⟷ *p* = *q*
⟨*proof*⟩

lemma *split-of-bool* [*split*]: *P* (*of-bool p*) ⟷ (*p* ⟶ *P 1*) ∧ (¬ *p* ⟶ *P 0*)
⟨*proof*⟩

lemma *split-of-bool-asm*: *P* (*of-bool p*) ⟷ ¬ (*p* ∧ ¬ *P 1* ∨ ¬ *p* ∧ ¬ *P 0*)
⟨*proof*⟩

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** *dvd 50*)
where *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

context *comm-monoid-mult*
begin

subclass *dvd* ⟨*proof*⟩

lemma *dvd-refl* [*simp*]: *a dvd a*
⟨*proof*⟩

lemma *dvd-trans* [*trans*]:
assumes *a dvd b* **and** *b dvd c*
shows *a dvd c*
⟨*proof*⟩

lemma *subset-divisors-dvd*: {*c*. *c dvd a*} ⊆ {*c*. *c dvd b*} ⟷ *a dvd b*
⟨*proof*⟩

lemma *strict-subset-divisors-dvd*: {*c*. *c dvd a*} ⊂ {*c*. *c dvd b*} ⟷ *a dvd b* ∧ ¬ *b dvd a*

⟨*proof*⟩

lemma *one-dvd* [*simp*]: *1 dvd a*
⟨*proof*⟩

lemma *dvd-mult* [*simp*]: $a\ dvd\ c \Longrightarrow a\ dvd\ (b * c)$
⟨*proof*⟩

lemma *dvd-mult2* [*simp*]: $a\ dvd\ b \Longrightarrow a\ dvd\ (b * c)$
⟨*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*: $a * b\ dvd\ c \Longrightarrow a\ dvd\ c$
⟨*proof*⟩

lemma *dvd-mult-right*: $a * b\ dvd\ c \Longrightarrow b\ dvd\ c$
⟨*proof*⟩

end

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*
begin

subclass *semiring-1* ⟨*proof*⟩

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

lemma *dvd-0-right* [*iff*]: *a dvd 0*
⟨*proof*⟩

lemma *dvd-0-left*: $0\ dvd\ a \Longrightarrow a = 0$
⟨*proof*⟩

lemma *dvd-add* [*simp*]:
assumes *a dvd b* **and** *a dvd c*
shows $a\ dvd\ (b + c)$
⟨*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*
+
assumes *right-diff-distrib'* [*algebra-simps*]: $a * (b - c) = a * b - a * c$
begin

subclass *semiring-1-cancel* ⟨*proof*⟩
subclass *comm-semiring-0-cancel* ⟨*proof*⟩
subclass *comm-semiring-1* ⟨*proof*⟩

lemma *left-diff-distrib'* [*algebra-simps*]: $(b - c) * a = b * a - c * a$
⟨*proof*⟩

lemma *dvd-add-times-triv-left-iff* [*simp*]: a *dvd* $c * a + b \longleftrightarrow a$ *dvd* b
⟨*proof*⟩

lemma *dvd-add-times-triv-right-iff* [*simp*]: a *dvd* $b + c * a \longleftrightarrow a$ *dvd* b
⟨*proof*⟩

lemma *dvd-add-triv-left-iff* [*simp*]: a *dvd* $a + b \longleftrightarrow a$ *dvd* b
⟨*proof*⟩

lemma *dvd-add-triv-right-iff* [*simp*]: a *dvd* $b + a \longleftrightarrow a$ *dvd* b
⟨*proof*⟩

lemma *dvd-add-right-iff*:
assumes a *dvd* b
shows a *dvd* $b + c \longleftrightarrow a$ *dvd* c (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *dvd-add-left-iff*: a *dvd* $c \Longrightarrow a$ *dvd* $b + c \longleftrightarrow a$ *dvd* b
⟨*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*] = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* [*simp*] = *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*]: $a * (b - c) = a * b - a * c$
⟨*proof*⟩

lemma *left-diff-distrib* [*algebra-simps*]: $(a - b) * c = a * c - b * c$
⟨*proof*⟩

lemmas *ring-distribs* = *distrib-left distrib-right 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* = *distrib-left distrib-right left-diff-distrib right-diff-distrib*

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* ⟨*proof*⟩
subclass *comm-semiring-0-cancel* ⟨*proof*⟩

lemma *square-diff-square-factored*: $x * x - y * y = (x + y) * (x - y)$
⟨*proof*⟩

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* ⟨*proof*⟩

lemma *square-diff-one-factored*: $x * x - 1 = (x + 1) * (x - 1)$
⟨*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* $- y \longleftrightarrow x$ *dvd* y
⟨*proof*⟩

lemma *minus-dvd-iff* [*simp*]: $- x$ *dvd* $y \longleftrightarrow x$ *dvd* y
⟨*proof*⟩

lemma *dvd-diff* [*simp*]: x *dvd* $y \Longrightarrow x$ *dvd* $z \Longrightarrow x$ *dvd* $(y - z)$
⟨*proof*⟩

end

class *semiring-no-zero-divisors* = *semiring-0* +
assumes *no-zero-divisors*: $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$
begin

lemma *divisors-zero*:
assumes $a * b = 0$
shows $a = 0 \lor b = 0$
⟨*proof*⟩

lemma *mult-eq-0-iff* [*simp*]: $a * b = 0 \longleftrightarrow a = 0 \lor b = 0$
⟨*proof*⟩

end

class *semiring-1-no-zero-divisors* = *semiring-1* + *semiring-no-zero-divisors*

class *semiring-no-zero-divisors-cancel* = *semiring-no-zero-divisors* +
assumes *mult-cancel-right* [*simp*]: $a * c = b * c \longleftrightarrow c = 0 \lor a = b$
and *mult-cancel-left* [*simp*]: $c * a = c * b \longleftrightarrow c = 0 \lor a = b$
begin

lemma *mult-left-cancel*: $c \neq 0 \Longrightarrow c * a = c * b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *mult-right-cancel*: $c \neq 0 \Longrightarrow a * c = b * c \longleftrightarrow a = b$
⟨*proof*⟩

end

class *ring-no-zero-divisors* = *ring* + *semiring-no-zero-divisors*
begin

subclass *semiring-no-zero-divisors-cancel*
⟨*proof*⟩

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

subclass *semiring-1-no-zero-divisors* ⟨*proof*⟩

lemma *square-eq-1-iff*: $x * x = 1 \longleftrightarrow x = 1 \lor x = - 1$
⟨*proof*⟩

lemma *mult-cancel-right1* [*simp*]: $c = b * c \longleftrightarrow c = 0 \lor b = 1$
⟨*proof*⟩

lemma *mult-cancel-right2* [*simp*]: $a * c = c \longleftrightarrow c = 0 \lor a = 1$
⟨*proof*⟩

lemma *mult-cancel-left1* [*simp*]: $c = c * b \longleftrightarrow c = 0 \lor b = 1$
⟨*proof*⟩

lemma *mult-cancel-left2* [*simp*]: $c * a = c \longleftrightarrow c = 0 \lor a = 1$
⟨*proof*⟩

end

class *semidom* = *comm-semiring-1-cancel* + *semiring-no-zero-divisors*
begin

subclass *semiring-1-no-zero-divisors* ⟨*proof*⟩

end

class *idom* = *comm-ring-1* + *semiring-no-zero-divisors*
begin

subclass *semidom* ⟨*proof*⟩

subclass *ring-1-no-zero-divisors* ⟨*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*⟩

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow a = b \vee a = - b$
⟨*proof*⟩

end

class *idom-abs-sgn* = *idom* + *abs* + *sgn* +
  assumes *sgn-mult-abs*: $sgn\ a * |a| = a$
    and *sgn-sgn* [*simp*]: $sgn\ (sgn\ a) = sgn\ a$
    and *abs-abs* [*simp*]: $||a|| = |a|$
    and *abs-0* [*simp*]: $|0| = 0$
    and *sgn-0* [*simp*]: $sgn\ 0 = 0$
    and *sgn-1* [*simp*]: $sgn\ 1 = 1$
    and *sgn-minus-1*: $sgn\ (- 1) = - 1$
    and *sgn-mult*: $sgn\ (a * b) = sgn\ a * sgn\ b$
begin

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

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

lemma *abs-mult-sgn*:
  $|a| * sgn\ a = a$
  ⟨*proof*⟩

lemma *abs-1* [*simp*]:
  $|1| = 1$
  ⟨*proof*⟩

lemma *sgn-abs* [*simp*]:
  $|sgn\ a| = of\text{-}bool\ (a \neq 0)$
  ⟨*proof*⟩

lemma *abs-sgn* [*simp*]:
  $sgn\ |a| = of\text{-}bool\ (a \neq 0)$
  ⟨*proof*⟩

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

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

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

end

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

Syntactic division operator

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

⟨*ML*⟩

context *semiring*
begin

lemma [*field-simps*]:
shows *distrib-left-NO-MATCH*: $NO\text{-}MATCH\ (x\ div\ y)\ a \Longrightarrow a * (b + c) = a * b + a * c$
and *distrib-right-NO-MATCH*: $NO\text{-}MATCH\ (x\ div\ y)\ c \Longrightarrow (a + b) * c = a * c + b * c$
⟨*proof*⟩

end

context *ring*
begin

lemma [*field-simps*]:
shows *left-diff-distrib-NO-MATCH*: *NO-MATCH* $(x\ div\ y)\ c \Longrightarrow (a - b) * c = a * c - b * c$
and *right-diff-distrib-NO-MATCH*: *NO-MATCH* $(x\ div\ y)\ a \Longrightarrow a * (b - c) = a * b - a * c$
⟨*proof*⟩

end

⟨*ML*⟩

Algebraic classes with division

class *semidom-divide* = *semidom* + *divide* +
assumes *nonzero-mult-div-cancel-right* [*simp*]: $b \neq 0 \Longrightarrow (a * b)\ div\ b = a$
assumes *div-by-0* [*simp*]: $a\ div\ 0 = 0$
begin

lemma *nonzero-mult-div-cancel-left* [*simp*]: $a \neq 0 \Longrightarrow (a * b)\ div\ a = b$
⟨*proof*⟩

subclass *semiring-no-zero-divisors-cancel*
⟨*proof*⟩

lemma *div-self* [*simp*]: $a \neq 0 \Longrightarrow a\ div\ a = 1$
⟨*proof*⟩

lemma *div-0* [*simp*]: $0\ div\ a = 0$
⟨*proof*⟩

lemma *div-by-1* [*simp*]: $a\ div\ 1 = a$
⟨*proof*⟩

lemma *dvd-div-eq-0-iff*:
assumes $b\ dvd\ a$
shows $a\ div\ b = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *dvd-div-eq-cancel*:
$a\ div\ c = b\ div\ c \Longrightarrow c\ dvd\ a \Longrightarrow c\ dvd\ b \Longrightarrow a = b$
⟨*proof*⟩

lemma *dvd-div-eq-iff*:
$c\ dvd\ a \Longrightarrow c\ dvd\ b \Longrightarrow a\ div\ c = b\ div\ c \longleftrightarrow a = b$
⟨*proof*⟩

end

class *idom-divide* = *idom* + *semidom-divide*

begin

lemma *dvd-neg-div*:
  assumes *b dvd a*
  shows $-$ *a div b* $= -$ (*a div b*)
⟨*proof*⟩

lemma *dvd-div-neg*:
  assumes *b dvd a*
  shows *a div* $-$ *b* $= -$ (*a div b*)
⟨*proof*⟩

end

class *algebraic-semidom* = *semidom-divide*
begin

Class *algebraic-semidom* enriches a integral domain by notions from algebra, like units in a ring. It is a separate class to avoid spoiling fields with notions which are degenerated there.

lemma *dvd-times-left-cancel-iff* [*simp*]:
  assumes $a \neq 0$
  shows *a* $*$ *b dvd a* $*$ *c* $\longleftrightarrow$ *b dvd c*
    (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *dvd-times-right-cancel-iff* [*simp*]:
  assumes $a \neq 0$
  shows *b* $*$ *a dvd c* $*$ *a* $\longleftrightarrow$ *b dvd c*
  ⟨*proof*⟩

lemma *div-dvd-iff-mult*:
  assumes $b \neq 0$ **and** *b dvd a*
  shows *a div b dvd c* $\longleftrightarrow$ *a dvd c* $*$ *b*
⟨*proof*⟩

lemma *dvd-div-iff-mult*:
  assumes $c \neq 0$ **and** *c dvd b*
  shows *a dvd b div c* $\longleftrightarrow$ *a* $*$ *c dvd b*
⟨*proof*⟩

lemma *div-dvd-div* [*simp*]:
  assumes *a dvd b* **and** *a dvd c*
  shows *b div a dvd c div a* $\longleftrightarrow$ *b dvd c*
⟨*proof*⟩

lemma *div-add* [*simp*]:
  assumes *c dvd a* **and** *c dvd b*
  shows (*a* + *b*) *div c* = *a div c* + *b div c*

$\langle proof \rangle$

lemma *div-mult-div-if-dvd*:
assumes b *dvd* a **and** d *dvd* c
shows $(a\ div\ b) * (c\ div\ d) = (a * c)\ div\ (b * d)$
$\langle proof \rangle$

lemma *dvd-div-eq-mult*:
assumes $a \neq 0$ **and** a *dvd* b
shows $b\ div\ a = c \longleftrightarrow b = c * a$
(**is** $?lhs \longleftrightarrow ?rhs$)
$\langle proof \rangle$

lemma *dvd-div-mult-self* [*simp*]: $a\ dvd\ b \Longrightarrow b\ div\ a * a = b$
$\langle proof \rangle$

lemma *dvd-mult-div-cancel* [*simp*]: $a\ dvd\ b \Longrightarrow a * (b\ div\ a) = b$
$\langle proof \rangle$

lemma *div-mult-swap*:
assumes c *dvd* b
shows $a * (b\ div\ c) = (a * b)\ div\ c$
$\langle proof \rangle$

lemma *dvd-div-mult*: $c\ dvd\ b \Longrightarrow b\ div\ c * a = (b * a)\ div\ c$
$\langle proof \rangle$

lemma *dvd-div-mult2-eq*:
assumes $b * c$ *dvd* a
shows $a\ div\ (b * c) = a\ div\ b\ div\ c$
$\langle proof \rangle$

lemma *dvd-div-div-eq-mult*:
assumes $a \neq 0$ $c \neq 0$ **and** a *dvd* b c *dvd* d
shows $b\ div\ a = d\ div\ c \longleftrightarrow b * c = a * d$
(**is** $?lhs \longleftrightarrow ?rhs$)
$\langle proof \rangle$

lemma *dvd-mult-imp-div*:
assumes $a * c$ *dvd* b
shows a *dvd* $b\ div\ c$
$\langle proof \rangle$

lemma *div-div-eq-right*:
assumes c *dvd* b b *dvd* a
shows $a\ div\ (b\ div\ c) = a\ div\ b * c$
$\langle proof \rangle$

lemma *div-div-div-same*:

assumes *d dvd b b dvd a*
shows $(a\ div\ d)\ div\ (b\ div\ d) = a\ div\ b$
⟨*proof*⟩

Units: invertible elements in a ring

abbreviation *is-unit* :: $'a \Rightarrow bool$
where *is-unit* $a \equiv a\ dvd\ 1$

lemma *not-is-unit-0* [*simp*]: $\neg$ *is-unit 0*
⟨*proof*⟩

lemma *unit-imp-dvd* [*dest*]: *is-unit b* $\Longrightarrow$ *b dvd a*
⟨*proof*⟩

lemma *unit-dvdE*:
assumes *is-unit a*
obtains *c* **where** $a \neq 0$ **and** $b = a * c$
⟨*proof*⟩

lemma *dvd-unit-imp-unit*: *a dvd b* $\Longrightarrow$ *is-unit b* $\Longrightarrow$ *is-unit a*
⟨*proof*⟩

lemma *unit-div-1-unit* [*simp*, *intro*]:
assumes *is-unit a*
shows *is-unit* (*1 div a*)
⟨*proof*⟩

lemma *is-unitE* [*elim?*]:
assumes *is-unit a*
obtains *b* **where** $a \neq 0$ **and** $b \neq 0$
and *is-unit b* **and** $1\ div\ a = b$ **and** $1\ div\ b = a$
and $a * b = 1$ **and** $c\ div\ a = c * b$
⟨*proof*⟩

lemma *unit-prod* [*intro*]: *is-unit a* $\Longrightarrow$ *is-unit b* $\Longrightarrow$ *is-unit* $(a * b)$
⟨*proof*⟩

lemma *is-unit-mult-iff*: *is-unit* $(a * b) \longleftrightarrow$ *is-unit a* $\wedge$ *is-unit b*
⟨*proof*⟩

lemma *unit-div* [*intro*]: *is-unit a* $\Longrightarrow$ *is-unit b* $\Longrightarrow$ *is-unit* (*a div b*)
⟨*proof*⟩

lemma *mult-unit-dvd-iff*:
assumes *is-unit b*
shows $a * b\ dvd\ c \longleftrightarrow a\ dvd\ c$
⟨*proof*⟩

lemma *mult-unit-dvd-iff'*: *is-unit a* $\Longrightarrow (a * b)\ dvd\ c \longleftrightarrow b\ dvd\ c$

⟨*proof*⟩

lemma *dvd-mult-unit-iff*:
assumes *is-unit b*
shows $a\ dvd\ c * b \longleftrightarrow a\ dvd\ c$
⟨*proof*⟩

lemma *dvd-mult-unit-iff'*: $is\text{-}unit\ b \Longrightarrow a\ dvd\ b * c \longleftrightarrow a\ dvd\ c$
⟨*proof*⟩

lemma *div-unit-dvd-iff*: $is\text{-}unit\ b \Longrightarrow a\ div\ b\ dvd\ c \longleftrightarrow a\ dvd\ c$
⟨*proof*⟩

lemma *dvd-div-unit-iff*: $is\text{-}unit\ b \Longrightarrow a\ dvd\ c\ div\ b \longleftrightarrow a\ dvd\ c$
⟨*proof*⟩

lemmas *unit-dvd-iff* = *mult-unit-dvd-iff mult-unit-dvd-iff'*
dvd-mult-unit-iff dvd-mult-unit-iff'
div-unit-dvd-iff dvd-div-unit-iff

lemma *unit-mult-div-div* [*simp*]: $is\text{-}unit\ a \Longrightarrow b * (1\ div\ a) = b\ div\ a$
⟨*proof*⟩

lemma *unit-div-mult-self* [*simp*]: $is\text{-}unit\ a \Longrightarrow b\ div\ a * a = b$
⟨*proof*⟩

lemma *unit-div-1-div-1* [*simp*]: $is\text{-}unit\ a \Longrightarrow 1\ div\ (1\ div\ a) = a$
⟨*proof*⟩

lemma *unit-div-mult-swap*: $is\text{-}unit\ c \Longrightarrow a * (b\ div\ c) = (a * b)\ div\ c$
⟨*proof*⟩

lemma *unit-div-commute*: $is\text{-}unit\ b \Longrightarrow (a\ div\ b) * c = (a * c)\ div\ b$
⟨*proof*⟩

lemma *unit-eq-div1*: $is\text{-}unit\ b \Longrightarrow a\ div\ b = c \longleftrightarrow a = c * b$
⟨*proof*⟩

lemma *unit-eq-div2*: $is\text{-}unit\ b \Longrightarrow a = c\ div\ b \longleftrightarrow a * b = c$
⟨*proof*⟩

lemma *unit-mult-left-cancel*: $is\text{-}unit\ a \Longrightarrow a * b = a * c \longleftrightarrow b = c$
⟨*proof*⟩

lemma *unit-mult-right-cancel*: $is\text{-}unit\ a \Longrightarrow b * a = c * a \longleftrightarrow b = c$
⟨*proof*⟩

lemma *unit-div-cancel*:
assumes *is-unit a*

shows b *div* $a = c$ *div* $a \longleftrightarrow b = c$
⟨*proof*⟩

lemma *is-unit-div-mult2-eq*:
assumes *is-unit* b **and** *is-unit* c
shows a *div* $(b * c) = a$ *div* b *div* c
⟨*proof*⟩

lemmas *unit-simps* = *mult-unit-dvd-iff div-unit-dvd-iff dvd-mult-unit-iff dvd-div-unit-iff unit-div-mult-swap unit-div-commute unit-mult-left-cancel unit-mult-right-cancel unit-div-cancel unit-eq-div1 unit-eq-div2*

lemma *is-unit-div-mult-cancel-left*:
assumes $a \neq 0$ **and** *is-unit* b
shows a *div* $(a * b) = 1$ *div* b
⟨*proof*⟩

lemma *is-unit-div-mult-cancel-right*:
assumes $a \neq 0$ **and** *is-unit* b
shows a *div* $(b * a) = 1$ *div* b
⟨*proof*⟩

lemma *unit-div-eq-0-iff*:
assumes *is-unit* b
shows a *div* $b = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *div-mult-unit2*:
is-unit $c \Longrightarrow b$ *dvd* $a \Longrightarrow a$ *div* $(b * c) = a$ *div* b *div* c
⟨*proof*⟩

end

class *unit-factor* =
fixes *unit-factor* :: $'a \Rightarrow 'a$

class *semidom-divide-unit-factor* = *semidom-divide* + *unit-factor* +
assumes *unit-factor-0* [*simp*]: *unit-factor* $0 = 0$
and *is-unit-unit-factor*: a *dvd* $1 \Longrightarrow$ *unit-factor* $a = a$
and *unit-factor-is-unit*: $a \neq 0 \Longrightarrow$ *unit-factor* a *dvd* 1
and *unit-factor-mult*: *unit-factor* $(a * b)$ = *unit-factor* a * *unit-factor* b
— This fine-grained hierarchy will later on allow lean normalization of polynomials

class *normalization-semidom* = *algebraic-semidom* + *semidom-divide-unit-factor* +
fixes *normalize* :: $'a \Rightarrow 'a$
assumes *unit-factor-mult-normalize* [*simp*]: *unit-factor* a * *normalize* $a = a$

and *normalize-0* [*simp*]: *normalize 0 = 0*
begin

Class *normalization-semidom* cultivates the idea that each integral domain can be split into equivalence classes whose representants are associated, i.e. divide each other. *normalize* specifies a canonical representant for each equivalence class. The rationale behind this is that it is easier to reason about equality than equivalences, hence we prefer to think about equality of normalized values rather than associated elements.

declare *unit-factor-is-unit* [*iff*]

lemma *unit-factor-dvd* [*simp*]: $a \neq 0 \Longrightarrow$ *unit-factor a dvd b*
⟨*proof*⟩

lemma *unit-factor-self* [*simp*]: *unit-factor a dvd a*
⟨*proof*⟩

lemma *normalize-mult-unit-factor* [*simp*]: *normalize a* $*$ *unit-factor a = a*
⟨*proof*⟩

lemma *normalize-eq-0-iff* [*simp*]: *normalize a = 0* $\longleftrightarrow$ *a = 0*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *unit-factor-eq-0-iff* [*simp*]: *unit-factor a = 0* $\longleftrightarrow$ *a = 0*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *div-unit-factor* [*simp*]: *a div unit-factor a = normalize a*
⟨*proof*⟩

lemma *normalize-div* [*simp*]: *normalize a div a = 1 div unit-factor a*
⟨*proof*⟩

lemma *is-unit-normalize*:
assumes *is-unit a*
shows *normalize a = 1*
⟨*proof*⟩

lemma *unit-factor-1* [*simp*]: *unit-factor 1 = 1*
⟨*proof*⟩

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

lemma *normalize-1-iff*: *normalize a = 1* $\longleftrightarrow$ *is-unit a*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *div-normalize* [*simp*]: *a div normalize a* = *unit-factor a*
⟨*proof*⟩

lemma *mult-one-div-unit-factor* [*simp*]: *a* ∗ (*1 div unit-factor b*) = *a div unit-factor b*
⟨*proof*⟩

lemma *inv-unit-factor-eq-0-iff* [*simp*]:
1 div unit-factor a = *0* ⟷ *a* = *0*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *normalize-mult*: *normalize* (*a* ∗ *b*) = *normalize a* ∗ *normalize b*
⟨*proof*⟩

lemma *unit-factor-idem* [*simp*]: *unit-factor* (*unit-factor a*) = *unit-factor a*
⟨*proof*⟩

lemma *normalize-unit-factor* [*simp*]: *a* ≠ *0* ⟹ *normalize* (*unit-factor a*) = *1*
⟨*proof*⟩

lemma *normalize-idem* [*simp*]: *normalize* (*normalize a*) = *normalize a*
⟨*proof*⟩

lemma *unit-factor-normalize* [*simp*]:
assumes *a* ≠ *0*
shows *unit-factor* (*normalize a*) = *1*
⟨*proof*⟩

lemma *dvd-unit-factor-div*:
assumes *b dvd a*
shows *unit-factor* (*a div b*) = *unit-factor a div unit-factor b*
⟨*proof*⟩

lemma *dvd-normalize-div*:
assumes *b dvd a*
shows *normalize* (*a div b*) = *normalize a div normalize b*
⟨*proof*⟩

lemma *normalize-dvd-iff* [*simp*]: *normalize a dvd b* ⟷ *a dvd b*
⟨*proof*⟩

lemma *dvd-normalize-iff* [*simp*]: *a dvd normalize b* ⟷ *a dvd b*
⟨*proof*⟩

lemma *normalize-idem-imp-unit-factor-eq*:
assumes *normalize a* = *a*
shows *unit-factor a* = *of-bool* (*a* ≠ *0*)
⟨*proof*⟩

lemma *normalize-idem-imp-is-unit-iff*:
  assumes *normalize a = a*
  shows *is-unit a* $\longleftrightarrow$ *a = 1*
  ⟨*proof*⟩

We avoid an explicit definition of associated elements but prefer explicit normalisation instead. In theory we could define an abbreviation like *associated a b* = (*normalize a* = *normalize b*) but this is counterproductive without suggestive infix syntax, which we do not want to sacrifice for this purpose here.

lemma *associatedI*:
  assumes *a dvd b* **and** *b dvd a*
  shows *normalize a = normalize b*
⟨*proof*⟩

lemma *associatedD1*: *normalize a = normalize b* $\Longrightarrow$ *a dvd b*
  ⟨*proof*⟩

lemma *associatedD2*: *normalize a = normalize b* $\Longrightarrow$ *b dvd a*
  ⟨*proof*⟩

lemma *associated-unit*: *normalize a = normalize b* $\Longrightarrow$ *is-unit a* $\Longrightarrow$ *is-unit b*
  ⟨*proof*⟩

lemma *associated-iff-dvd*: *normalize a = normalize b* $\longleftrightarrow$ *a dvd b* $\wedge$ *b dvd a*
  (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *associated-eqI*:
  assumes *a dvd b* **and** *b dvd a*
  assumes *normalize a = a* **and** *normalize b = b*
  shows *a = b*
⟨*proof*⟩

lemma *normalize-unit-factor-eqI*:
  assumes *normalize a = normalize b*
    and *unit-factor a = unit-factor b*
  shows *a = b*
⟨*proof*⟩

end

Syntactic division remainder operator

class *modulo = dvd + divide +*
  fixes *modulo* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *mod 70*)

Arbitrary quotient and remainder partitions

class *semiring-modulo = comm-semiring-1-cancel + divide + modulo +*

assumes *div-mult-mod-eq*: $a\ div\ b * b + a\ mod\ b = a$
begin

lemma *mod-div-decomp*:
fixes $a\ b$
obtains $q\ r$ **where** $q = a\ div\ b$ **and** $r = a\ mod\ b$
and $a = q * b + r$
⟨*proof*⟩

lemma *mult-div-mod-eq*: $b * (a\ div\ b) + a\ mod\ b = a$
⟨*proof*⟩

lemma *mod-div-mult-eq*: $a\ mod\ b + a\ div\ b * b = a$
⟨*proof*⟩

lemma *mod-mult-div-eq*: $a\ mod\ b + b * (a\ div\ b) = a$
⟨*proof*⟩

lemma *minus-div-mult-eq-mod*: $a - a\ div\ b * b = a\ mod\ b$
⟨*proof*⟩

lemma *minus-mult-div-eq-mod*: $a - b * (a\ div\ b) = a\ mod\ b$
⟨*proof*⟩

lemma *minus-mod-eq-div-mult*: $a - a\ mod\ b = a\ div\ b * b$
⟨*proof*⟩

lemma *minus-mod-eq-mult-div*: $a - a\ mod\ b = b * (a\ div\ b)$
⟨*proof*⟩

end

class *ordered-semiring* = *semiring* + *ordered-comm-monoid-add* +
assumes *mult-left-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
assumes *mult-right-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * c$
begin

lemma *mult-mono*: $a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * d$
⟨*proof*⟩

lemma *mult-mono'*: $a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq a \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * d$
⟨*proof*⟩

end

class *ordered-semiring-0* = *semiring-0* + *ordered-semiring*
begin

lemma *mult-nonneg-nonneg* [*simp*]: $0 \le a \Longrightarrow 0 \le b \Longrightarrow 0 \le a * b$
⟨*proof*⟩

lemma *mult-nonneg-nonpos*: $0 \le a \Longrightarrow b \le 0 \Longrightarrow a * b \le 0$
⟨*proof*⟩

lemma *mult-nonpos-nonneg*: $a \le 0 \Longrightarrow 0 \le b \Longrightarrow a * b \le 0$
⟨*proof*⟩

Legacy – use *mult-nonpos-nonneg*.

lemma *mult-nonneg-nonpos2*: $0 \le a \Longrightarrow b \le 0 \Longrightarrow b * a \le 0$
⟨*proof*⟩

lemma *split-mult-neg-le*: $(0 \le a \wedge b \le 0) \vee (a \le 0 \wedge 0 \le b) \Longrightarrow a * b \le 0$
⟨*proof*⟩

end

class *ordered-cancel-semiring* = *ordered-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *ordered-semiring-0* ⟨*proof*⟩

end

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

subclass *ordered-cancel-semiring* ⟨*proof*⟩

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

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

lemma *mult-left-less-imp-less*: $c * a < c * b \Longrightarrow 0 \le c \Longrightarrow a < b$
⟨*proof*⟩

lemma *mult-right-less-imp-less*: $a * c < b * c \Longrightarrow 0 \le c \Longrightarrow a < b$
⟨*proof*⟩

end

class *linordered-semiring-1* = *linordered-semiring* + *semiring-1*
begin

lemma *convex-bound-le*:
assumes $x \le a$ $y \le a$ $0 \le u$ $0 \le v$ $u + v = 1$

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*[*simp*]: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
⟨*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*⟩

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 *ordered-comm-semiring* = *comm-semiring-0* + *ordered-ab-semigroup-add* +
assumes *comm-mult-left-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
begin

subclass *ordered-semiring*
⟨*proof*⟩

end

class *ordered-cancel-comm-semiring* = *ordered-comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *comm-semiring-0-cancel* ⟨*proof*⟩
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 *comm-mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$

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

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*⟩

proposition *abs-eq-iff*: $|x| = |y| \longleftrightarrow x = y \vee x = -y$
⟨*proof*⟩

lemma *abs-eq-iff'*:
$|a| = b \longleftrightarrow b \geq 0 \wedge (a = b \vee a = - b)$
⟨*proof*⟩

lemma *eq-abs-iff'*:
$a = |b| \longleftrightarrow a \geq 0 \wedge (b = a \vee b = - a)$
⟨*proof*⟩

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

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 *zero-less-one* = *order* + *zero* + *one* +
assumes *zero-less-one* [*simp*]: $0 < 1$

class *linordered-nonzero-semiring* = *ordered-comm-semiring* + *monoid-mult* + *linorder* + *zero-less-one*
begin

subclass *zero-neq-one*
⟨*proof*⟩

subclass *comm-semiring-1*
⟨*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 *mult-left-le*: $c \leq 1 \Longrightarrow 0 \leq a \Longrightarrow a * c \leq a$
⟨*proof*⟩

lemma *mult-le-one*: $a \leq 1 \Longrightarrow 0 \leq b \Longrightarrow b \leq 1 \Longrightarrow a * b \leq 1$
⟨*proof*⟩

lemma *zero-less-two*: $0 < 1 + 1$

⟨*proof*⟩

end

class *linordered-semidom* = *semidom* + *linordered-comm-semiring-strict* + *zero-less-one* +
assumes *le-add-diff-inverse2* [*simp*]: $b \leq a \Longrightarrow a - b + b = a$
begin

subclass *linordered-nonzero-semiring* ⟨*proof*⟩

Addition is the inverse of subtraction.

lemma *le-add-diff-inverse* [*simp*]: $b \leq a \Longrightarrow b + (a - b) = a$
⟨*proof*⟩

lemma *add-diff-inverse*: $\neg\ a < b \Longrightarrow b + (a - b) = a$
⟨*proof*⟩

lemma *add-le-imp-le-diff*: $i + k \leq n \Longrightarrow i \leq n - k$
⟨*proof*⟩

lemma *add-le-add-imp-diff-le*:
assumes *1*: $i + k \leq n$
and *2*: $n \leq j + k$
shows $i + k \leq n \Longrightarrow n \leq j + k \Longrightarrow n - k \leq j$
⟨*proof*⟩

lemma *less-1-mult*: $1 < m \Longrightarrow 1 < n \Longrightarrow 1 < m * n$
⟨*proof*⟩

end

class *linordered-idom* =
comm-ring-1 + *linordered-comm-semiring-strict* + *ordered-ab-group-add* + *abs-if* + *sgn* +
assumes *sgn-if*: *sgn* x = (*if* $x = 0$ *then* 0 *else if* $0 < x$ *then* 1 *else* $- 1$)
begin

subclass *linordered-semiring-1-strict* ⟨*proof*⟩
subclass *linordered-ring-strict* ⟨*proof*⟩
subclass *ordered-comm-ring* ⟨*proof*⟩
subclass *idom* ⟨*proof*⟩

subclass *linordered-semidom*
⟨*proof*⟩

subclass *idom-abs-sgn*
⟨*proof*⟩

lemma *linorder-neqE-linordered-idom*:
assumes $x \neq y$
obtains $x < y \mid y < x$
⟨*proof*⟩

These cancellation simp rules 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-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 *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-sgn-eq-1* [*simp*]:
$a \neq 0 \Longrightarrow |sgn\ a| = 1$
⟨*proof*⟩

lemma *abs-sgn-eq*: $|sgn\ a| = (if\ a = 0\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *sgn-mult-self-eq* [*simp*]:
$sgn\ a * sgn\ a = of\text{-}bool\ (a \neq 0)$
⟨*proof*⟩

lemma *abs-mult-self-eq* [*simp*]:
$|a| * |a| = a * a$
⟨*proof*⟩

lemma *same-sgn-sgn-add*:
$sgn\ (a + b) = sgn\ a$ **if** $sgn\ b = sgn\ a$
⟨*proof*⟩

lemma *same-sgn-abs-add*:
$|a + b| = |a| + |b|$ **if** $sgn\ b = sgn\ a$
⟨*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*⟩

The following lemmas can be proven in more general structures, but are dangerous as simp rules in absence of $(-\ ?a = ?a) = (?a = (0::'a))$, $(-\ ?a < ?a) = ((0::'a) < ?a)$, $(-\ ?a \leq ?a) = ((0::'a) \leq ?a)$.

lemma *equation-minus-iff-1* [*simp*, *no-atp*]: $1 = -\ a \longleftrightarrow a = -\ 1$
⟨*proof*⟩

lemma *minus-equation-iff-1* [*simp*, *no-atp*]: $-\ a = 1 \longleftrightarrow a = -\ 1$
⟨*proof*⟩

lemma *le-minus-iff-1* [*simp*, *no-atp*]: $1 \le -\ b \longleftrightarrow b \le -\ 1$
⟨*proof*⟩

lemma *minus-le-iff-1* [*simp*, *no-atp*]: $-\ a \le 1 \longleftrightarrow -\ 1 \le a$
⟨*proof*⟩

lemma *less-minus-iff-1* [*simp*, *no-atp*]: $1 < -\ b \longleftrightarrow b < -\ 1$
⟨*proof*⟩

lemma *minus-less-iff-1* [*simp*, *no-atp*]: $-\ a < 1 \longleftrightarrow -\ 1 < a$
⟨*proof*⟩

end

Simprules for comparisons where common factors can be cancelled.

lemmas *mult-compare-simps* =
mult-le-cancel-right mult-le-cancel-left
mult-le-cancel-right1 mult-le-cancel-right2
mult-le-cancel-left1 mult-le-cancel-left2
mult-less-cancel-right mult-less-cancel-left
mult-less-cancel-right1 mult-less-cancel-right2
mult-less-cancel-left1 mult-less-cancel-left2
mult-cancel-right mult-cancel-left
mult-cancel-right1 mult-cancel-right2
mult-cancel-left1 mult-cancel-left2

Reasoning about inequalities with division

context *linordered-semidom*
begin

lemma *less-add-one*: $a < a + 1$
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *mult-right-le-one-le*: $0 \le x \Longrightarrow 0 \le y \Longrightarrow y \le 1 \Longrightarrow x * y \le x$
⟨*proof*⟩

lemma *mult-left-le-one-le*: $0 \le x \Longrightarrow 0 \le y \Longrightarrow y \le 1 \Longrightarrow y * x \le x$
⟨*proof*⟩

end

Absolute Value

context *linordered-idom*
begin

lemma *mult-sgn-abs*: $sgn\ x * |x| = x$
⟨*proof*⟩

lemma *abs-one*: $|1| = 1$
⟨*proof*⟩

end

class *ordered-ring-abs* = *ordered-ring* + *ordered-ab-group-add-abs* +
assumes *abs-eq-mult*:
$(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \Longrightarrow |a * b| = |a| * |b|$

context *linordered-idom*
begin

subclass *ordered-ring-abs*
⟨*proof*⟩

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

lemma *abs-mult-less*:
assumes *ac*: $|a| < c$
and *bd*: $|b| < d$
shows $|a| * |b| < c * d$
⟨*proof*⟩

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

lemma *abs-mult-pos*: $0 \leq x \Longrightarrow |y| * x = |y * x|$
⟨*proof*⟩

lemma *abs-diff-less-iff*: $|x - a| < r \longleftrightarrow a - r < x \wedge x < a + r$
⟨*proof*⟩

lemma *abs-diff-le-iff*: $|x - a| \leq r \longleftrightarrow a - r \leq x \wedge x \leq a + r$
⟨*proof*⟩

lemma *abs-add-one-gt-zero*: $0 < 1 + |x|$
⟨*proof*⟩

end

15.1 Dioids

Dioids are the alternative extensions of semirings, a semiring can either be a ring or a dioid but never both.

class *dioid* = *semiring-1* + *canonically-ordered-monoid-add*
begin

subclass *ordered-semiring*
⟨*proof*⟩

end

hide-fact (**open**) *comm-mult-left-mono comm-mult-strict-left-mono distrib*

code-identifier
code-module *Rings* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

16 Natural numbers

theory *Nat*
imports *Inductive Typedef Fun Rings*
begin

named-theorems *arith arith facts* −− *only ground formulas*
⟨*ML*⟩

16.1 Type *ind*

typedecl *ind*

axiomatization *Zero-Rep* :: *ind* **and** *Suc-Rep* :: *ind* ⇒ *ind*
— The axiom of infinity in 2 parts:
where *Suc-Rep-inject*: $Suc\text{-}Rep\ x = Suc\text{-}Rep\ y \Longrightarrow x = y$
and *Suc-Rep-not-Zero-Rep*: $Suc\text{-}Rep\ x \neq Zero\text{-}Rep$

16.2 Type nat

Type definition

inductive *Nat* :: *ind* ⇒ *bool*
where
Zero-RepI: *Nat Zero-Rep*
| *Suc-RepI*: $Nat\ i \Longrightarrow Nat\ (Suc\text{-}Rep\ i)$

typedef *nat* = $\{n.\ Nat\ n\}$
morphisms *Rep-Nat Abs-Nat*
⟨*proof*⟩

lemma *Nat-Rep-Nat*: $Nat\ (Rep\text{-}Nat\ n)$
⟨*proof*⟩

lemma *Nat-Abs-Nat-inverse*: $Nat\ n \Longrightarrow Rep\text{-}Nat\ (Abs\text{-}Nat\ n) = n$
⟨*proof*⟩

lemma *Nat-Abs-Nat-inject*: $Nat\ n \Longrightarrow Nat\ m \Longrightarrow Abs\text{-}Nat\ n = Abs\text{-}Nat\ m \longleftrightarrow n = m$
⟨*proof*⟩

instantiation *nat* :: *zero*
begin

definition *Zero-nat-def*: $0 = Abs\text{-}Nat\ Zero\text{-}Rep$

instance ⟨*proof*⟩

end

definition *Suc* :: $nat \Rightarrow nat$
where $Suc\ n = Abs\text{-}Nat\ (Suc\text{-}Rep\ (Rep\text{-}Nat\ n))$

lemma *Suc-not-Zero*: $Suc\ m \neq 0$
⟨*proof*⟩

lemma *Zero-not-Suc*: $0 \neq Suc\ m$
⟨*proof*⟩

lemma *Suc-Rep-inject'*: $Suc\text{-}Rep\ x = Suc\text{-}Rep\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *nat-induct0*:
assumes $P\ 0$
and $\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)$
shows $P\ n$
⟨*proof*⟩

free-constructors *case-nat* **for** $0 :: nat \mid Suc\ pred$
where $pred\ (0 :: nat) = (0 :: nat)$
⟨*proof*⟩
⟨*ML*⟩

old-rep-datatype $0 :: nat\ Suc$
⟨*proof*⟩

⟨*ML*⟩

declare *old.nat.inject*[*iff del*]
and *old.nat.distinct*(*1*)[*simp del*, *induct-simp del*]

lemmas *induct* = *old.nat.induct*
lemmas *inducts* = *old.nat.inducts*
lemmas *rec* = *old.nat.rec*
lemmas *simps* = *nat.inject nat.distinct nat.case nat.rec*

⟨*ML*⟩

abbreviation *rec-nat* :: $'a \Rightarrow (nat \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a$
where *rec-nat* ≡ *old.rec-nat*

declare *nat.sel*[*code del*]

hide-const (**open**) *Nat.pred* — hide everything related to the selector
hide-fact
nat.case-eq-if
nat.collapse
nat.expand
nat.sel
nat.exhaust-sel
nat.split-sel
nat.split-sel-asm

lemma *nat-exhaust* [*case-names 0 Suc*, *cases type*: *nat*]:
$(y = 0 \Longrightarrow P) \Longrightarrow (\bigwedge nat.\ y = Suc\ nat \Longrightarrow P) \Longrightarrow P$
— for backward compatibility – names of variables differ
⟨*proof*⟩

lemma *nat-induct* [*case-names 0 Suc*, *induct type*: *nat*]:
fixes *n*
assumes $P\ 0$ **and** $\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)$
shows $P\ n$
— for backward compatibility – names of variables differ
⟨*proof*⟩

hide-fact
nat-exhaust
nat-induct0

⟨*ML*⟩

Injectiveness and distinctness lemmas

lemma (**in** *semidom-divide*) *inj-times*:
inj (*times a*) **if** $a \neq 0$

⟨*proof*⟩

lemma (**in** *cancel-ab-semigroup-add*) *inj-plus*:
inj (*plus a*)
⟨*proof*⟩

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
⟨*proof*⟩

lemma *Suc-neq-Zero*: $Suc\ m = 0 \Longrightarrow R$
⟨*proof*⟩

lemma *Zero-neq-Suc*: $0 = Suc\ m \Longrightarrow R$
⟨*proof*⟩

lemma *Suc-inject*: $Suc\ x = Suc\ y \Longrightarrow x = y$
⟨*proof*⟩

lemma *n-not-Suc-n*: $n \neq Suc\ n$
⟨*proof*⟩

lemma *Suc-n-not-n*: $Suc\ n \neq n$
⟨*proof*⟩

A special form of induction for reasoning about $m < n$ and $m - n$.

lemma *diff-induct*:
assumes $\bigwedge x.\ P\ x\ 0$
and $\bigwedge y.\ P\ 0\ (Suc\ y)$
and $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (Suc\ x)\ (Suc\ y)$
shows $P\ m\ n$
⟨*proof*⟩

16.3 Arithmetic operators

instantiation *nat* :: *comm-monoid-diff*
begin

primrec *plus-nat*
where
add-0: $0 + n = (n::nat)$
| *add-Suc*: $Suc\ m + n = Suc\ (m + n)$

lemma *add-0-right* [*simp*]: $m + 0 = m$
for *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 [*code*]: *m − 0 = (m::nat)*
| *diff-Suc*: *m − Suc n = (case m − n of 0 ⇒ 0 | Suc k ⇒ k)*

declare *diff-Suc* [*simp del*]

lemma *diff-0-eq-0* [*simp*, *code*]: *0 − n = 0*
for *n* :: *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 ∗ 0 = 0*
for *m* :: *nat*
⟨*proof*⟩

lemma *mult-Suc-right* [*simp*]: *m ∗ Suc n = m + (m ∗ n)*
⟨*proof*⟩

lemma *add-mult-distrib*: *(m + n) ∗ k = (m ∗ k) + (n ∗ k)*
for *m n k* :: *nat*
⟨*proof*⟩

instance
⟨*proof*⟩

end

16.3.1 Addition

Reasoning about $m + 0 = 0$, etc.

lemma *add-is-0* [*iff*]: $m + n = 0 \longleftrightarrow m = 0 \land n = 0$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *add-is-1*: $m + n = Suc\ 0 \longleftrightarrow m = Suc\ 0 \land n = 0 \mid m = 0 \land n = Suc\ 0$
⟨*proof*⟩

lemma *one-is-add*: $Suc\ 0 = m + n \longleftrightarrow m = Suc\ 0 \land n = 0 \mid m = 0 \land n = Suc\ 0$
⟨*proof*⟩

lemma *add-eq-self-zero*: $m + n = m \Longrightarrow n = 0$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *inj-on-add-nat* [*simp*]: *inj-on* $(\lambda n.\ n + k)\ N$
for k :: *nat*
⟨*proof*⟩

lemma *Suc-eq-plus1*: $Suc\ n = n + 1$
⟨*proof*⟩

lemma *Suc-eq-plus1-left*: $Suc\ n = 1 + n$
⟨*proof*⟩

16.3.2 Difference

lemma *Suc-diff-diff* [*simp*]: $(Suc\ m - n) - Suc\ k = m - n - k$
⟨*proof*⟩

lemma *diff-Suc-1* [*simp*]: $Suc\ n - 1 = n$
⟨*proof*⟩

16.3.3 Multiplication

lemma *mult-is-0* [*simp*]: $m * n = 0 \longleftrightarrow m = 0 \lor n = 0$ **for** $m\ n$ :: *nat*
⟨*proof*⟩

lemma *mult-eq-1-iff* [*simp*]: $m * n = Suc\ 0 \longleftrightarrow m = Suc\ 0 \land n = Suc\ 0$
⟨*proof*⟩

lemma *one-eq-mult-iff* [*simp*]: $Suc\ 0 = m * n \longleftrightarrow m = Suc\ 0 \land n = Suc\ 0$
⟨*proof*⟩

lemma *nat-mult-eq-1-iff* [*simp*]: $m * n = 1 \longleftrightarrow m = 1 \wedge n = 1$
for $m\ n :: nat$
⟨*proof*⟩

lemma *nat-1-eq-mult-iff* [*simp*]: $1 = m * n \longleftrightarrow m = 1 \wedge n = 1$
for $m\ n :: nat$
⟨*proof*⟩

lemma *mult-cancel1* [*simp*]: $k * m = k * n \longleftrightarrow m = n \vee k = 0$
for $k\ m\ n :: nat$
⟨*proof*⟩

lemma *mult-cancel2* [*simp*]: $m * k = n * k \longleftrightarrow m = n \vee k = 0$
for $k\ m\ n :: nat$
⟨*proof*⟩

lemma *Suc-mult-cancel1*: $Suc\ k * m = Suc\ k * n \longleftrightarrow m = n$
⟨*proof*⟩

16.4 Orders on *nat*

16.4.1 Operation definition

instantiation $nat :: linorder$
begin

primrec *less-eq-nat*
where
$(0::nat) \leq n \longleftrightarrow True$
$\mid Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*]

lemma *le0* [*iff*]: $0 \leq n$ **for**
$n :: nat$
⟨*proof*⟩

lemma [*code*]: $0 \leq n \longleftrightarrow True$
for $n :: nat$
⟨*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*]: $n \leq 0 \longleftrightarrow n = 0$
  for n :: *nat*
  ⟨*proof*⟩

lemma *not-less0* [*iff*]: $\neg\ n < 0$
  for n :: *nat*
  ⟨*proof*⟩

lemma *less-nat-zero-code* [*code*]: $n < 0 \longleftrightarrow$ *False*
  for n :: *nat*
  ⟨*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 *Suc-less-eq2*: $Suc\ n < m \longleftrightarrow (\exists\, m'.\ m = Suc\ m' \wedge n < m')$
  ⟨*proof*⟩

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$
  ⟨*proof*⟩

lemma *Suc-leD*: $Suc\ m \leq n \Longrightarrow m \leq n$
  ⟨*proof*⟩

lemma *less-SucI*: $m < n \Longrightarrow m < Suc\ n$
  ⟨*proof*⟩

lemma *Suc-lessD*: $Suc\ m < n \Longrightarrow m < n$
  ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *nat* :: *order-bot*
begin

definition *bot-nat* :: *nat*
  where *bot-nat* = *0*

instance
  ⟨*proof*⟩

end

instance *nat* :: *no-top*
⟨*proof*⟩

16.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
⟨*proof*⟩

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
⟨*proof*⟩

16.4.3 Elimination properties

lemma *less-not-refl*: $\neg\ n < n$
for n :: *nat*
⟨*proof*⟩

lemma *less-not-refl2*: $n < m \Longrightarrow m \neq n$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *less-not-refl3*: $s < t \Longrightarrow s \neq t$
for $s\ t$ :: *nat*
⟨*proof*⟩

lemma *less-irrefl-nat*: $n < n \Longrightarrow R$
for n :: *nat*
⟨*proof*⟩

lemma *less-zeroE*: $n < 0 \Longrightarrow R$
for n :: *nat*
⟨*proof*⟩

lemma *less-Suc-eq*: $m < Suc\ n \longleftrightarrow m < n \lor m = n$
⟨*proof*⟩

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
⟨*proof*⟩

lemma *less-one* [*iff*]: $n < 1 \longleftrightarrow n = 0$
for n :: *nat*
⟨*proof*⟩

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
⟨*proof*⟩

"Less than" is antisymmetric, sort of.

lemma *less-antisym*: $\neg\ n < m \Longrightarrow n < Suc\ m \Longrightarrow m = n$
⟨*proof*⟩

lemma *nat-neq-iff*: $m \neq n \longleftrightarrow m < n \vee n < m$
for $m\ n$ :: *nat*
⟨*proof*⟩

16.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \Longrightarrow Suc\ m \neq n \Longrightarrow Suc\ m < n$
⟨*proof*⟩

lemma *lessE*:
assumes *major*: $i < k$
and *1*: $k = Suc\ i \Longrightarrow P$
and *2*: $\bigwedge j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$
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*: $\bigwedge j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$
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*: $\neg$ *Suc* $n \le n$
⟨*proof*⟩

lemma *le-Suc-eq*: $m \le$ *Suc* $n \longleftrightarrow m \le n \vee m =$ *Suc* n
⟨*proof*⟩

lemma *le-SucE*: $m \le$ *Suc* $n \Longrightarrow (m \le n \Longrightarrow R) \Longrightarrow (m =$ *Suc* $n \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *Suc-leI*: $m < n \Longrightarrow$ *Suc* $m \le n$
⟨*proof*⟩

Stronger version of *Suc-leD*.

lemma *Suc-le-lessD*: *Suc* $m \le n \Longrightarrow m < n$
⟨*proof*⟩

lemma *less-imp-le-nat*: $m < n \Longrightarrow m \le n$ **for** m n :: *nat*
⟨*proof*⟩

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \le n) = (m < n)$

lemmas *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \le n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \vee m = n \Longrightarrow m \le n$
for m n :: *nat*
⟨*proof*⟩

lemma *le-eq-less-or-eq*: $m \le n \longleftrightarrow m < n \vee m = n$
for m n :: *nat*
⟨*proof*⟩

Useful with *blast*.

lemma *eq-imp-le*: $m = n \Longrightarrow m \le n$
for m n :: *nat*
⟨*proof*⟩

lemma *le-refl*: $n \le n$
for n :: *nat*
⟨*proof*⟩

lemma *le-trans*: $i \le j \Longrightarrow j \le k \Longrightarrow i \le k$
for i j k :: *nat*
⟨*proof*⟩

lemma *le-antisym*: $m \le n \Longrightarrow n \le m \Longrightarrow m = n$
for m n :: *nat*
⟨*proof*⟩

lemma *nat-less-le*: $m < n \longleftrightarrow m \leq n \wedge m \neq n$
for $m\ n :: nat$
$\langle proof \rangle$

lemma *le-neq-implies-less*: $m \leq n \Longrightarrow m \neq n \Longrightarrow m < n$
for $m\ n :: nat$
$\langle proof \rangle$

lemma *nat-le-linear*: $m \leq n \mid n \leq m$
for $m\ n :: nat$
$\langle proof \rangle$

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = nat$]

lemma *le-less-Suc-eq*: $m \leq n \Longrightarrow n < Suc\ m \longleftrightarrow n = m$
$\langle proof \rangle$

lemma *not-less-less-Suc-eq*: $\neg\ n < m \Longrightarrow n < Suc\ m \longleftrightarrow n = m$
$\langle proof \rangle$

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

lemma *not0-implies-Suc*: $n \neq 0 \Longrightarrow \exists\, m.\ n = Suc\ m$
$\langle proof \rangle$

lemma *gr0-implies-Suc*: $n > 0 \Longrightarrow \exists\, m.\ n = Suc\ m$
$\langle proof \rangle$

lemma *gr-implies-not0*: $m < n \Longrightarrow n \neq 0$
for $m\ n :: nat$
$\langle proof \rangle$

lemma *neq0-conv*[*iff*]: $n \neq 0 \longleftrightarrow 0 < n$
for $n :: nat$
$\langle proof \rangle$

This theorem is useful with *blast*

lemma *gr0I*: $(n = 0 \Longrightarrow False) \Longrightarrow 0 < n$
for $n :: nat$
$\langle proof \rangle$

lemma *gr0-conv-Suc*: $0 < n \longleftrightarrow (\exists\, m.\ n = Suc\ m)$
$\langle proof \rangle$

lemma *not-gr0* [*iff*]: $\neg\ 0 < n \longleftrightarrow n = 0$
for $n :: nat$
$\langle proof \rangle$

lemma *Suc-le-D*: $Suc\ n \leq m' \Longrightarrow \exists\, m.\ m' = Suc\ m$

⟨*proof*⟩

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $m < Suc\ n \longleftrightarrow m = 0 \lor (\exists j.\ m = Suc\ j \land j < n)$
⟨*proof*⟩

lemma *All-less-Suc*: $(\forall i < Suc\ n.\ P\ i) = (P\ n \land (\forall i < n.\ P\ i))$
⟨*proof*⟩

lemma *All-less-Suc2*: $(\forall i < Suc\ n.\ P\ i) = (P\ 0 \land (\forall i < n.\ P(Suc\ i)))$
⟨*proof*⟩

lemma *Ex-less-Suc*: $(\exists i < Suc\ n.\ P\ i) = (P\ n \lor (\exists i < n.\ P\ i))$
⟨*proof*⟩

lemma *Ex-less-Suc2*: $(\exists i < Suc\ n.\ P\ i) = (P\ 0 \lor (\exists i < n.\ P(Suc\ i)))$
⟨*proof*⟩

16.4.5 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: $n > 0 \Longrightarrow Suc\ (n - Suc\ 0) = n$
⟨*proof*⟩

lemma *Suc-diff-1* [*simp*]: $0 < n \Longrightarrow Suc\ (n - 1) = n$
⟨*proof*⟩

lemma *nat-add-left-cancel-le* [*simp*]: $k + m \le k + n \longleftrightarrow m \le n$
for $k\ m\ n :: nat$
⟨*proof*⟩

lemma *nat-add-left-cancel-less* [*simp*]: $k + m < k + n \longleftrightarrow m < n$
for $k\ m\ n :: nat$
⟨*proof*⟩

lemma *add-gr-0* [*iff*]: $m + n > 0 \longleftrightarrow m > 0 \lor n > 0$
for $m\ n :: nat$
⟨*proof*⟩

strict, in 1st argument

lemma *add-less-mono1*: $i < j \Longrightarrow i + k < j + k$
for $i\ j\ k :: nat$
⟨*proof*⟩

strict, in both arguments

lemma *add-less-mono*: $i < j \Longrightarrow k < l \Longrightarrow i + k < j + l$
for $i\ j\ k\ l :: nat$
⟨*proof*⟩

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

lemma *less-imp-Suc-add*: $m < n \Longrightarrow \exists k.\ n = Suc\ (m + k)$
$\langle proof \rangle$

lemma *le-Suc-ex*: $k \le l \Longrightarrow (\exists n.\ l = k + n)$
for $k\ l$ :: *nat*
$\langle proof \rangle$

strict, in 1st argument; proof is by induction on $k > 0$

lemma *mult-less-mono2*:
fixes $i\ j$ :: *nat*
assumes $i < j$ **and** $0 < k$
shows $k * i < k * j$
$\langle proof \rangle$

Addition is the inverse of subtraction: if $n \le m$ then $n + (m - n) = m$.

lemma *add-diff-inverse-nat*: $\neg\ m < n \Longrightarrow n + (m - n) = m$
for $m\ n$ :: *nat*
$\langle proof \rangle$

lemma *nat-le-iff-add*: $m \le n \longleftrightarrow (\exists k.\ n = m + k)$
for $m\ n$ :: *nat*
$\langle proof \rangle$

The naturals form an ordered *semidom* and a *dioid*.

instance *nat* :: *linordered-semidom*
$\langle proof \rangle$

instance *nat* :: *dioid*
$\langle proof \rangle$

declare *le0*[*simp del*] — This is now $(0::?'a) \le ?x$
declare *le-0-eq*[*simp del*] — This is now $(?n \le (0::?'a)) = (?n = (0::?'a))$
declare *not-less0*[*simp del*] — This is now $\neg\ ?n < (0::?'a)$
declare *not-gr0*[*simp del*] — This is now $(\neg\ (0::?'a) < ?n) = (?n = (0::?'a))$

instance *nat* :: *ordered-cancel-comm-monoid-add* $\langle proof \rangle$
instance *nat* :: *ordered-cancel-comm-monoid-diff* $\langle proof \rangle$

16.4.6 *min* and *max*

lemma *mono-Suc*: *mono Suc*
$\langle proof \rangle$

lemma *min-0L* [*simp*]: $min\ 0\ n = 0$
for n :: *nat*
$\langle proof \rangle$

lemma *min-0R* [*simp*]: $min\ n\ 0 = 0$
for n :: *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* | *Suc m'* ⇒ *Suc*(*min n m'*))
⟨*proof*⟩

lemma *min-Suc2*: *min m* (*Suc n*) = (*case m of 0* ⇒ *0* | *Suc m'* ⇒ *Suc*(*min m' n*))
⟨*proof*⟩

lemma *max-0L* [*simp*]: *max 0 n* = *n*
for *n* :: *nat*
⟨*proof*⟩

lemma *max-0R* [*simp*]: *max n 0* = *n*
for *n* :: *nat*
⟨*proof*⟩

lemma *max-Suc-Suc* [*simp*]: *max* (*Suc m*) (*Suc n*) = *Suc* (*max m n*)
⟨*proof*⟩

lemma *max-Suc1*: *max* (*Suc n*) *m* = (*case m of 0* ⇒ *Suc n* | *Suc m'* ⇒ *Suc* (*max n m'*))
⟨*proof*⟩

lemma *max-Suc2*: *max m* (*Suc n*) = (*case m of 0* ⇒ *Suc n* | *Suc m'* ⇒ *Suc* (*max m' n*))
⟨*proof*⟩

lemma *nat-mult-min-left*: *min m n* ∗ *q* = *min* (*m* ∗ *q*) (*n* ∗ *q*)
for *m n q* :: *nat*
⟨*proof*⟩

lemma *nat-mult-min-right*: *m* ∗ *min n q* = *min* (*m* ∗ *n*) (*m* ∗ *q*)
for *m n q* :: *nat*
⟨*proof*⟩

lemma *nat-add-max-left*: *max m n* + *q* = *max* (*m* + *q*) (*n* + *q*)
for *m n q* :: *nat*
⟨*proof*⟩

lemma *nat-add-max-right*: *m* + *max n q* = *max* (*m* + *n*) (*m* + *q*)
for *m n q* :: *nat*
⟨*proof*⟩

lemma *nat-mult-max-left*: *max m n* ∗ *q* = *max* (*m* ∗ *q*) (*n* ∗ *q*)

for $m\ n\ q$ :: *nat*
⟨*proof*⟩

lemma *nat-mult-max-right*: $m * max\ n\ q = max\ (m * n)\ (m * q)$
for $m\ n\ q$ :: *nat*
⟨*proof*⟩

16.4.7 Additional theorems about $op \leq$

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder*
⟨*proof*⟩

lemma *Least-eq-0*[*simp*]: $P\ 0 \Longrightarrow Least\ P = 0$
for P :: $nat \Rightarrow bool$
⟨*proof*⟩

lemma *Least-Suc*: $P\ n \Longrightarrow \neg\ P\ 0 \Longrightarrow (LEAST\ n.\ P\ n) = Suc\ (LEAST\ m.\ P\ (Suc\ m))$
⟨*proof*⟩

lemma *Least-Suc2*: $P\ n \Longrightarrow Q\ m \Longrightarrow \neg\ P\ 0 \Longrightarrow \forall k.\ P\ (Suc\ k) = Q\ k \Longrightarrow Least\ P = Suc\ (Least\ Q)$
⟨*proof*⟩

lemma *ex-least-nat-le*: $\neg\ P\ 0 \Longrightarrow P\ n \Longrightarrow \exists k{\leq}n.\ (\forall i{<}k.\ \neg\ P\ i) \wedge P\ k$
for P :: $nat \Rightarrow bool$
⟨*proof*⟩

lemma *ex-least-nat-less*: $\neg\ P\ 0 \Longrightarrow P\ n \Longrightarrow \exists k{<}n.\ (\forall i{\leq}k.\ \neg\ P\ i) \wedge P\ (k + 1)$
for P :: $nat \Rightarrow bool$
⟨*proof*⟩

lemma *nat-less-induct*:
fixes P :: $nat \Rightarrow bool$
assumes $\bigwedge n.\ \forall m.\ m < n \longrightarrow P\ m \Longrightarrow P\ n$
shows $P\ n$
⟨*proof*⟩

lemma *measure-induct-rule* [*case-names less*]:
fixes f :: $'a \Rightarrow\ 'b$::*wellorder*
assumes *step*: $\bigwedge x.\ (\bigwedge y.\ f\ y < f\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows $P\ a$
⟨*proof*⟩

old style induction rules:

lemma *measure-induct*:
fixes f :: $'a \Rightarrow\ 'b$::*wellorder*

shows $(\bigwedge x.\ \forall y.\ f\ y < f\ x \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
⟨*proof*⟩

lemma *full-nat-induct*:
assumes *step*: $\bigwedge n.\ (\forall m.\ Suc\ m \leq n \longrightarrow P\ m) \Longrightarrow P\ n$
shows $P\ n$
⟨*proof*⟩

An induction rule for establishing binary relations

lemma *less-Suc-induct* [*consumes 1*]:
assumes *less*: $i < j$
and *step*: $\bigwedge i.\ P\ i\ (Suc\ i)$
and *trans*: $\bigwedge i\ j\ k.\ i < j \Longrightarrow j < k \Longrightarrow P\ i\ j \Longrightarrow P\ j\ k \Longrightarrow P\ i\ k$
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 natural numbers if

- case "0": given $n = 0$ prove $P\ n$
- case "smaller": given $n > 0$ and $\neg\ P\ n$ prove there exists a smaller natural number m such that $\neg\ P\ m$.

lemma *infinite-descent*: $(\bigwedge n.\ \neg\ P\ n \Longrightarrow \exists\, m<n.\ \neg\ P\ m) \Longrightarrow P\ n$ **for** $P :: nat \Rightarrow bool$
— compact version without explicit base case
⟨*proof*⟩

lemma *infinite-descent0* [*case-names 0 smaller*]:
fixes $P :: nat \Rightarrow bool$
assumes $P\ 0$
and $\bigwedge n.\ n > 0 \Longrightarrow \neg\ P\ n \Longrightarrow \exists\, m.\ m < n \wedge \neg\ P\ m$
shows $P\ n$
⟨*proof*⟩

Infinite descent using a mapping to *nat*: $P\ x$ is true for all $x \in D$ if there exists a $V \in D \Rightarrow nat$ and

- case "0": given $V\ x = 0$ prove $P\ x$
- "smaller": given $V\ x > 0$ and $\neg\ P\ x$ prove there exists a $y \in D$ such that $V\ y < V\ x$ and $\neg\ P\ y$.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
fixes $V :: 'a \Rightarrow nat$
assumes *1*: $\bigwedge x.\ V\ x = 0 \Longrightarrow P\ x$
and *2*: $\bigwedge x.\ V\ x > 0 \Longrightarrow \neg\ P\ x \Longrightarrow \exists\, y.\ V\ y < V\ x \wedge \neg\ P\ y$

shows $P\ x$
$\langle proof \rangle$

Again, without explicit base case:

lemma *infinite-descent-measure*:
fixes $V :: 'a \Rightarrow nat$
assumes $\bigwedge x.\ \neg\ P\ x \Longrightarrow \exists\, y.\ V\ y < V\ x \wedge \neg\ P\ y$
shows $P\ x$
$\langle proof \rangle$

A (clumsy) way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
fixes $f :: nat \Rightarrow nat$
and $i\ j :: nat$
assumes $\bigwedge i\ j{::}nat.\ i < j \Longrightarrow f\ i < f\ j$
and $i \leq j$
shows $f\ i \leq f\ j$
$\langle proof \rangle$

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j \Longrightarrow i + k \leq j + k$
for $i\ j\ k :: nat$
$\langle proof \rangle$

non-strict, in both arguments

lemma *add-le-mono*: $i \leq j \Longrightarrow k \leq l \Longrightarrow i + k \leq j + l$
for $i\ j\ k\ l :: nat$
$\langle proof \rangle$

lemma *le-add2*: $n \leq m + n$
for $m\ n :: nat$
$\langle proof \rangle$

lemma *le-add1*: $n \leq n + m$
for $m\ n :: nat$
$\langle proof \rangle$

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
$\langle proof \rangle$

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
$\langle proof \rangle$

lemma *less-iff-Suc-add*: $m < n \longleftrightarrow (\exists\, k.\ n = Suc\ (m + k))$
$\langle proof \rangle$

lemma *trans-le-add1*: $i \leq j \Longrightarrow i \leq j + m$
for $i\ j\ m :: nat$

⟨*proof*⟩

lemma *trans-le-add2*: $i \leq j \Longrightarrow i \leq m + j$
for $i\ j\ m :: nat$
⟨*proof*⟩

lemma *trans-less-add1*: $i < j \Longrightarrow i < j + m$
for $i\ j\ m :: nat$
⟨*proof*⟩

lemma *trans-less-add2*: $i < j \Longrightarrow i < m + j$
for $i\ j\ m :: nat$
⟨*proof*⟩

lemma *add-lessD1*: $i + j < k \Longrightarrow i < k$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *not-add-less1* [*iff*]: $\neg\ i + j < i$
for $i\ j :: nat$
⟨*proof*⟩

lemma *not-add-less2* [*iff*]: $\neg\ j + i < i$
for $i\ j :: nat$
⟨*proof*⟩

lemma *add-leD1*: $m + k \leq n \Longrightarrow m \leq n$
for $k\ m\ n :: nat$
⟨*proof*⟩

lemma *add-leD2*: $m + k \leq n \Longrightarrow k \leq n$
for $k\ m\ n :: nat$
⟨*proof*⟩

lemma *add-leE*: $m + k \leq n \Longrightarrow (m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
for $k\ m\ n :: nat$
⟨*proof*⟩

needs $\bigwedge k$ for *ac-simps* to work

lemma *less-add-eq-less*: $\bigwedge k.\ k < l \Longrightarrow m + l = k + n \Longrightarrow m < n$
for $l\ m\ n :: nat$
⟨*proof*⟩

16.4.8 More results about difference

lemma *Suc-diff-le*: $n \leq m \Longrightarrow Suc\ m - n = Suc\ (m - n)$
⟨*proof*⟩

lemma *diff-less-Suc*: $m - n < Suc\ m$

⟨*proof*⟩

lemma *diff-le-self* [*simp*]: $m - n \le m$
for $m\ n :: nat$
⟨*proof*⟩

lemma *less-imp-diff-less*: $j < k \Longrightarrow j - n < k$
for $j\ k\ n :: nat$
⟨*proof*⟩

lemma *diff-Suc-less* [*simp*]: $0 < n \Longrightarrow n - Suc\ i < n$
⟨*proof*⟩

lemma *diff-add-assoc*: $k \le j \Longrightarrow (i + j) - k = i + (j - k)$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *add-diff-assoc* [*simp*]: $k \le j \Longrightarrow i + (j - k) = i + j - k$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *diff-add-assoc2*: $k \le j \Longrightarrow (j + i) - k = (j - k) + i$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *add-diff-assoc2* [*simp*]: $k \le j \Longrightarrow j - k + i = j + i - k$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *le-imp-diff-is-add*: $i \le j \Longrightarrow (j - i = k) = (j = k + i)$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *diff-is-0-eq* [*simp*]: $m - n = 0 \longleftrightarrow m \le n$
for $m\ n :: nat$
⟨*proof*⟩

lemma *diff-is-0-eq'* [*simp*]: $m \le n \Longrightarrow m - n = 0$
for $m\ n :: nat$
⟨*proof*⟩

lemma *zero-less-diff* [*simp*]: $0 < n - m \longleftrightarrow m < n$
for $m\ n :: nat$
⟨*proof*⟩

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k::nat.\ 0 < k \wedge i + k = j$
⟨*proof*⟩

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*: $n - m + m = max\ n\ m$
for $m\ n :: nat$
⟨*proof*⟩

lemma *nat-diff-split*: $P\ (a - b) \longleftrightarrow (a < b \longrightarrow P\ 0) \wedge (\forall d.\ a = b + d \longrightarrow P\ d)$
for $a\ b :: nat$
— elimination of $-$ on *nat*
⟨*proof*⟩

lemma *nat-diff-split-asm*: $P\ (a - b) \longleftrightarrow \neg\ (a < b \wedge \neg\ P\ 0 \vee (\exists d.\ a = b + d \wedge \neg\ P\ d))$
for $a\ b :: nat$
— elimination of $-$ on *nat* in assumptions
⟨*proof*⟩

lemma *Suc-pred'*: $0 < n \Longrightarrow n = Suc(n - 1)$
⟨*proof*⟩

lemma *add-eq-if*: $m + n = (if\ m = 0\ then\ n\ else\ Suc\ ((m - 1) + n))$
⟨*proof*⟩

lemma *mult-eq-if*: $m * n = (if\ m = 0\ then\ 0\ else\ n + ((m - 1) * n))$
for $m\ n :: nat$
⟨*proof*⟩

lemma *Suc-diff-eq-diff-pred*: $0 < n \Longrightarrow Suc\ m - n = m - (n - 1)$
⟨*proof*⟩

lemma *diff-Suc-eq-diff-pred*: $m - Suc\ n = (m - 1) - n$
⟨*proof*⟩

lemma *Let-Suc* [*simp*]: $Let\ (Suc\ n)\ f \equiv f\ (Suc\ n)$
⟨*proof*⟩

16.4.9 Monotonicity of multiplication

lemma *mult-le-mono1*: $i \leq j \Longrightarrow i * k \leq j * k$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *mult-le-mono2*: $i \leq j \Longrightarrow k * i \leq k * j$
for $i\ j\ k :: nat$
⟨*proof*⟩

$\leq$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq j \Longrightarrow k \leq l \Longrightarrow i * k \leq j * l$
for $i\ j\ k\ l :: nat$

⟨*proof*⟩

lemma *mult-less-mono1*: $i < j \Longrightarrow 0 < k \Longrightarrow i * k < j * k$
for $i\ j\ k$ :: *nat*
⟨*proof*⟩

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $0 < m * n \longleftrightarrow 0 < m \wedge 0 < n$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *one-le-mult-iff* [*simp*]: $Suc\ 0 \leq m * n \longleftrightarrow Suc\ 0 \leq m \wedge Suc\ 0 \leq n$
⟨*proof*⟩

lemma *mult-less-cancel2* [*simp*]: $m * k < n * k \longleftrightarrow 0 < k \wedge m < n$
for $k\ m\ n$ :: *nat*
⟨*proof*⟩

lemma *mult-less-cancel1* [*simp*]: $k * m < k * n \longleftrightarrow 0 < k \wedge m < n$
for $k\ m\ n$ :: *nat*
⟨*proof*⟩

lemma *mult-le-cancel1* [*simp*]: $k * m \leq k * n \longleftrightarrow (0 < k \longrightarrow m \leq n)$
for $k\ m\ n$ :: *nat*
⟨*proof*⟩

lemma *mult-le-cancel2* [*simp*]: $m * k \leq n * k \longleftrightarrow (0 < k \longrightarrow m \leq n)$
for $k\ m\ n$ :: *nat*
⟨*proof*⟩

lemma *Suc-mult-less-cancel1*: $Suc\ k * m < Suc\ k * n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *Suc-mult-le-cancel1*: $Suc\ k * m \leq Suc\ k * n \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *le-square*: $m \leq m * m$
for m :: *nat*
⟨*proof*⟩

lemma *le-cube*: $m \leq m * (m * m)$
for m :: *nat*
⟨*proof*⟩

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $m = m * n \Longrightarrow n = 1 \vee m = 0$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *mono-times-nat*:
  fixes n :: *nat*
  assumes $n > 0$
  shows *mono* (*times* n)
⟨*proof*⟩

The lattice order on *nat*.

instantiation *nat* :: *distrib-lattice*
begin

definition (*inf* :: *nat* ⇒ *nat* ⇒ *nat*) = *min*

definition (*sup* :: *nat* ⇒ *nat* ⇒ *nat*) = *max*

instance
  ⟨*proof*⟩

end

16.5 Natural operation of natural numbers on functions

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

consts *compow* :: *nat* ⇒ *'a* ⇒ *'a*

abbreviation *compower* :: *'a* ⇒ *nat* ⇒ *'a* (**infixr** ^^ *80*)
  where f ^^ n ≡ *compow* n f

notation (*latex* **output**)
  compower ((-) [*1000*] *1000*)

f ^^ $n = f \circ \ldots \circ f$, 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* ∘ *funpow n f*

end

lemma *funpow-0* [*simp*]: (f ^^ *0*) $x = x$
  ⟨*proof*⟩

lemma *funpow-Suc-right*: f ^^ *Suc n* = f ^^ $n \circ f$

⟨*proof*⟩

lemmas *funpow-simps-right* = *funpow.simps*(*1*) *funpow-Suc-right*

For code generation.

definition *funpow* :: *nat* ⇒ (′*a* ⇒ ′*a*) ⇒ ′*a* ⇒ ′*a*
where *funpow-code-def* [*code-abbrev*]: *funpow* = *compow*

lemma [*code*]:
funpow (*Suc n*) *f* = *f* ∘ *funpow n f*
funpow 0 f = *id*
⟨*proof*⟩

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

lemma *funpow-add*: *f* ^^ (*m* + *n*) = *f* ^^ *m* ∘ *f* ^^ *n*
⟨*proof*⟩

lemma *funpow-mult*: (*f* ^^ *m*) ^^ *n* = *f* ^^ (*m* ∗ *n*)
for *f* :: ′*a* ⇒ ′*a*
⟨*proof*⟩

lemma *funpow-swap1*: *f* ((*f* ^^ *n*) *x*) = (*f* ^^ *n*) (*f x*)
⟨*proof*⟩

lemma *comp-funpow*: *comp f* ^^ *n* = *comp* (*f* ^^ *n*)
for *f* :: ′*a* ⇒ ′*a*
⟨*proof*⟩

lemma *Suc-funpow*[*simp*]: *Suc* ^^ *n* = (*op* + *n*)
⟨*proof*⟩

lemma *id-funpow*[*simp*]: *id* ^^ *n* = *id*
⟨*proof*⟩

lemma *funpow-mono*: *mono f* ⟹ *A* ≤ *B* ⟹ (*f* ^^ *n*) *A* ≤ (*f* ^^ *n*) *B*
for *f* :: ′*a* ⇒ (′*a*::*order*)
⟨*proof*⟩

lemma *funpow-mono2*:
assumes *mono f*
and *i* ≤ *j*
and *x* ≤ *y*
and *x* ≤ *f x*
shows (*f* ^^ *i*) *x* ≤ (*f* ^^ *j*) *y*
⟨*proof*⟩

16.6 Kleene iteration

lemma *Kleene-iter-lpfp*:
 fixes f :: $'a$::*order-bot* $\Rightarrow$ $'a$
 assumes *mono f*
  and $f\ p \leq p$
 shows $(f\ \hat{}\hat{}\ k)\ bot \leq p$
⟨*proof*⟩

lemma *lfp-Kleene-iter*:
 assumes *mono f*
  and $(f\ \hat{}\hat{}\ Suc\ k)\ bot = (f\ \hat{}\hat{}\ k)\ bot$
 shows $lfp\ f = (f\ \hat{}\hat{}\ k)\ bot$
⟨*proof*⟩

lemma *mono-pow*: *mono f* $\Longrightarrow$ *mono* $(f\ \hat{}\hat{}\ n)$
 for f :: $'a \Rightarrow 'a$::*complete-lattice*
 ⟨*proof*⟩

lemma *lfp-funpow*:
 assumes f: *mono f*
 shows $lfp\ (f\ \hat{}\hat{}\ Suc\ n) = lfp\ f$
⟨*proof*⟩

lemma *gfp-funpow*:
 assumes f: *mono f*
 shows $gfp\ (f\ \hat{}\hat{}\ Suc\ n) = gfp\ f$
⟨*proof*⟩

lemma *Kleene-iter-gpfp*:
 fixes f :: $'a$::*order-top* $\Rightarrow$ $'a$
 assumes *mono f*
  and $p \leq f\ p$
 shows $p \leq (f\ \hat{}\hat{}\ k)\ top$
⟨*proof*⟩

lemma *gfp-Kleene-iter*:
 assumes *mono f*
  and $(f\ \hat{}\hat{}\ Suc\ k)\ top = (f\ \hat{}\hat{}\ k)\ top$
 shows $gfp\ f = (f\ \hat{}\hat{}\ k)\ top$
  (**is** $?lhs = ?rhs$)
⟨*proof*⟩

16.7 Embedding of the naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

definition *of-nat* :: $nat \Rightarrow 'a$
 where $of\text{-}nat\ n = (plus\ 1\ \hat{}\hat{}\ n)\ 0$

lemma *of-nat-simps* [*simp*]:
 shows *of-nat-0*: *of-nat 0* = *0*
 and *of-nat-Suc*: *of-nat* (*Suc m*) = *1* + *of-nat m*
 ⟨*proof*⟩

lemma *of-nat-1* [*simp*]: *of-nat 1* = *1*
 ⟨*proof*⟩

lemma *of-nat-add* [*simp*]: *of-nat* (*m* + *n*) = *of-nat m* + *of-nat n*
 ⟨*proof*⟩

lemma *of-nat-mult* [*simp*]: *of-nat* (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
 ⟨*proof*⟩

lemma *mult-of-nat-commute*: *of-nat x* ∗ *y* = *y* ∗ *of-nat x*
 ⟨*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*]

context *ring-1*
begin

lemma *of-nat-diff*: *n* ≤ *m* ⟹ *of-nat* (*m* − *n*) = *of-nat m* − *of-nat n*
 ⟨*proof*⟩

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0* = *semiring-1* +
 assumes *inj-of-nat*: *inj of-nat*
begin

lemma *of-nat-eq-iff* [*simp*]: *of-nat m* = *of-nat n* ⟷ *m* = *n*
 ⟨*proof*⟩

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*]: *0* = *of-nat n* ⟷ *0* = *n*

⟨*proof*⟩

lemma *of-nat-eq-0-iff* [*simp*]: *of-nat* $m = 0 \longleftrightarrow m = 0$
⟨*proof*⟩

lemma *of-nat-1-eq-iff* [*simp*]: $1 =$ *of-nat* $n \longleftrightarrow n=1$
⟨*proof*⟩

lemma *of-nat-eq-1-iff* [*simp*]: *of-nat* $n = 1 \longleftrightarrow n=1$
⟨*proof*⟩

lemma *of-nat-neq-0* [*simp*]: *of-nat* (*Suc* n) $\neq 0$
⟨*proof*⟩

lemma *of-nat-0-neq* [*simp*]: $0 \neq$ *of-nat* (*Suc* n)
⟨*proof*⟩

end

class *ring-char-0* = *ring-1* + *semiring-char-0*

context *linordered-semidom*
begin

lemma *of-nat-0-le-iff* [*simp*]: $0 \leq$ *of-nat* n
⟨*proof*⟩

lemma *of-nat-less-0-iff* [*simp*]: $\neg$ *of-nat* $m < 0$
⟨*proof*⟩

lemma *of-nat-less-iff* [*simp*]: *of-nat* $m <$ *of-nat* $n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *of-nat-le-iff* [*simp*]: *of-nat* $m \leq$ *of-nat* $n \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *less-imp-of-nat-less*: $m < n \Longrightarrow$ *of-nat* $m <$ *of-nat* n
⟨*proof*⟩

lemma *of-nat-less-imp-less*: *of-nat* $m <$ *of-nat* $n \Longrightarrow m < n$
⟨*proof*⟩

Every *linordered-semidom* has characteristic zero.

subclass *semiring-char-0*
⟨*proof*⟩

Special cases where either operand is zero

lemma *of-nat-le-0-iff* [*simp*]: *of-nat* $m \leq 0 \longleftrightarrow m = 0$
⟨*proof*⟩

lemma *of-nat-0-less-iff* [*simp*]: $0 < \textit{of-nat } n \longleftrightarrow 0 < n$
  ⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *abs-of-nat* [*simp*]: $|\textit{of-nat } n| = \textit{of-nat } n$
  ⟨*proof*⟩

end

lemma *of-nat-id* [*simp*]: $\textit{of-nat } n = n$
  ⟨*proof*⟩

lemma *of-nat-eq-id* [*simp*]: $\textit{of-nat} = \textit{id}$
  ⟨*proof*⟩

16.8 The set of natural numbers

context *semiring-1*
begin

definition *Nats* :: *'a set* ($\mathbb{N}$)
  where $\mathbb{N} = \textit{range of-nat}$

lemma *of-nat-in-Nats* [*simp*]: $\textit{of-nat } n \in \mathbb{N}$
  ⟨*proof*⟩

lemma *Nats-0* [*simp*]: $0 \in \mathbb{N}$
  ⟨*proof*⟩

lemma *Nats-1* [*simp*]: $1 \in \mathbb{N}$
  ⟨*proof*⟩

lemma *Nats-add* [*simp*]: $a \in \mathbb{N} \Longrightarrow b \in \mathbb{N} \Longrightarrow a + b \in \mathbb{N}$
  ⟨*proof*⟩

lemma *Nats-mult* [*simp*]: $a \in \mathbb{N} \Longrightarrow b \in \mathbb{N} \Longrightarrow a * b \in \mathbb{N}$
  ⟨*proof*⟩

lemma *Nats-cases* [*cases set*: *Nats*]:
  assumes $x \in \mathbb{N}$
  obtains (*of-nat*) n **where** $x = \textit{of-nat } n$
  ⟨*proof*⟩

lemma *Nats-induct* [*case-names of-nat*, *induct set*: *Nats*]: $x \in \mathbb{N} \Longrightarrow (\bigwedge n.\ P$

$(of\text{-}nat\ n)) \Longrightarrow P\ x$
⟨*proof*⟩

end

16.9 Further arithmetic facts concerning the natural numbers

lemma *subst-equals*:
assumes $t = s$ **and** $u = t$
shows $u = s$
⟨*proof*⟩

⟨*ML*⟩

context *order*
begin

lemma *lift-Suc-mono-le*:
assumes *mono*: $\bigwedge n.\ f\ n \leq f\ (Suc\ n)$
and $n \leq n'$
shows $f\ n \leq f\ n'$
⟨*proof*⟩

lemma *lift-Suc-antimono-le*:
assumes *mono*: $\bigwedge n.\ f\ n \geq f\ (Suc\ n)$
and $n \leq n'$
shows $f\ n \geq f\ n'$
⟨*proof*⟩

lemma *lift-Suc-mono-less*:
assumes *mono*: $\bigwedge n.\ f\ n < f\ (Suc\ n)$
and $n < n'$
shows $f\ n < f\ n'$
⟨*proof*⟩

lemma *lift-Suc-mono-less-iff*: $(\bigwedge n.\ f\ n < f\ (Suc\ n)) \Longrightarrow f\ n < f\ m \longleftrightarrow n < m$
⟨*proof*⟩

end

lemma *mono-iff-le-Suc*: $mono\ f \longleftrightarrow (\forall\, n.\ f\ n \leq f\ (Suc\ n))$
⟨*proof*⟩

lemma *antimono-iff-le-Suc*: $antimono\ f \longleftrightarrow (\forall\, n.\ f\ (Suc\ n) \leq f\ n)$
⟨*proof*⟩

lemma *mono-nat-linear-lb*:
fixes $f :: nat \Rightarrow nat$

assumes $\bigwedge m\ n.\ m < n \Longrightarrow f\ m < f\ n$
shows $f\ m + k \leq f\ (m + k)$
⟨*proof*⟩

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*:
fixes $a\ b\ c :: nat$
assumes $a < b$ **and** $c \leq a$
shows $a - c < b - c$
⟨*proof*⟩

lemma *less-diff-conv*: $i < j - k \longleftrightarrow i + k < j$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *less-diff-conv2*: $k \leq j \Longrightarrow j - k < i \longleftrightarrow j < i + k$
for $j\ k\ i :: nat$
⟨*proof*⟩

lemma *le-diff-conv*: $j - k \leq i \longleftrightarrow j \leq i + k$
for $j\ k\ i :: nat$
⟨*proof*⟩

lemma *diff-diff-cancel* [*simp*]: $i \leq n \Longrightarrow n - (n - i) = i$
for $i\ n :: nat$
⟨*proof*⟩

lemma *diff-less* [*simp*]: $0 < n \Longrightarrow 0 < m \Longrightarrow m - n < m$
for $i\ n :: nat$
⟨*proof*⟩

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k - (n - k) = m - n$
for $m\ n\ k :: nat$
⟨*proof*⟩

hide-fact (**open**) *diff-diff-eq*

lemma *eq-diff-iff*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k = n - k \longleftrightarrow m = n$
for $m\ n\ k :: nat$
⟨*proof*⟩

lemma *less-diff-iff*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k < n - k \longleftrightarrow m < n$
for $m\ n\ k :: nat$
⟨*proof*⟩

lemma *le-diff-iff*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k \leq n - k \longleftrightarrow m \leq n$
for $m\ n\ k :: nat$
⟨*proof*⟩

lemma *le-diff-iff'*: $a \leq c \Longrightarrow b \leq c \Longrightarrow c - a \leq c - b \longleftrightarrow b \leq a$
for $a\ b\ c :: nat$
⟨*proof*⟩

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq n \Longrightarrow m - l \leq n - l$
for $m\ n\ l :: nat$
⟨*proof*⟩

lemma *diff-le-mono2*: $m \leq n \Longrightarrow l - n \leq l - m$
for $m\ n\ l :: nat$
⟨*proof*⟩

lemma *diff-less-mono2*: $m < n \Longrightarrow m < l \Longrightarrow l - n < l - m$
for $m\ n\ l :: nat$
⟨*proof*⟩

lemma *diffs0-imp-equal*: $m - n = 0 \Longrightarrow n - m = 0 \Longrightarrow m = n$
for $m\ n :: nat$
⟨*proof*⟩

lemma *min-diff*: $min\ (m - i)\ (n - i) = min\ m\ n - i$
for $m\ n\ i :: nat$
⟨*proof*⟩

lemma *inj-on-diff-nat*:
fixes $k :: nat$
assumes $\forall\, n \in N.\ k \leq n$
shows $inj\text{-}on\ (\lambda n.\ n - k)\ N$
⟨*proof*⟩

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j \Longrightarrow i - (j - k) = i + k - j$
for $i\ j\ k :: nat$
⟨*proof*⟩

lemma *diff-Suc-diff-eq1* [*simp*]:
assumes $k \leq j$
shows $i - Suc\ (j - k) = i + k - Suc\ j$
⟨*proof*⟩

lemma *diff-Suc-diff-eq2* [*simp*]:
assumes $k \leq j$
shows $Suc\ (j - k) - i = Suc\ j - (k + i)$
⟨*proof*⟩

lemma *Suc-diff-Suc*:
assumes $n < m$

shows $Suc\ (m - Suc\ n) = m - n$
⟨*proof*⟩

lemma *one-less-mult*: $Suc\ 0 < n \Longrightarrow Suc\ 0 < m \Longrightarrow Suc\ 0 < m * n$
⟨*proof*⟩

lemma *n-less-m-mult-n*: $0 < n \Longrightarrow Suc\ 0 < m \Longrightarrow n < m * n$
⟨*proof*⟩

lemma *n-less-n-mult-m*: $0 < n \Longrightarrow Suc\ 0 < m \Longrightarrow n < n * m$
⟨*proof*⟩

Specialized induction principles that work "backwards":

lemma *inc-induct* [*consumes 1*, *case-names base step*]:
assumes *less*: $i \leq j$
and *base*: $P\ j$
and *step*: $\bigwedge n.\ i \leq n \Longrightarrow n < j \Longrightarrow P\ (Suc\ n) \Longrightarrow P\ n$
shows $P\ i$
⟨*proof*⟩

lemma *strict-inc-induct* [*consumes 1*, *case-names base step*]:
assumes *less*: $i < j$
and *base*: $\bigwedge i.\ j = Suc\ i \Longrightarrow P\ i$
and *step*: $\bigwedge i.\ i < j \Longrightarrow P\ (Suc\ i) \Longrightarrow P\ i$
shows $P\ i$
⟨*proof*⟩

lemma *zero-induct-lemma*: $P\ k \Longrightarrow (\bigwedge n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ (k - i)$
⟨*proof*⟩

lemma *zero-induct*: $P\ k \Longrightarrow (\bigwedge n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ 0$
⟨*proof*⟩

Further induction rule similar to $[\![?i \leq ?j;\ ?P\ ?j;\ \bigwedge n.\ [\![?i \leq n;\ n < ?j;\ ?P\ (Suc\ n)]\!] \Longrightarrow ?P\ n]\!] \Longrightarrow ?P\ ?i$.

lemma *dec-induct* [*consumes 1*, *case-names base step*]:
$i \leq j \Longrightarrow P\ i \Longrightarrow (\bigwedge n.\ i \leq n \Longrightarrow n < j \Longrightarrow P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ j$
⟨*proof*⟩

lemma *transitive-stepwise-le*:
assumes $m \leq n$ $\bigwedge x.\ R\ x\ x$ $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$ **and** $\bigwedge n.\ R\ n\ (Suc\ n)$
shows $R\ m\ n$
⟨*proof*⟩

16.9.1 Greatest operator

lemma *ex-has-greatest-nat*:
$P\ (k{::}nat) \Longrightarrow \forall y.\ P\ y \longrightarrow y \leq b \Longrightarrow \exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow y \leq x)$

⟨*proof*⟩

lemma *GreatestI-nat*:
$[\![P(k{::}nat); \forall y.\ P\ y \longrightarrow y \leq b\ ]\!] \Longrightarrow P\ (Greatest\ P)$
⟨*proof*⟩

lemma *Greatest-le-nat*:
$[\![P(k{::}nat);\ \forall y.\ P\ y \longrightarrow y \leq b\ ]\!] \Longrightarrow k \leq (Greatest\ P)$
⟨*proof*⟩

lemma *GreatestI-ex-nat*:
$[\![\exists k{::}nat.\ P\ k;\ \forall y.\ P\ y \longrightarrow y \leq b\ ]\!] \Longrightarrow P\ (Greatest\ P)$
⟨*proof*⟩

16.10 Monotonicity of *funpow*

lemma *funpow-increasing*: $m \leq n \Longrightarrow mono\ f \Longrightarrow (f\ \hat{}\hat{}\ n)\ \top \leq (f\ \hat{}\hat{}\ m)\ \top$
for f :: $'a{::}\{lattice,order\text{-}top\} \Rightarrow 'a$
⟨*proof*⟩

lemma *funpow-decreasing*: $m \leq n \Longrightarrow mono\ f \Longrightarrow (f\ \hat{}\hat{}\ m)\ \bot \leq (f\ \hat{}\hat{}\ n)\ \bot$
for f :: $'a{::}\{lattice,order\text{-}bot\} \Rightarrow 'a$
⟨*proof*⟩

lemma *mono-funpow*: $mono\ Q \Longrightarrow mono\ (\lambda i.\ (Q\ \hat{}\hat{}\ i)\ \bot)$
for Q :: $'a{::}\{lattice,order\text{-}bot\} \Rightarrow 'a$
⟨*proof*⟩

lemma *antimono-funpow*: $mono\ Q \Longrightarrow antimono\ (\lambda i.\ (Q\ \hat{}\hat{}\ i)\ \top)$
for Q :: $'a{::}\{lattice,order\text{-}top\} \Rightarrow 'a$
⟨*proof*⟩

16.11 The divides relation on *nat*

lemma *dvd-1-left* [*iff*]: *Suc 0 dvd k*
⟨*proof*⟩

lemma *dvd-1-iff-1* [*simp*]: $m\ dvd\ Suc\ 0 \longleftrightarrow m = Suc\ 0$
⟨*proof*⟩

lemma *nat-dvd-1-iff-1* [*simp*]: $m\ dvd\ 1 \longleftrightarrow m = 1$
for m :: nat
⟨*proof*⟩

lemma *dvd-antisym*: $m\ dvd\ n \Longrightarrow n\ dvd\ m \Longrightarrow m = n$
for $m\ n$ :: nat
⟨*proof*⟩

lemma *dvd-diff-nat* [*simp*]: $k\ dvd\ m \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ (m - n)$
for $k\ m\ n$ :: nat

⟨*proof*⟩

lemma *dvd-diffD*: $k\ dvd\ m - n \Longrightarrow k\ dvd\ n \Longrightarrow n \leq m \Longrightarrow k\ dvd\ m$
for $k\ m\ n$ :: *nat*
⟨*proof*⟩

lemma *dvd-diffD1*: $k\ dvd\ m - n \Longrightarrow k\ dvd\ m \Longrightarrow n \leq m \Longrightarrow k\ dvd\ n$
for $k\ m\ n$ :: *nat*
⟨*proof*⟩

lemma *dvd-mult-cancel*:
fixes $m\ n\ k$ :: *nat*
assumes $k * m\ dvd\ k * n$ **and** $0 < k$
shows $m\ dvd\ n$
⟨*proof*⟩

lemma *dvd-mult-cancel1*: $0 < m \Longrightarrow m * n\ dvd\ m \longleftrightarrow n = 1$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *dvd-mult-cancel2*: $0 < m \Longrightarrow n * m\ dvd\ m \longleftrightarrow n = 1$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *dvd-imp-le*: $k\ dvd\ n \Longrightarrow 0 < n \Longrightarrow k \leq n$
for $k\ n$ :: *nat*
⟨*proof*⟩

lemma *nat-dvd-not-less*: $0 < m \Longrightarrow m < n \Longrightarrow \neg\ n\ dvd\ m$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *less-eq-dvd-minus*:
fixes $m\ n$ :: *nat*
assumes $m \leq n$
shows $m\ dvd\ n \longleftrightarrow m\ dvd\ n - m$
⟨*proof*⟩

lemma *dvd-minus-self*: $m\ dvd\ n - m \longleftrightarrow n < m \lor m\ dvd\ n$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *dvd-minus-add*:
fixes $m\ n\ q\ r$ :: *nat*
assumes $q \leq n\ q \leq r * m$
shows $m\ dvd\ n - q \longleftrightarrow m\ dvd\ n + (r * m - q)$
⟨*proof*⟩

16.12 Aliasses

lemma *nat-mult-1*: $1 * n = n$
  for $n :: nat$
  ⟨*proof*⟩

lemma *nat-mult-1-right*: $n * 1 = n$
  for $n :: nat$
  ⟨*proof*⟩

lemma *nat-add-left-cancel*: $k + m = k + n \longleftrightarrow m = n$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemma *nat-add-right-cancel*: $m + k = n + k \longleftrightarrow m = n$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemma *diff-mult-distrib*: $(m - n) * k = (m * k) - (n * k)$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemma *diff-mult-distrib2*: $k * (m - n) = (k * m) - (k * n)$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemma *le-add-diff*: $k \le n \Longrightarrow m \le n + m - k$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemma *le-diff-conv2*: $k \le j \Longrightarrow (i \le j - k) = (i + k \le j)$
  for $i\ j\ k :: nat$
  ⟨*proof*⟩

lemma *diff-self-eq-0* [*simp*]: $m - m = 0$
  for $m :: nat$
  ⟨*proof*⟩

lemma *diff-diff-left* [*simp*]: $i - j - k = i - (j + k)$
  for $i\ j\ k :: nat$
  ⟨*proof*⟩

lemma *diff-commute*: $i - j - k = i - k - j$
  for $i\ j\ k :: nat$
  ⟨*proof*⟩

lemma *diff-add-inverse*: $(n + m) - n = m$
  for $m\ n :: nat$
  ⟨*proof*⟩

lemma *diff-add-inverse2*: $(m + n) - n = m$
  for $m\ n :: nat$
  ⟨*proof*⟩

lemma *diff-cancel*: $(k + m) - (k + n) = m - n$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemma *diff-cancel2*: $(m + k) - (n + k) = m - n$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemma *diff-add-0*: $n - (n + m) = 0$
  for $m\ n :: nat$
  ⟨*proof*⟩

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + (k * n)$
  for $k\ m\ n :: nat$
  ⟨*proof*⟩

lemmas *nat-distrib* =
  add-mult-distrib distrib-left diff-mult-distrib diff-mult-distrib2

16.13 Size of a datatype value

class *size* =
  fixes *size* :: $'a \Rightarrow nat$ — see further theory *Wellfounded*

instantiation *nat* :: *size*
begin

definition *size-nat* **where** [*simp*, *code*]: *size* (*n*::*nat*) = *n*

instance ⟨*proof*⟩

end

16.14 Code module namespace

code-identifier
  code-module *Nat* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

hide-const (**open**) *of-nat-aux*

end

17 Fields

theory *Fields*

imports *Nat*
begin

17.1 Division rings

A division ring is like a field, but without the commutativity requirement.

class *inverse* = *divide* +
  fixes *inverse* :: $'a \Rightarrow 'a$
begin

abbreviation *inverse-divide* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $'/$ *70*)
where
  inverse-divide $\equiv$ *divide*

end

Setup for linear arithmetic prover

⟨*ML*⟩

lemmas [*arith-split*] = *nat-diff-split split-min split-max*

Lemmas *divide-simps* move division to the outside and eliminates them on (in)equalities.

named-theorems *divide-simps rewrite rules to eliminate divisions*

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*: $a\ /\ b = a * inverse\ b$
  assumes *inverse-zero* [*simp*]: $inverse\ 0 = 0$
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** $a \neq 0$ **and** $b \neq 0$
shows $a = b$
⟨*proof*⟩

lemma *inverse-1* [*simp*]: $inverse\ 1 = 1$
⟨*proof*⟩

lemma *nonzero-inverse-mult-distrib*:
assumes $a \neq 0$ **and** $b \neq 0$
shows $inverse\ (a * b) = inverse\ b * inverse\ a$
⟨*proof*⟩

lemma *division-ring-inverse-add*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a + inverse\ b = inverse\ a * (a + b) * inverse\ b$
⟨*proof*⟩

lemma *division-ring-inverse-diff*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a - inverse\ b = inverse\ a * (b - a) * inverse\ b$
⟨*proof*⟩

lemma *right-inverse-eq*: $b \neq 0 \Longrightarrow a\ /\ b = 1 \longleftrightarrow a = b$
⟨*proof*⟩

lemma *nonzero-inverse-eq-divide*: $a \neq 0 \Longrightarrow inverse\ a = 1\ /\ a$
⟨*proof*⟩

lemma *divide-self* [*simp*]: $a \neq 0 \Longrightarrow a\ /\ a = 1$
⟨*proof*⟩

lemma *inverse-eq-divide* [*field-simps*, *divide-simps*]: $inverse\ a = 1\ /\ a$
⟨*proof*⟩

lemma *add-divide-distrib*: $(a+b)\ /\ c = a/c + b/c$
⟨*proof*⟩

lemma *times-divide-eq-right* [*simp*]: $a * (b\ /\ c) = (a * b)\ /\ c$
⟨*proof*⟩

lemma *minus-divide-left*: $-\ (a\ /\ b) = (-a)\ /\ b$

⟨proof⟩

lemma *nonzero-minus-divide-right*: $b \neq 0$ ==> $- (a / b) = a / (- b)$
⟨proof⟩

lemma *nonzero-minus-divide-divide*: $b \neq 0$ ==> $(-a) / (-b) = a / b$
⟨proof⟩

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

lemma *diff-divide-distrib*: $(a - b) / c = a / c - b / c$
⟨proof⟩

lemma *nonzero-eq-divide-eq* [*field-simps*]: $c \neq 0 \Longrightarrow a = b / c \longleftrightarrow a * c = b$
⟨proof⟩

lemma *nonzero-divide-eq-eq* [*field-simps*]: $c \neq 0 \Longrightarrow b / c = a \longleftrightarrow b = a * c$
⟨proof⟩

lemma *nonzero-neg-divide-eq-eq* [*field-simps*]: $b \neq 0 \Longrightarrow - (a / b) = c \longleftrightarrow - a = c * b$
⟨proof⟩

lemma *nonzero-neg-divide-eq-eq2* [*field-simps*]: $b \neq 0 \Longrightarrow c = - (a / b) \longleftrightarrow c * b = - a$
⟨proof⟩

lemma *divide-eq-imp*: $c \neq 0 \Longrightarrow b = a * c \Longrightarrow b / c = a$
⟨proof⟩

lemma *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b / c$
⟨proof⟩

lemma *add-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x + y / z = (x * z + y) / z$
⟨proof⟩

lemma *divide-add-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x / z + y = (x + y * z) / z$
⟨proof⟩

lemma *diff-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x - y / z = (x * z - y) / z$
⟨proof⟩

lemma *minus-divide-add-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow - (x / z) + y = (- x + y * z) / z$
⟨proof⟩

lemma *divide-diff-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x / z - y = (x - y * z) / z$
⟨*proof*⟩

lemma *minus-divide-diff-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow - (x / z) - y = (- x - y * z) / z$
⟨*proof*⟩

lemma *division-ring-divide-zero* [*simp*]:
$a / 0 = 0$
⟨*proof*⟩

lemma *divide-self-if* [*simp*]:
$a / a = (\mathit{if}\ a = 0\ \mathit{then}\ 0\ \mathit{else}\ 1)$
⟨*proof*⟩

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
$\mathit{inverse}\ a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *inverse-minus-eq* [*simp*]:
$\mathit{inverse}\ (- a) = - \mathit{inverse}\ a$
⟨*proof*⟩

lemma *inverse-inverse-eq* [*simp*]:
$\mathit{inverse}\ (\mathit{inverse}\ a) = a$
⟨*proof*⟩

lemma *inverse-eq-imp-eq*:
$\mathit{inverse}\ a = \mathit{inverse}\ b \Longrightarrow a = b$
⟨*proof*⟩

lemma *inverse-eq-iff-eq* [*simp*]:
$\mathit{inverse}\ a = \mathit{inverse}\ b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *add-divide-eq-if-simps* [*divide-simps*]:
$a + b / z = (\mathit{if}\ z = 0\ \mathit{then}\ a\ \mathit{else}\ (a * z + b) / z)$
$a / z + b = (\mathit{if}\ z = 0\ \mathit{then}\ b\ \mathit{else}\ (a + b * z) / z)$
$- (a / z) + b = (\mathit{if}\ z = 0\ \mathit{then}\ b\ \mathit{else}\ (-a + b * z) / z)$
$a - b / z = (\mathit{if}\ z = 0\ \mathit{then}\ a\ \mathit{else}\ (a * z - b) / z)$
$a / z - b = (\mathit{if}\ z = 0\ \mathit{then}\ -b\ \mathit{else}\ (a - b * z) / z)$
$- (a / z) - b = (\mathit{if}\ z = 0\ \mathit{then}\ -b\ \mathit{else}\ (- a - b * z) / z)$
⟨*proof*⟩

lemma [*divide-simps*]:
shows *divide-eq-eq*: $b / c = a \longleftrightarrow (\mathit{if}\ c \neq 0\ \mathit{then}\ b = a * c\ \mathit{else}\ a = 0)$
and *eq-divide-eq*: $a = b / c \longleftrightarrow (\mathit{if}\ c \neq 0\ \mathit{then}\ a * c = b\ \mathit{else}\ a = 0)$

and *minus-divide-eq-eq*: $- (b / c) = a \longleftrightarrow (\textit{if } c \neq 0 \textit{ then } - b = a * c \textit{ else } a = 0)$
and *eq-minus-divide-eq*: $a = - (b / c) \longleftrightarrow (\textit{if } c \neq 0 \textit{ then } a * c = - b \textit{ else } a = 0)$
⟨*proof*⟩

end

17.2 Fields

class *field* = *comm-ring-1* + *inverse* +
assumes *field-inverse*: $a \neq 0 \Longrightarrow \textit{inverse } a * a = 1$
assumes *field-divide-inverse*: $a / b = a * \textit{inverse } b$
assumes *field-inverse-zero*: $\textit{inverse } 0 = 0$
begin

subclass *division-ring*
⟨*proof*⟩

subclass *idom-divide*
⟨*proof*⟩

There is no slick version using division by zero.

lemma *inverse-add*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow \textit{inverse } a + \textit{inverse } b = (a + b) * \textit{inverse } a * \textit{inverse } b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*]:
assumes [*simp*]: $c \neq 0$
shows $(c * a) / (c * b) = a / b$
⟨*proof*⟩

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*]:
$c \neq 0 \Longrightarrow (a * c) / (b * c) = a / b$
⟨*proof*⟩

lemma *times-divide-eq-left* [*simp*]: $(b / c) * a = (b * a) / c$
⟨*proof*⟩

lemma *divide-inverse-commute*: $a / b = \textit{inverse } b * a$
⟨*proof*⟩

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-divide-mult-cancel-right* [*simp*]:

$b \neq 0 \Longrightarrow b \ / \ (a * b) = 1 \ / \ a$
⟨proof⟩

lemma *nonzero-divide-mult-cancel-left* [*simp*]:
$a \neq 0 \Longrightarrow a \ / \ (a * b) = 1 \ / \ b$
⟨proof⟩

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*]:
$c \neq 0 \Longrightarrow (c * a) \ / \ (b * c) = a \ / \ b$
⟨proof⟩

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*]:
$c \neq 0 \Longrightarrow (a * c) \ / \ (c * b) = a \ / \ b$
⟨proof⟩

lemma *diff-frac-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \ / \ y - w \ / \ z = (x * z - w * y) \ / \ (y * z)$
⟨proof⟩

lemma *frac-eq-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow (x \ / \ y = w \ / \ z) = (x * z = w * y)$
⟨proof⟩

lemma *divide-minus1* [*simp*]: $x \ / - 1 = - x$
⟨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*:
$c \neq 0 \Longrightarrow (c * a) \ / \ (c * b) = a \ / \ b$
⟨proof⟩

lemma *mult-divide-mult-cancel-right*:
$c \neq 0 \Longrightarrow (a * c) \ / \ (b * c) = a \ / \ b$
⟨proof⟩

lemma *divide-divide-eq-right* [*simp*]:

$a / (b / c) = (a * c) / b$
⟨proof⟩

lemma *divide-divide-eq-left* [*simp*]:
$(a / b) / c = a / (b * c)$
⟨proof⟩

lemma *divide-divide-times-eq*:
$(x / y) / (z / w) = (x * w) / (y * z)$
⟨proof⟩

Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if* [*simp*]:
shows $(c * a) / (c * b) = (\mathit{if}\ c = 0\ \mathit{then}\ 0\ \mathit{else}\ a / b)$
⟨proof⟩

Division and Unary Minus

lemma *minus-divide-right*:
$- (a / b) = a / - b$
⟨proof⟩

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

lemma *minus-divide-divide*:
$(- a) / (- b) = a / b$
⟨proof⟩

lemma *inverse-eq-1-iff* [*simp*]:
$\mathit{inverse}\ x = 1 \longleftrightarrow x = 1$
⟨proof⟩

lemma *divide-eq-0-iff* [*simp*]:
$a / b = 0 \longleftrightarrow a = 0 \lor b = 0$
⟨proof⟩

lemma *divide-cancel-right* [*simp*]:
$a / c = b / c \longleftrightarrow c = 0 \lor a = b$
⟨proof⟩

lemma *divide-cancel-left* [*simp*]:
$c / a = c / b \longleftrightarrow c = 0 \lor a = b$
⟨proof⟩

lemma *divide-eq-1-iff* [*simp*]:
$a / b = 1 \longleftrightarrow b \neq 0 \land a = b$
⟨proof⟩

lemma *one-eq-divide-iff* [*simp*]:
$1 = a / b \longleftrightarrow b \neq 0 \wedge a = b$
⟨*proof*⟩

lemma *divide-eq-minus-1-iff*:
$(a / b = - 1) \longleftrightarrow b \neq 0 \wedge a = - b$
⟨*proof*⟩

lemma *times-divide-times-eq*:
$(x / y) * (z / w) = (x * z) / (y * w)$
⟨*proof*⟩

lemma *add-frac-num*:
$y \neq 0 \Longrightarrow x / y + z = (x + z * y) / y$
⟨*proof*⟩

lemma *add-num-frac*:
$y \neq 0 \Longrightarrow z + x / y = (x + z * y) / y$
⟨*proof*⟩

lemma *dvd-field-iff*:
$a \; dvd \; b \longleftrightarrow (a = 0 \longrightarrow b = 0)$
⟨*proof*⟩

end

class *field-char-0* = *field* + *ring-char-0*

17.3 Ordered fields

class *field-abs-sgn* = *field* + *idom-abs-sgn*
begin

lemma *sgn-inverse* [*simp*]:
$sgn \; (inverse \; a) = inverse \; (sgn \; a)$
⟨*proof*⟩

lemma *abs-inverse* [*simp*]:
$|inverse \; a| = inverse \; |a|$
⟨*proof*⟩

lemma *sgn-divide* [*simp*]:
$sgn \; (a / b) = sgn \; a / sgn \; b$
⟨*proof*⟩

lemma *abs-divide* [*simp*]:
$|a / b| = |a| / |b|$
⟨*proof*⟩

end

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*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a < inverse\ b \longleftrightarrow b < a$

⟨*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*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$
⟨*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*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow inverse\ a < inverse\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *le-imp-inverse-le-neg*:
$a \leq b \Longrightarrow b < 0 ==> inverse\ b \leq inverse\ a$
⟨*proof*⟩

lemma *inverse-le-iff-le-neg* [*simp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$
⟨*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*]:
assumes $0 < c$
shows $a \leq b\ /\ c \longleftrightarrow a * c \leq b$
⟨*proof*⟩

lemma *pos-less-divide-eq* [*field-simps*]:
 assumes $0 < c$
 shows $a < b / c \longleftrightarrow a * c < b$
⟨*proof*⟩

lemma *neg-less-divide-eq* [*field-simps*]:
 assumes $c < 0$
 shows $a < b / c \longleftrightarrow b < a * c$
⟨*proof*⟩

lemma *neg-le-divide-eq* [*field-simps*]:
 assumes $c < 0$
 shows $a \leq b / c \longleftrightarrow b \leq a * c$
⟨*proof*⟩

lemma *pos-divide-le-eq* [*field-simps*]:
 assumes $0 < c$
 shows $b / c \leq a \longleftrightarrow b \leq a * c$
⟨*proof*⟩

lemma *pos-divide-less-eq* [*field-simps*]:
 assumes $0 < c$
 shows $b / c < a \longleftrightarrow b < a * c$
⟨*proof*⟩

lemma *neg-divide-le-eq* [*field-simps*]:
 assumes $c < 0$
 shows $b / c \leq a \longleftrightarrow a * c \leq b$
⟨*proof*⟩

lemma *neg-divide-less-eq* [*field-simps*]:
 assumes $c < 0$
 shows $b / c < a \longleftrightarrow a * c < b$
⟨*proof*⟩

The following *field-simps* rules are necessary, as minus is always moved atop of division but we want to get rid of division.

lemma *pos-le-minus-divide-eq* [*field-simps*]: $0 < c \Longrightarrow a \leq - (b / c) \longleftrightarrow a * c \leq - b$
 ⟨*proof*⟩

lemma *neg-le-minus-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow a \leq - (b / c) \longleftrightarrow - b \leq a * c$
 ⟨*proof*⟩

lemma *pos-less-minus-divide-eq* [*field-simps*]: $0 < c \Longrightarrow a < - (b / c) \longleftrightarrow a * c < - b$
 ⟨*proof*⟩

lemma *neg-less-minus-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow a < - (b \ / \ c) \longleftrightarrow - b < a * c$
⟨*proof*⟩

lemma *pos-minus-divide-less-eq* [*field-simps*]: $0 < c \Longrightarrow - (b \ / \ c) < a \longleftrightarrow - b < a * c$
⟨*proof*⟩

lemma *neg-minus-divide-less-eq* [*field-simps*]: $c < 0 \Longrightarrow - (b \ / \ c) < a \longleftrightarrow a * c < - b$
⟨*proof*⟩

lemma *pos-minus-divide-le-eq* [*field-simps*]: $0 < c \Longrightarrow - (b \ / \ c) \leq a \longleftrightarrow - b \leq a * c$
⟨*proof*⟩

lemma *neg-minus-divide-le-eq* [*field-simps*]: $c < 0 \Longrightarrow - (b \ / \ c) \leq a \longleftrightarrow a * c \leq - b$
⟨*proof*⟩

lemma *frac-less-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \ / \ y < w \ / \ z \longleftrightarrow (x * z - w * y) \ / \ (y * z) < 0$
⟨*proof*⟩

lemma *frac-le-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \ / \ y \leq w \ / \ z \longleftrightarrow (x * z - w * y) \ / \ (y * z) \leq 0$
⟨*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* = *algebra-simps zero-less-mult-iff mult-less-0-iff*

lemmas (**in** −) *sign-simps* = *algebra-simps zero-less-mult-iff mult-less-0-iff*

lemma *divide-pos-pos*[*simp*]:
0 < x ==> 0 < y ==> 0 < x / y
⟨*proof*⟩

lemma *divide-nonneg-pos*:
0 <= x ==> 0 < y ==> 0 <= x / y
⟨*proof*⟩

lemma *divide-neg-pos*:
x < 0 ==> 0 < y ==> x / 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*⟩

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 *unbounded-dense-linorder*
⟨*proof*⟩

subclass *field-abs-sgn* ⟨*proof*⟩

lemma *inverse-sgn* [*simp*]:
$inverse\ (sgn\ a) = sgn\ a$
⟨*proof*⟩

lemma *divide-sgn* [*simp*]:
$a / sgn\ b = a * sgn\ b$
⟨*proof*⟩

lemma *nonzero-abs-inverse*:
$a \neq 0 ==> |inverse\ a| = inverse\ |a|$
⟨*proof*⟩

lemma *nonzero-abs-divide*:
$b \neq 0 ==> |a / b| = |a| / |b|$
⟨*proof*⟩

lemma *field-le-epsilon*:

assumes e: $\bigwedge e.\ 0 < e \Longrightarrow x \leq y + e$
shows $x \leq y$
$\langle proof \rangle$

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 < inverse\ a) = (0 < a)$
$\langle proof \rangle$

lemma *inverse-negative-iff-negative* [*simp*]:
$(inverse\ a < 0) = (a < 0)$
$\langle proof \rangle$

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
$0 \leq inverse\ a \longleftrightarrow 0 \leq a$
$\langle proof \rangle$

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
$inverse\ a \leq 0 \longleftrightarrow a \leq 0$
$\langle proof \rangle$

lemma *one-less-inverse-iff*: $1 < inverse\ x \longleftrightarrow 0 < x \wedge x < 1$
$\langle proof \rangle$

lemma *one-le-inverse-iff*: $1 \leq inverse\ x \longleftrightarrow 0 < x \wedge x \leq 1$
$\langle proof \rangle$

lemma *inverse-less-1-iff*: $inverse\ x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$
$\langle proof \rangle$

lemma *inverse-le-1-iff*: $inverse\ x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$
$\langle proof \rangle$

lemma [*divide-simps*]:
shows *le-divide-eq*: $a \leq b\ /\ c \longleftrightarrow (if\ 0 < c\ then\ a * c \leq b\ else\ if\ c < 0\ then\ b \leq a * c\ else\ a \leq 0)$
and *divide-le-eq*: $b\ /\ c \leq a \longleftrightarrow (if\ 0 < c\ then\ b \leq a * c\ else\ if\ c < 0\ then\ a * c \leq b\ else\ 0 \leq a)$
and *less-divide-eq*: $a < b\ /\ c \longleftrightarrow (if\ 0 < c\ then\ a * c < b\ else\ if\ c < 0\ then\ b < a * c\ else\ a < 0)$
and *divide-less-eq*: $b\ /\ c < a \longleftrightarrow (if\ 0 < c\ then\ b < a * c\ else\ if\ c < 0\ then\ a * c < b\ else\ 0 < a)$
and *le-minus-divide-eq*: $a \leq -\ (b\ /\ c) \longleftrightarrow (if\ 0 < c\ then\ a * c \leq -\ b\ else\ if\ c < 0\ then\ -\ b \leq a * c\ else\ a \leq 0)$
and *minus-divide-le-eq*: $-\ (b\ /\ c) \leq a \longleftrightarrow (if\ 0 < c\ then\ -\ b \leq a * c\ else\ if\ c < 0\ then\ a * c \leq -\ b\ else\ 0 \leq a)$
and *less-minus-divide-eq*: $a < -\ (b\ /\ c) \longleftrightarrow (if\ 0 < c\ then\ a * c < -\ b\ else\ if\ c < 0\ then\ -\ b < a * c\ else\ \ a < 0)$
and *minus-divide-less-eq*: $-\ (b\ /\ c) < a \longleftrightarrow (if\ 0 < c\ then\ -\ b < a * c\ else\ if\ c < 0\ then\ a * c < -\ b\ else\ 0 < a)$

⟨proof⟩

Division and Signs

lemma
shows *zero-less-divide-iff*: $0 < a\ /\ b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
and *divide-less-0-iff*: $a\ /\ b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
and *zero-le-divide-iff*: $0 \leq a\ /\ b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$
and *divide-le-0-iff*: $a\ /\ b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
⟨proof⟩

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp*]: $0 = 1\ /\ a \longleftrightarrow a = 0$
⟨proof⟩

lemma *one-divide-eq-0-iff* [*simp*]: $1\ /\ a = 0 \longleftrightarrow a = 0$
⟨proof⟩

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma *zero-le-divide-1-iff* [*simp*]:
$0 \leq 1\ /\ a \longleftrightarrow 0 \leq a$
⟨proof⟩

lemma *zero-less-divide-1-iff* [*simp*]:
$0 < 1\ /\ a \longleftrightarrow 0 < a$
⟨proof⟩

lemma *divide-le-0-1-iff* [*simp*]:
$1\ /\ a \leq 0 \longleftrightarrow a \leq 0$
⟨proof⟩

lemma *divide-less-0-1-iff* [*simp*]:
$1\ /\ a < 0 \longleftrightarrow a < 0$
⟨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⟩

lemma *inverse-le-iff*: $inverse\ a \leq inverse\ b \longleftrightarrow (0 < a * b \longrightarrow b \leq a) \wedge (a * b \leq 0 \longrightarrow a \leq b)$
⟨*proof*⟩

lemma *inverse-less-iff*: $inverse\ a < inverse\ b \longleftrightarrow (0 < a * b \longrightarrow b < a) \wedge (a * b \leq 0 \longrightarrow a < b)$
⟨*proof*⟩

lemma *divide-le-cancel*: $a\ /\ c \leq b\ /\ c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
⟨*proof*⟩

lemma *divide-less-cancel*: $a\ /\ c < b\ /\ c \longleftrightarrow (0 < c \longrightarrow a < b) \wedge (c < 0 \longrightarrow b < a) \wedge c \neq 0$
⟨*proof*⟩

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1*:
$(1 \leq b\ /\ a) = ((0 < a\ \&\ a \leq b)\ |\ (a < 0\ \&\ b \leq a))$
⟨*proof*⟩

lemma *divide-le-eq-1*:
$(b\ /\ a \leq 1) = ((0 < a\ \&\ b \leq a)\ |\ (a < 0\ \&\ a \leq b)\ |\ a{=}0)$
⟨*proof*⟩

lemma *less-divide-eq-1*:
$(1 < b\ /\ a) = ((0 < a\ \&\ a < b)\ |\ (a < 0\ \&\ b < a))$
⟨*proof*⟩

lemma *divide-less-eq-1*:
$(b\ /\ a < 1) = ((0 < a\ \&\ b < a)\ |\ (a < 0\ \&\ a < b)\ |\ a{=}0)$
⟨*proof*⟩

lemma *divide-nonneg-nonneg* [*simp*]:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x\ /\ y$
⟨*proof*⟩

lemma *divide-nonpos-nonpos*:
$x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow 0 \leq x\ /\ y$
⟨*proof*⟩

lemma *divide-nonneg-nonpos*:
$0 \leq x \Longrightarrow y \leq 0 \Longrightarrow x\ /\ y \leq 0$
⟨*proof*⟩

lemma *divide-nonpos-nonneg*:
$x \leq 0 \Longrightarrow 0 \leq y \Longrightarrow x\ /\ y \leq 0$
⟨*proof*⟩

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
⟨*proof*⟩

lemma *le-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
⟨*proof*⟩

lemma *divide-le-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
⟨*proof*⟩

lemma *less-divide-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (1 < b/a) = (a < b)$
⟨*proof*⟩

lemma *less-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 < b/a) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a < 1) = (b < a)$
⟨*proof*⟩

lemma *divide-less-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow b/a < 1 \longleftrightarrow a < b$
⟨*proof*⟩

lemma *eq-divide-eq-1* [*simp*]:
$(1 = b/a) = ((a \neq 0 \ \& \ a = b))$
⟨*proof*⟩

lemma *divide-eq-eq-1* [*simp*]:
$(b/a = 1) = ((a \neq 0 \ \& \ a = b))$
⟨*proof*⟩

lemma *abs-div-pos*: $0 < y ==>$
$|x| \ / \ y = |x \ / \ y|$
⟨*proof*⟩

lemma *zero-le-divide-abs-iff* [*simp*]: $(0 \leq a \ / \ |b|) = (0 \leq a \mid b = 0)$
⟨*proof*⟩

lemma *divide-le-0-abs-iff* [*simp*]: $(a \ / \ |b| \leq 0) = (a \leq 0 \mid b = 0)$
⟨*proof*⟩

lemma *field-le-mult-one-interval*:
assumes $*$: $\bigwedge z.\ [\![\ 0 < z\ ;\ z < 1\ ]\!] \Longrightarrow z * x \le y$
shows $x \le y$
⟨*proof*⟩

For creating values between u and v.

lemma *scaling-mono*:
assumes $u \le v\ 0 \le r\ r \le s$
shows $u + r * (v - u) / s \le v$
⟨*proof*⟩

end

Min/max Simplification Rules

lemma *min-mult-distrib-left*:
fixes $x{::}'a{::}linordered\text{-}idom$
shows $p * min\ x\ y = (if\ 0 \le p\ then\ min\ (p{*}x)\ (p{*}y)\ else\ max\ (p{*}x)\ (p{*}y))$
⟨*proof*⟩

lemma *min-mult-distrib-right*:
fixes $x{::}'a{::}linordered\text{-}idom$
shows $min\ x\ y * p = (if\ 0 \le p\ then\ min\ (x{*}p)\ (y{*}p)\ else\ max\ (x{*}p)\ (y{*}p))$
⟨*proof*⟩

lemma *min-divide-distrib-right*:
fixes $x{::}'a{::}linordered\text{-}field$
shows $min\ x\ y\ /\ p = (if\ 0 \le p\ then\ min\ (x/p)\ (y/p)\ else\ max\ (x/p)\ (y/p))$
⟨*proof*⟩

lemma *max-mult-distrib-left*:
fixes $x{::}'a{::}linordered\text{-}idom$
shows $p * max\ x\ y = (if\ 0 \le p\ then\ max\ (p{*}x)\ (p{*}y)\ else\ min\ (p{*}x)\ (p{*}y))$
⟨*proof*⟩

lemma *max-mult-distrib-right*:
fixes $x{::}'a{::}linordered\text{-}idom$
shows $max\ x\ y * p = (if\ 0 \le p\ then\ max\ (x{*}p)\ (y{*}p)\ else\ min\ (x{*}p)\ (y{*}p))$
⟨*proof*⟩

lemma *max-divide-distrib-right*:
fixes $x{::}'a{::}linordered\text{-}field$
shows $max\ x\ y\ /\ p = (if\ 0 \le p\ then\ max\ (x/p)\ (y/p)\ else\ min\ (x/p)\ (y/p))$
⟨*proof*⟩

hide-fact (**open**) *field-inverse field-divide-inverse field-inverse-zero*

code-identifier
code-module *Fields* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

18 Finite sets

theory *Finite-Set*
  imports *Product-Type Sum-Type Fields*
begin

18.1 Predicate for finite sets

context notes [[*inductive-internals*]]
begin

inductive *finite* :: *'a set* ⇒ *bool*
  where
    emptyI [*simp*, *intro*!]: *finite* {}
  | *insertI* [*simp*, *intro*!]: *finite A* ⟹ *finite* (*insert a A*)

end

⟨*ML*⟩

declare [[*simproc del*: *finite-Collect*]]

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
  — Discharging $x \notin F$ entails extra work.
  assumes *finite F*
  assumes *P* {}
    and *insert*: $\bigwedge x\ F.$ *finite F* ⟹ $x \notin F$ ⟹ *P F* ⟹ *P* (*insert x F*)
  shows *P F*
  ⟨*proof*⟩

lemma *infinite-finite-induct* [*case-names infinite empty insert*]:
  assumes *infinite*: $\bigwedge A.$ ¬ *finite A* ⟹ *P A*
    and *empty*: *P* {}
    and *insert*: $\bigwedge x\ F.$ *finite F* ⟹ $x \notin F$ ⟹ *P F* ⟹ *P* (*insert x F*)
  shows *P A*
⟨*proof*⟩

18.1.1 Choice principles

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*. $a \notin A$
⟨*proof*⟩

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*: *finite A* ⟹ ∀ *x*∈*A*. ∃ *y*. *P x y* ⟹ ∃ *f*. ∀ *x*∈*A*. *P x* (*f x*)

⟨*proof*⟩

18.1.2 Finite sets are the images of initial segments of natural numbers

lemma *finite-imp-nat-seg-image-inj-on*:
assumes *finite* A
shows $\exists (n::nat)\ f.\ A = f\ `\ \{i.\ i < n\} \wedge inj\text{-}on\ f\ \{i.\ i < n\}$
⟨*proof*⟩

lemma *nat-seg-image-imp-finite*: $A = f\ `\ \{i::nat.\ i < n\} \Longrightarrow finite\ A$
⟨*proof*⟩

lemma *finite-conv-nat-seg-image*: $finite\ A \longleftrightarrow (\exists\, n\, f.\ A = f\ `\ \{i::nat.\ i < n\})$
⟨*proof*⟩

lemma *finite-imp-inj-to-nat-seg*:
assumes *finite* A
shows $\exists f\, n.\ f\ `\ A = \{i::nat.\ i < n\} \wedge inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *finite-Collect-less-nat* [*iff*]: $finite\ \{n::nat.\ n < k\}$
⟨*proof*⟩

lemma *finite-Collect-le-nat* [*iff*]: $finite\ \{n::nat.\ n \le k\}$
⟨*proof*⟩

18.1.3 Finiteness and common set operations

lemma *rev-finite-subset*: $finite\ B \Longrightarrow A \subseteq B \Longrightarrow finite\ A$
⟨*proof*⟩

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

lemma *finite-UnI*:
assumes *finite* F **and** *finite* G
shows *finite* $(F \cup G)$
⟨*proof*⟩

lemma *finite-Un* [*iff*]: $finite\ (F \cup G) \longleftrightarrow finite\ F \wedge finite\ G$
⟨*proof*⟩

lemma *finite-insert* [*simp*]: $finite\ (insert\ a\ A) \longleftrightarrow finite\ A$
⟨*proof*⟩

lemma *finite-Int* [*simp*, *intro*]: $finite\ F \vee finite\ G \Longrightarrow finite\ (F \cap G)$
⟨*proof*⟩

lemma *finite-Collect-conjI* [*simp*, *intro*]:

$finite\ \{x.\ P\ x\} \vee finite\ \{x.\ Q\ x\} \Longrightarrow finite\ \{x.\ P\ x \wedge Q\ x\}$
⟨*proof*⟩

lemma *finite-Collect-disjI* [*simp*]:
$finite\ \{x.\ P\ x \vee Q\ x\} \longleftrightarrow finite\ \{x.\ P\ x\} \wedge finite\ \{x.\ Q\ x\}$
⟨*proof*⟩

lemma *finite-Diff* [*simp*, *intro*]: $finite\ A \Longrightarrow finite\ (A - B)$
⟨*proof*⟩

lemma *finite-Diff2* [*simp*]:
assumes *finite B*
shows $finite\ (A - B) \longleftrightarrow finite\ A$
⟨*proof*⟩

lemma *finite-Diff-insert* [*iff*]: $finite\ (A - insert\ a\ B) \longleftrightarrow finite\ (A - B)$
⟨*proof*⟩

lemma *finite-compl* [*simp*]:
$finite\ (A :: 'a\ set) \Longrightarrow finite\ (-\ A) \longleftrightarrow finite\ (UNIV :: 'a\ set)$
⟨*proof*⟩

lemma *finite-Collect-not* [*simp*]:
$finite\ \{x :: 'a.\ P\ x\} \Longrightarrow finite\ \{x.\ \neg\ P\ x\} \longleftrightarrow finite\ (UNIV :: 'a\ set)$
⟨*proof*⟩

lemma *finite-Union* [*simp*, *intro*]:
$finite\ A \Longrightarrow (\bigwedge M.\ M \in A \Longrightarrow finite\ M) \Longrightarrow finite\ (\bigcup A)$
⟨*proof*⟩

lemma *finite-UN-I* [*intro*]:
$finite\ A \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow finite\ (B\ a)) \Longrightarrow finite\ (\bigcup a{\in}A.\ B\ a)$
⟨*proof*⟩

lemma *finite-UN* [*simp*]: $finite\ A \Longrightarrow finite\ (UNION\ A\ B) \longleftrightarrow (\forall x{\in}A.\ finite\ (B\ x))$
⟨*proof*⟩

lemma *finite-Inter* [*intro*]: $\exists A{\in}M.\ finite\ A \Longrightarrow finite\ (\bigcap M)$
⟨*proof*⟩

lemma *finite-INT* [*intro*]: $\exists x{\in}I.\ finite\ (A\ x) \Longrightarrow finite\ (\bigcap x{\in}I.\ A\ x)$
⟨*proof*⟩

lemma *finite-imageI* [*simp*, *intro*]: $finite\ F \Longrightarrow finite\ (h\ `\ F)$
⟨*proof*⟩

lemma *finite-image-set* [*simp*]: $finite\ \{x.\ P\ x\} \Longrightarrow finite\ \{f\ x\ |x.\ P\ x\}$
⟨*proof*⟩

lemma *finite-image-set2*:
$finite\ \{x.\ P\ x\} \Longrightarrow finite\ \{y.\ Q\ y\} \Longrightarrow finite\ \{f\ x\ y\ |x\ y.\ P\ x \wedge Q\ y\}$
⟨*proof*⟩

lemma *finite-imageD*:
assumes $finite\ (f\ `\ A)$ **and** $inj\text{-}on\ f\ A$
shows $finite\ A$
⟨*proof*⟩

lemma *finite-image-iff*: $inj\text{-}on\ f\ A \Longrightarrow finite\ (f\ `\ A) \longleftrightarrow finite\ A$
⟨*proof*⟩

lemma *finite-surj*: $finite\ A \Longrightarrow B \subseteq f\ `\ A \Longrightarrow finite\ B$
⟨*proof*⟩

lemma *finite-range-imageI*: $finite\ (range\ g) \Longrightarrow finite\ (range\ (\lambda x.\ f\ (g\ x)))$
⟨*proof*⟩

lemma *finite-subset-image*:
assumes $finite\ B$
shows $B \subseteq f\ `\ A \Longrightarrow \exists\, C{\subseteq}A.\ finite\ C \wedge B = f\ `\ C$
⟨*proof*⟩

lemma *finite-vimage-IntI*: $finite\ F \Longrightarrow inj\text{-}on\ h\ A \Longrightarrow finite\ (h\ -`\ F \cap A)$
⟨*proof*⟩

lemma *finite-finite-vimage-IntI*:
assumes $finite\ F$
and $\bigwedge y.\ y \in F \Longrightarrow finite\ ((h\ -`\ \{y\}) \cap A)$
shows $finite\ (h\ -`\ F \cap A)$
⟨*proof*⟩

lemma *finite-vimageI*: $finite\ F \Longrightarrow inj\ h \Longrightarrow finite\ (h\ -`\ F)$
⟨*proof*⟩

lemma *finite-vimageD'*: $finite\ (f\ -`\ A) \Longrightarrow A \subseteq range\ f \Longrightarrow finite\ A$
⟨*proof*⟩

lemma *finite-vimageD*: $finite\ (h\ -`\ F) \Longrightarrow surj\ h \Longrightarrow finite\ F$
⟨*proof*⟩

lemma *finite-vimage-iff*: $bij\ h \Longrightarrow finite\ (h\ -`\ F) \longleftrightarrow finite\ F$
⟨*proof*⟩

lemma *finite-Collect-bex* [*simp*]:
assumes $finite\ A$
shows $finite\ \{x.\ \exists\, y{\in}A.\ Q\ x\ y\} \longleftrightarrow (\forall\, y{\in}A.\ finite\ \{x.\ Q\ x\ y\})$
⟨*proof*⟩

lemma *finite-Collect-bounded-ex* [*simp*]:
assumes *finite* $\{y.\ P\ y\}$
shows *finite* $\{x.\ \exists\, y.\ P\ y \wedge Q\ x\ y\} \longleftrightarrow (\forall\, y.\ P\ y \longrightarrow$ *finite* $\{x.\ Q\ x\ y\})$
⟨*proof*⟩

lemma *finite-Plus*: *finite* $A \Longrightarrow$ *finite* $B \Longrightarrow$ *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* $(A <+> B) \longleftrightarrow$ *finite* $A \wedge$ *finite* B
⟨*proof*⟩

lemma *finite-Plus-UNIV-iff* [*simp*]:
finite $(UNIV :: ('a + 'b)\ set) \longleftrightarrow$ *finite* $(UNIV :: 'a\ set) \wedge$ *finite* $(UNIV :: 'b\ set)$
⟨*proof*⟩

lemma *finite-SigmaI* [*simp*, *intro*]:
finite $A \Longrightarrow (\bigwedge a.\ a{\in}A \Longrightarrow$ *finite* $(B\ a)) \Longrightarrow$ *finite* $(SIGMA\ a{:}A.\ B\ a)$
⟨*proof*⟩

lemma *finite-SigmaI2*:
assumes *finite* $\{x{\in}A.\ B\ x \neq \{\}\}$
and $\bigwedge a.\ a \in A \Longrightarrow$ *finite* $(B\ a)$
shows *finite* $(Sigma\ A\ B)$
⟨*proof*⟩

lemma *finite-cartesian-product*: *finite* $A \Longrightarrow$ *finite* $B \Longrightarrow$ *finite* $(A \times B)$
⟨*proof*⟩

lemma *finite-Prod-UNIV*:
finite $(UNIV :: 'a\ set) \Longrightarrow$ *finite* $(UNIV :: 'b\ set) \Longrightarrow$ *finite* $(UNIV :: ('a \times 'b)\ set)$
⟨*proof*⟩

lemma *finite-cartesian-productD1*:
assumes *finite* $(A \times B)$ **and** $B \neq \{\}$
shows *finite* A
⟨*proof*⟩

lemma *finite-cartesian-productD2*:
assumes *finite* $(A \times B)$ **and** $A \neq \{\}$
shows *finite* B

⟨*proof*⟩

lemma *finite-cartesian-product-iff*:
$finite\ (A \times B) \longleftrightarrow (A = \{\} \lor B = \{\} \lor (finite\ A \land finite\ B))$
⟨*proof*⟩

lemma *finite-prod*:
$finite\ (UNIV :: ('a \times 'b)\ set) \longleftrightarrow finite\ (UNIV :: 'a\ set) \land finite\ (UNIV :: 'b\ set)$
⟨*proof*⟩

lemma *finite-Pow-iff* [*iff*]: $finite\ (Pow\ A) \longleftrightarrow finite\ A$
⟨*proof*⟩

corollary *finite-Collect-subsets* [*simp*, *intro*]: $finite\ A \Longrightarrow finite\ \{B.\ B \subseteq A\}$
⟨*proof*⟩

lemma *finite-set*: $finite\ (UNIV :: 'a\ set\ set) \longleftrightarrow finite\ (UNIV :: 'a\ set)$
⟨*proof*⟩

lemma *finite-UnionD*: $finite\ (\bigcup A) \Longrightarrow finite\ A$
⟨*proof*⟩

lemma *finite-set-of-finite-funs*:
assumes $finite\ A$ $finite\ B$
shows $finite\ \{f.\ \forall x.\ (x \in A \longrightarrow f\ x \in B) \land (x \notin A \longrightarrow f\ x = d)\}$ (**is** $finite\ ?S$)
⟨*proof*⟩

lemma *not-finite-existsD*:
assumes $\neg\ finite\ \{a.\ P\ a\}$
shows $\exists\ a.\ P\ a$
⟨*proof*⟩

18.1.4 Further induction rules on finite sets

lemma *finite-ne-induct* [*case-names singleton insert*, *consumes 2*]:
assumes $finite\ F$ **and** $F \neq \{\}$
assumes $\bigwedge x.\ P\ \{x\}$
and $\bigwedge x\ F.\ finite\ F \Longrightarrow F \neq \{\} \Longrightarrow x \notin F \Longrightarrow P\ F \Longrightarrow P\ (insert\ x\ F)$
shows $P\ F$
⟨*proof*⟩

lemma *finite-subset-induct* [*consumes 2*, *case-names empty insert*]:
assumes $finite\ F$ **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: $\bigwedge a\ F.\ finite\ F \Longrightarrow a \in A \Longrightarrow a \notin F \Longrightarrow P\ F \Longrightarrow P\ (insert\ a\ F)$
shows $P\ F$
⟨*proof*⟩

lemma *finite-empty-induct*:
assumes *finite* A
and $P\ A$
and *remove*: $\bigwedge a\ A.\ \mathit{finite}\ A \Longrightarrow a \in A \Longrightarrow P\ A \Longrightarrow P\ (A - \{a\})$
shows $P\ \{\}$
⟨*proof*⟩

lemma *finite-update-induct* [*consumes 1*, *case-names const update*]:
assumes *finite*: *finite* $\{a.\ f\ a \neq c\}$
and *const*: $P\ (\lambda a.\ c)$
and *update*: $\bigwedge a\ b\ f.\ \mathit{finite}\ \{a.\ f\ a \neq c\} \Longrightarrow f\ a = c \Longrightarrow b \neq c \Longrightarrow P\ f \Longrightarrow P$
$(f(a := b))$
shows $P\ f$
⟨*proof*⟩

lemma *finite-subset-induct′* [*consumes 2*, *case-names empty insert*]:
assumes *finite* F **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: $\bigwedge a\ F.\ [\![\mathit{finite}\ F;\ a \in A;\ F \subseteq A;\ a \notin F;\ P\ F\ ]\!] \Longrightarrow P\ (\mathit{insert}\ a\ F)$
shows $P\ F$
⟨*proof*⟩

18.2 Class *finite*

class *finite* =
assumes *finite-UNIV*: *finite* $(\mathit{UNIV} :: {'a}\ \mathit{set})$
begin

lemma *finite* [*simp*]: *finite* $(A :: {'a}\ \mathit{set})$
⟨*proof*⟩

lemma *finite-code* [*code*]: *finite* $(A :: {'a}\ \mathit{set}) \longleftrightarrow \mathit{True}$
⟨*proof*⟩

end

instance *prod* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

lemma *inj-graph*: $\mathit{inj}\ (\lambda f.\ \{(x,\ y).\ y = f\ x\})$
⟨*proof*⟩

instance *fun* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

instance *bool* :: *finite*
⟨*proof*⟩

instance *set* :: (*finite*) *finite*

⟨*proof*⟩

instance *unit* :: *finite*
⟨*proof*⟩

instance *sum* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

18.3 A basic fold functional for finite sets

The intended behaviour is $fold\ f\ z\ \{x_1, \ldots, x_n\} = f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is "left-commutative":

locale *comp-fun-commute* =
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$
assumes *comp-fun-commute*: $f\ y \circ f\ x = f\ x \circ f\ y$
begin

lemma *fun-left-comm*: $f\ y\ (f\ x\ z) = f\ x\ (f\ y\ z)$
⟨*proof*⟩

lemma *commute-left-comp*: $f\ y \circ (f\ x \circ g) = f\ x \circ (f\ y \circ g)$
⟨*proof*⟩

end

inductive *fold-graph* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$ **and** $z :: 'b$
where
emptyI [*intro*]: *fold-graph* $f\ z\ \{\}\ z$
| *insertI* [*intro*]: $x \notin A \Longrightarrow$ *fold-graph* $f\ z\ A\ y \Longrightarrow$ *fold-graph* $f\ z\ (insert\ x\ A)\ (f\ x\ y)$

inductive-cases *empty-fold-graphE* [*elim*!]: *fold-graph* $f\ z\ \{\}\ x$

definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b$
where *fold* $f\ z\ A$ = (*if finite* A *then* (*THE* y. *fold-graph* $f\ z\ A\ y$) *else* z)

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 *finite-imp-fold-graph*: *finite* $A \Longrightarrow \exists x.$ *fold-graph* $f\ z\ A\ x$
⟨*proof*⟩

18.3.1 From *fold-graph* to *fold*

context *comp-fun-commute*
begin

lemma *fold-graph-finite*:
assumes *fold-graph f z A y*
shows *finite A*
⟨*proof*⟩

lemma *fold-graph-insertE-aux*:
fold-graph f z A y $\Longrightarrow a \in A \Longrightarrow \exists y'.\ y = f\ a\ y' \land$ *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*:
assumes *finite A*
shows *fold-graph f z A* (*fold f z A*)
⟨*proof*⟩

The base case for *fold*:

lemma (**in** −) *fold-infinite* [*simp*]: ¬ *finite A* $\Longrightarrow$ *fold f z A* $= z$
⟨*proof*⟩

lemma (**in** −) *fold-empty* [*simp*]: *fold f z* $\{\} = z$
⟨*proof*⟩

The various recursion equations for *fold*:

lemma *fold-insert* [*simp*]:
assumes *finite A* **and** $x \notin A$
shows *fold f z* (*insert x A*) $= f\ x$ (*fold f z A*)
⟨*proof*⟩

declare (**in** −) *empty-fold-graphE* [*rule del*] *fold-graph.intros* [*rule del*]
— No more proofs involve these.

lemma *fold-fun-left-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 $\mathit{fold}\ f\ z\ A = f\ x\ (\mathit{fold}\ f\ z\ (A - \{x\}))$
$\langle \mathit{proof} \rangle$

lemma *fold-insert-remove*:
  assumes *finite* A
  shows $\mathit{fold}\ f\ z\ (\mathit{insert}\ x\ A) = f\ x\ (\mathit{fold}\ f\ z\ (A - \{x\}))$
$\langle \mathit{proof} \rangle$

lemma *fold-set-union-disj*:
  assumes *finite* A *finite* B $A \cap B = \{\}$
  shows $\mathit{Finite\text{-}Set.fold}\ f\ z\ (A \cup B) = \mathit{Finite\text{-}Set.fold}\ f\ (\mathit{Finite\text{-}Set.fold}\ f\ z\ A)\ B$
  $\langle \mathit{proof} \rangle$

end

Other properties of *fold*:

lemma *fold-image*:
  assumes *inj-on* g A
  shows $\mathit{fold}\ f\ z\ (g\ `\ A) = \mathit{fold}\ (f \circ g)\ z\ A$
$\langle \mathit{proof} \rangle$

lemma *fold-cong*:
  assumes *comp-fun-commute* f *comp-fun-commute* g
    and *finite* A
    and *cong*: $\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x$
    and $s = t$ **and** $A = B$
  shows $\mathit{fold}\ f\ s\ A = \mathit{fold}\ g\ t\ B$
$\langle \mathit{proof} \rangle$

A simplified version for idempotent functions:

locale *comp-fun-idem* = *comp-fun-commute* +
  assumes *comp-fun-idem*: $f\ x \circ f\ x = f\ x$
begin

lemma *fun-left-idem*: $f\ x\ (f\ x\ z) = f\ x\ z$
  $\langle \mathit{proof} \rangle$

lemma *fold-insert-idem*:
  assumes *fin*: *finite* A
  shows $\mathit{fold}\ f\ z\ (\mathit{insert}\ x\ A) = f\ x\ (\mathit{fold}\ f\ z\ A)$
$\langle \mathit{proof} \rangle$

declare *fold-insert* [*simp del*] *fold-insert-idem* [*simp*]

lemma *fold-insert-idem2*: $\mathit{finite}\ A \Longrightarrow \mathit{fold}\ f\ z\ (\mathit{insert}\ x\ A) = \mathit{fold}\ f\ (f\ x\ z)\ A$
  $\langle \mathit{proof} \rangle$

end

18.3.2 Liftings to *comp-fun-commute* etc.

lemma (**in** *comp-fun-commute*) *comp-comp-fun-commute*: *comp-fun-commute* (f $\circ$ g)
⟨*proof*⟩

lemma (**in** *comp-fun-idem*) *comp-comp-fun-idem*: *comp-fun-idem* ($f \circ g$)
⟨*proof*⟩

lemma (**in** *comp-fun-commute*) *comp-fun-commute-funpow*: *comp-fun-commute* ($\lambda x.$ $f\ x$ ^^ $g\ x$)
⟨*proof*⟩

18.3.3 Expressing set operations via *fold*

lemma *comp-fun-commute-const*: *comp-fun-commute* (λ-. f)
⟨*proof*⟩

lemma *comp-fun-idem-insert*: *comp-fun-idem insert*
⟨*proof*⟩

lemma *comp-fun-idem-remove*: *comp-fun-idem Set.remove*
⟨*proof*⟩

lemma (**in** *semilattice-inf*) *comp-fun-idem-inf*: *comp-fun-idem inf*
⟨*proof*⟩

lemma (**in** *semilattice-sup*) *comp-fun-idem-sup*: *comp-fun-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 Set.remove* B A
⟨*proof*⟩

lemma *comp-fun-commute-filter-fold*:
comp-fun-commute ($\lambda x\ A'.$ *if* $P\ x$ *then Set.insert* $x\ A'$ *else* A')
⟨*proof*⟩

lemma *Set-filter-fold*:
assumes *finite* A
shows *Set.filter* $P\ A =$ *fold* ($\lambda x\ A'.$ *if* $P\ x$ *then Set.insert* $x\ A'$ *else* A') {} A
⟨*proof*⟩

lemma *inter-Set-filter*:
  assumes *finite* B
  shows $A \cap B = Set.filter\ (\lambda x.\ x \in A)\ B$
  ⟨*proof*⟩

lemma *image-fold-insert*:
  assumes *finite* A
  shows $image\ f\ A = fold\ (\lambda k\ A.\ Set.insert\ (f\ k)\ A)\ \{\}\ A$
⟨*proof*⟩

lemma *Ball-fold*:
  assumes *finite* A
  shows $Ball\ A\ P = fold\ (\lambda k\ s.\ s \wedge P\ k)\ True\ A$
⟨*proof*⟩

lemma *Bex-fold*:
  assumes *finite* A
  shows $Bex\ A\ P = fold\ (\lambda k\ s.\ s \vee P\ k)\ False\ A$
⟨*proof*⟩

lemma *comp-fun-commute-Pow-fold*: $comp\text{-}fun\text{-}commute\ (\lambda x\ A.\ A \cup Set.insert\ x$ ‘ $A)$
  ⟨*proof*⟩

lemma *Pow-fold*:
  assumes *finite* A
  shows $Pow\ A = fold\ (\lambda x\ A.\ A \cup Set.insert\ x$ ‘ $A)\ \{\{\}\}\ A$
⟨*proof*⟩

lemma *fold-union-pair*:
  assumes *finite* B
  shows $(\bigcup y{\in}B.\ \{(x,\ y)\}) \cup A = fold\ (\lambda y.\ Set.insert\ (x,\ y))\ A\ B$
⟨*proof*⟩

lemma *comp-fun-commute-product-fold*:
  $finite\ B \Longrightarrow comp\text{-}fun\text{-}commute\ (\lambda x\ z.\ fold\ (\lambda y.\ Set.insert\ (x,\ y))\ z\ B)$
  ⟨*proof*⟩

lemma *product-fold*:
  assumes *finite* A *finite* B
  shows $A \times B = fold\ (\lambda x\ z.\ fold\ (\lambda y.\ Set.insert\ (x,\ y))\ z\ B)\ \{\}\ A$
  ⟨*proof*⟩

context *complete-lattice*
begin

lemma *inf-Inf-fold-inf*:
  assumes *finite* A

shows *inf* (*Inf A*) *B* = *fold inf B A*
⟨*proof*⟩

lemma *sup-Sup-fold-sup*:
assumes *finite A*
shows *sup* (*Sup A*) *B* = *fold sup B A*
⟨*proof*⟩

lemma *Inf-fold-inf*: *finite A* ⟹ *Inf A* = *fold inf top A*
⟨*proof*⟩

lemma *Sup-fold-sup*: *finite A* ⟹ *Sup A* = *fold sup bot A*
⟨*proof*⟩

lemma *inf-INF-fold-inf*:
assumes *finite A*
shows *inf B* (*INFIMUM A f*) = *fold* (*inf* ∘ *f*) *B A* (**is** *?inf* = *?fold*)
⟨*proof*⟩

lemma *sup-SUP-fold-sup*:
assumes *finite A*
shows *sup B* (*SUPREMUM A f*) = *fold* (*sup* ∘ *f*) *B A* (**is** *?sup* = *?fold*)
⟨*proof*⟩

lemma *INF-fold-inf*: *finite A* ⟹ *INFIMUM A f* = *fold* (*inf* ∘ *f*) *top A*
⟨*proof*⟩

lemma *SUP-fold-sup*: *finite A* ⟹ *SUPREMUM A f* = *fold* (*sup* ∘ *f*) *bot A*
⟨*proof*⟩

end

18.4 Locales as mini-packages for fold operations

18.4.1 The natural case

locale *folding* =
fixes *f* :: *'a* ⇒ *'b* ⇒ *'b* **and** *z* :: *'b*
assumes *comp-fun-commute*: *f y* ∘ *f x* = *f x* ∘ *f y*
begin

interpretation *fold?*: *comp-fun-commute f*
⟨*proof*⟩

definition *F* :: *'a set* ⇒ *'b*
where *eq-fold*: *F A* = *fold f z A*

lemma *empty* [*simp*]:*F* {} = *z*
⟨*proof*⟩

lemma *infinite* [*simp*]: $\neg$ *finite* $A \Longrightarrow F\ A = z$
⟨*proof*⟩

lemma *insert* [*simp*]:
assumes *finite* A **and** $x \notin A$
shows F (*insert* x A) $= f\ x\ (F\ A)$
⟨*proof*⟩

lemma *remove*:
assumes *finite* A **and** $x \in A$
shows $F\ A = f\ x\ (F\ (A - \{x\}))$
⟨*proof*⟩

lemma *insert-remove*: *finite* $A \Longrightarrow F$ (*insert* x A) $= f\ x\ (F\ (A - \{x\}))$
⟨*proof*⟩

end

18.4.2 With idempotency

locale *folding-idem* = *folding* +
assumes *comp-fun-idem*: $f\ x \circ f\ x = f\ x$
begin

declare *insert* [*simp del*]

interpretation *fold?*: *comp-fun-idem f*
⟨*proof*⟩

lemma *insert-idem* [*simp*]:
assumes *finite* A
shows F (*insert* x A) $= f\ x\ (F\ A)$
⟨*proof*⟩

end

18.5 Finite cardinality

The traditional definition *card* $A \equiv$ *LEAST* $n.\ \exists f.\ A = \{f\ i\ |i.\ i < n\}$ is ugly to work with. But now that we have *fold* things are easy:

global-interpretation *card*: *folding* λ-. *Suc 0*
defines *card* = *folding.F* (λ-. *Suc*) *0*
⟨*proof*⟩

lemma *card-infinite*: $\neg$ *finite* $A \Longrightarrow$ *card* $A = 0$
⟨*proof*⟩

lemma *card-empty*: *card* $\{\} = 0$
⟨*proof*⟩

lemma *card-insert-disjoint*: $finite\ A \Longrightarrow x \notin A \Longrightarrow card\ (insert\ x\ A) = Suc\ (card\ A)$
⟨*proof*⟩

lemma *card-insert-if*: $finite\ A \Longrightarrow card\ (insert\ x\ A) = (if\ x \in A\ then\ card\ A\ else\ Suc\ (card\ A))$
⟨*proof*⟩

lemma *card-ge-0-finite*: $card\ A > 0 \Longrightarrow finite\ A$
⟨*proof*⟩

lemma *card-0-eq* [*simp*]: $finite\ A \Longrightarrow card\ A = 0 \longleftrightarrow A = \{\}$
⟨*proof*⟩

lemma *finite-UNIV-card-ge-0*: $finite\ (UNIV :: 'a\ set) \Longrightarrow card\ (UNIV :: 'a\ set) > 0$
⟨*proof*⟩

lemma *card-eq-0-iff*: $card\ A = 0 \longleftrightarrow A = \{\} \lor \neg\ finite\ A$
⟨*proof*⟩

lemma *card-range-greater-zero*: $finite\ (range\ f) \Longrightarrow card\ (range\ f) > 0$
⟨*proof*⟩

lemma *card-gt-0-iff*: $0 < card\ A \longleftrightarrow A \neq \{\} \land finite\ A$
⟨*proof*⟩

lemma *card-Suc-Diff1*: $finite\ A \Longrightarrow x \in A \Longrightarrow Suc\ (card\ (A - \{x\})) = card\ A$
⟨*proof*⟩

lemma *card-insert-le-m1*: $n > 0 \Longrightarrow card\ y \le n - 1 \Longrightarrow card\ (insert\ x\ y) \le n$
⟨*proof*⟩

lemma *card-Diff-singleton*: $finite\ A \Longrightarrow x \in A \Longrightarrow card\ (A - \{x\}) = card\ A - 1$
⟨*proof*⟩

lemma *card-Diff-singleton-if*:
$finite\ A \Longrightarrow card\ (A - \{x\}) = (if\ x \in A\ then\ card\ A - 1\ else\ card\ A)$
⟨*proof*⟩

lemma *card-Diff-insert*[*simp*]:
assumes $finite\ A$ **and** $a \in A$ **and** $a \notin B$
shows $card\ (A - insert\ a\ B) = card\ (A - B) - 1$
⟨*proof*⟩

lemma *card-insert*: $finite\ A \Longrightarrow card\ (insert\ x\ A) = Suc\ (card\ (A - \{x\}))$
⟨*proof*⟩

lemma *card-insert-le*: *finite* $A \Longrightarrow$ *card* $A \leq$ *card* (*insert* x A)
⟨*proof*⟩

lemma *card-Collect-less-nat*[*simp*]: *card* $\{i::nat.\ i < n\} = n$
⟨*proof*⟩

lemma *card-Collect-le-nat*[*simp*]: *card* $\{i::nat.\ i \leq n\} =$ *Suc* n
⟨*proof*⟩

lemma *card-mono*:
assumes *finite* B **and** $A \subseteq B$
shows *card* $A \leq$ *card* B
⟨*proof*⟩

lemma *card-seteq*: *finite* $B \Longrightarrow (\bigwedge A.\ A \subseteq B \Longrightarrow$ *card* $B \leq$ *card* $A \Longrightarrow A = B)$
⟨*proof*⟩

lemma *psubset-card-mono*: *finite* $B \Longrightarrow A < B \Longrightarrow$ *card* $A <$ *card* B
⟨*proof*⟩

lemma *card-Un-Int*:
assumes *finite* A *finite* B
shows *card* A + *card* B = *card* $(A \cup B)$ + *card* $(A \cap B)$
⟨*proof*⟩

lemma *card-Un-disjoint*: *finite* $A \Longrightarrow$ *finite* $B \Longrightarrow A \cap B = \{\} \Longrightarrow$ *card* $(A \cup B)$ = *card* A + *card* B
⟨*proof*⟩

lemma *card-Un-le*: *card* $(A \cup B) \leq$ *card* A + *card* B
⟨*proof*⟩

lemma *card-Diff-subset*:
assumes *finite* B
and $B \subseteq A$
shows *card* $(A - B)$ = *card* A − *card* B
⟨*proof*⟩

lemma *card-Diff-subset-Int*:
assumes *finite* $(A \cap B)$
shows *card* $(A - B)$ = *card* A − *card* $(A \cap B)$
⟨*proof*⟩

lemma *diff-card-le-card-Diff*:
assumes *finite* B
shows *card* A − *card* $B \leq$ *card* $(A - B)$
⟨*proof*⟩

lemma *card-Diff1-less*: *finite* $A \Longrightarrow x \in A \Longrightarrow$ *card* $(A - \{x\}) <$ *card* A

⟨*proof*⟩

lemma *card-Diff2-less*: $finite\ A \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow card\ (A - \{x\} - \{y\}) < card\ A$
⟨*proof*⟩

lemma *card-Diff1-le*: $finite\ A \Longrightarrow card\ (A - \{x\}) \leq card\ A$
⟨*proof*⟩

lemma *card-psubset*: $finite\ B \Longrightarrow A \subseteq B \Longrightarrow card\ A < card\ B \Longrightarrow A < B$
⟨*proof*⟩

lemma *card-le-inj*:
assumes *fA*: $finite\ A$
and *fB*: $finite\ B$
and *c*: $card\ A \leq card\ B$
shows $\exists f.\ f\ `\ A \subseteq B \wedge inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *card-subset-eq*:
assumes *fB*: $finite\ B$
and *AB*: $A \subseteq B$
and *c*: $card\ A = card\ B$
shows $A = B$
⟨*proof*⟩

lemma *insert-partition*:
$x \notin F \Longrightarrow \forall c1 \in insert\ x\ F.\ \forall c2 \in insert\ x\ F.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\} \Longrightarrow x \cap \bigcup F = \{\}$
⟨*proof*⟩

lemma *finite-psubset-induct* [*consumes 1*, *case-names psubset*]:
assumes *finite*: $finite\ A$
and *major*: $\bigwedge A.\ finite\ A \Longrightarrow (\bigwedge B.\ B \subset A \Longrightarrow P\ B) \Longrightarrow P\ A$
shows $P\ A$
⟨*proof*⟩

lemma *finite-induct-select* [*consumes 1*, *case-names empty select*]:
assumes $finite\ S$
and $P\ \{\}$
and *select*: $\bigwedge T.\ T \subset S \Longrightarrow P\ T \Longrightarrow \exists s{\in}S - T.\ P\ (insert\ s\ T)$
shows $P\ S$
⟨*proof*⟩

lemma *remove-induct* [*case-names empty infinite remove*]:
assumes *empty*: $P\ (\{\} :: 'a\ set)$
and *infinite*: $\neg\ finite\ B \Longrightarrow P\ B$
and *remove*: $\bigwedge A.\ finite\ A \Longrightarrow A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow P\ (A - \{x\})) \Longrightarrow P\ A$

shows $P\ B$
⟨*proof*⟩

lemma *finite-remove-induct* [*consumes 1*, *case-names empty remove*]:
fixes P :: $'a\ set \Rightarrow bool$
assumes *finite* B
and $P\ \{\}$
and $\bigwedge A.\ finite\ A \Longrightarrow A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow P\ (A - \{x\}))$
$\Longrightarrow P\ A$
defines $B' \equiv B$
shows $P\ B'$
⟨*proof*⟩

Main cardinality theorem.

lemma *card-partition* [*rule-format*]:
$finite\ C \Longrightarrow finite\ (\bigcup C) \Longrightarrow (\forall\, c{\in}C.\ card\ c = k) \Longrightarrow$
$(\forall\, c1 \in C.\ \forall\, c2 \in C.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}) \Longrightarrow$
$k * card\ C = card\ (\bigcup C)$
⟨*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 $\exists\, b\ B.\ A = insert\ b\ B \wedge b \notin B \wedge card\ B = k \wedge (k = 0 \longrightarrow B = \{\})$
⟨*proof*⟩

lemma *card-Suc-eq*:
$card\ A = Suc\ k \longleftrightarrow$
$(\exists\, b\ B.\ A = insert\ b\ B \wedge b \notin B \wedge card\ B = k \wedge (k = 0 \longrightarrow B = \{\}))$
⟨*proof*⟩

lemma *card-1-singletonE*:
assumes $card\ A = 1$
obtains x **where** $A = \{x\}$
⟨*proof*⟩

lemma *is-singleton-altdef*: $is\text{-}singleton\ A \longleftrightarrow card\ A = 1$
⟨*proof*⟩

lemma *card-le-Suc-iff*:
$finite\ A \Longrightarrow Suc\ n \leq card\ A = (\exists\, a\ B.\ A = insert\ a\ B \wedge a \notin B \wedge n \leq card\ B$
$\wedge\ finite\ B)$
⟨*proof*⟩

lemma *finite-fun-UNIVD2*:
  assumes *fin*: $finite\ (UNIV :: ('a \Rightarrow 'b)\ set)$
  shows $finite\ (UNIV :: 'b\ set)$
⟨*proof*⟩

lemma *card-UNIV-unit* [*simp*]: $card\ (UNIV :: unit\ set) = 1$
  ⟨*proof*⟩

lemma *infinite-arbitrarily-large*:
  assumes $\neg\ finite\ A$
  shows $\exists B.\ finite\ B \wedge card\ B = n \wedge B \subseteq A$
⟨*proof*⟩

18.5.1 Cardinality of image

lemma *card-image-le*: $finite\ A \Longrightarrow card\ (f\ `\ A) \leq card\ A$
  ⟨*proof*⟩

lemma *card-image*: $inj\text{-}on\ f\ A \Longrightarrow card\ (f\ `\ A) = card\ A$
⟨*proof*⟩

lemma *bij-betw-same-card*: $bij\text{-}betw\ f\ A\ B \Longrightarrow card\ A = card\ B$
  ⟨*proof*⟩

lemma *endo-inj-surj*: $finite\ A \Longrightarrow f\ `\ A \subseteq A \Longrightarrow inj\text{-}on\ f\ A \Longrightarrow f\ `\ A = A$
  ⟨*proof*⟩

lemma *eq-card-imp-inj-on*:
  assumes $finite\ A\ card(f\ `\ A) = card\ A$
  shows $inj\text{-}on\ f\ A$
  ⟨*proof*⟩

lemma *inj-on-iff-eq-card*: $finite\ A \Longrightarrow inj\text{-}on\ f\ A \longleftrightarrow card\ (f\ `\ A) = card\ A$
  ⟨*proof*⟩

lemma *card-inj-on-le*:
  assumes $inj\text{-}on\ f\ A\ f\ `\ A \subseteq B\ finite\ B$
  shows $card\ A \leq card\ B$
⟨*proof*⟩

lemma *surj-card-le*: $finite\ A \Longrightarrow B \subseteq f\ `\ A \Longrightarrow card\ B \leq card\ A$
  ⟨*proof*⟩

lemma *card-bij-eq*:
  $inj\text{-}on\ f\ A \Longrightarrow f\ `\ A \subseteq B \Longrightarrow inj\text{-}on\ g\ B \Longrightarrow g\ `\ B \subseteq A \Longrightarrow finite\ A \Longrightarrow finite\ B$
    $\Longrightarrow card\ A = card\ B$
  ⟨*proof*⟩

lemma *bij-betw-finite*: $\textit{bij-betw}\ f\ A\ B \Longrightarrow \textit{finite}\ A \longleftrightarrow \textit{finite}\ B$
⟨*proof*⟩

lemma *inj-on-finite*: $\textit{inj-on}\ f\ A \Longrightarrow f\ \text{`}\ A \leq B \Longrightarrow \textit{finite}\ B \Longrightarrow \textit{finite}\ A$
⟨*proof*⟩

lemma *card-vimage-inj*: $\textit{inj}\ f \Longrightarrow A \subseteq \textit{range}\ f \Longrightarrow \textit{card}\ (f\ -\text{`}\ A) = \textit{card}\ A$
⟨*proof*⟩

18.5.2 Pigeonhole Principles

lemma *pigeonhole*: $\textit{card}\ A > \textit{card}\ (f\ \text{`}\ A) \Longrightarrow \neg\ \textit{inj-on}\ f\ A$
⟨*proof*⟩

lemma *pigeonhole-infinite*:
assumes $\neg\ \textit{finite}\ A$ **and** $\textit{finite}\ (f\text{`}A)$
shows $\exists\, a0{\in}A.\ \neg\ \textit{finite}\ \{a{\in}A.\ f\ a = f\ a0\}$
⟨*proof*⟩

lemma *pigeonhole-infinite-rel*:
assumes $\neg\ \textit{finite}\ A$
and $\textit{finite}\ B$
and $\forall\, a{\in}A.\ \exists\, b{\in}B.\ R\ a\ b$
shows $\exists\, b{\in}B.\ \neg\ \textit{finite}\ \{a{:}A.\ R\ a\ b\}$
⟨*proof*⟩

18.5.3 Cardinality of sums

lemma *card-Plus*:
assumes $\textit{finite}\ A\ \textit{finite}\ B$
shows $\textit{card}\ (A <\!+\!> B) = \textit{card}\ A + \textit{card}\ B$
⟨*proof*⟩

lemma *card-Plus-conv-if*:
$\textit{card}\ (A <\!+\!> B) = (\textit{if finite}\ A \wedge \textit{finite}\ B\ \textit{then card}\ A + \textit{card}\ B\ \textit{else}\ 0)$
⟨*proof*⟩

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma *dvd-partition*:
assumes f: $\textit{finite}\ (\bigcup C)$
and $\forall\, c{\in}C.\ k\ \textit{dvd card}\ c\ \forall\, c1{\in}C.\ \forall\, c2{\in}C.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}$
shows $k\ \textit{dvd card}\ (\bigcup C)$
⟨*proof*⟩

18.5.4 Relating injectivity and surjectivity

lemma *finite-surj-inj*:
assumes $\textit{finite}\ A\ A \subseteq f\ \text{`}\ A$
shows $\textit{inj-on}\ f\ A$
⟨*proof*⟩

lemma *finite-UNIV-surj-inj*: $finite(UNIV :: 'a\ set) \Longrightarrow surj\ f \Longrightarrow inj\ f$
  for $f :: 'a \Rightarrow 'a$
  ⟨*proof*⟩

lemma *finite-UNIV-inj-surj*: $finite(UNIV :: 'a\ set) \Longrightarrow inj\ f \Longrightarrow surj\ f$
  for $f :: 'a \Rightarrow 'a$
  ⟨*proof*⟩

corollary *infinite-UNIV-nat* [*iff*]: $\neg\ finite\ (UNIV :: nat\ set)$
⟨*proof*⟩

lemma *infinite-UNIV-char-0*: $\neg\ finite\ (UNIV :: 'a::semiring\text{-}char\text{-}0\ set)$
⟨*proof*⟩

hide-const (**open**) *Finite-Set.fold*

18.6 Infinite Sets

Some elementary facts about infinite sets, mostly by Stephan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation $infinite :: 'a\ set \Rightarrow bool$
  where $infinite\ S \equiv \neg\ finite\ S$

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-imp-nonempty*: $infinite\ S \Longrightarrow S \neq \{\}$
  ⟨*proof*⟩

lemma *infinite-remove*: $infinite\ S \Longrightarrow infinite\ (S - \{a\})$
  ⟨*proof*⟩

lemma *Diff-infinite-finite*:
  assumes $finite\ T\ infinite\ S$
  shows $infinite\ (S - T)$
  ⟨*proof*⟩

lemma *Un-infinite*: $infinite\ S \Longrightarrow infinite\ (S \cup T)$
  ⟨*proof*⟩

lemma *infinite-Un*: $infinite\ (S \cup T) \longleftrightarrow infinite\ S \lor infinite\ T$
  ⟨*proof*⟩

lemma *infinite-super*:
  assumes $S \subseteq T$
    and $infinite\ S$
  shows $infinite\ T$

⟨proof⟩

proposition *infinite-coinduct* [*consumes 1*, *case-names infinite*]:
assumes $X\ A$
and *step*: $\bigwedge A.\ X\ A \Longrightarrow \exists x{\in}A.\ X\ (A - \{x\}) \vee \mathit{infinite}\ (A - \{x\})$
shows *infinite* A
⟨proof⟩

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

lemma *inf-img-fin-dom'*:
assumes *img*: *finite* $(f\ `\ A)$
and *dom*: *infinite* A
shows $\exists\, y \in f\ `\ A.\ \mathit{infinite}\ (f\ -`\ \{y\} \cap A)$
⟨proof⟩

lemma *inf-img-fin-domE'*:
assumes *finite* $(f\ `\ A)$ **and** *infinite* A
obtains y **where** $y \in f`A$ **and** *infinite* $(f\ -`\ \{y\} \cap A)$
⟨proof⟩

lemma *inf-img-fin-dom*:
assumes *img*: *finite* $(f`A)$ **and** *dom*: *infinite* A
shows $\exists\, y \in f`A.\ \mathit{infinite}\ (f\ -`\ \{y\})$
⟨proof⟩

lemma *inf-img-fin-domE*:
assumes *finite* $(f`A)$ **and** *infinite* A
obtains y **where** $y \in f`A$ **and** *infinite* $(f\ -`\ \{y\})$
⟨proof⟩

proposition *finite-image-absD*: *finite* $(\mathit{abs}\ `\ S) \Longrightarrow$ *finite* S
for S :: *'a::linordered-ring set*
⟨proof⟩

end

19 Relations – as sets of pairs, and binary predicates

theory *Relation*
imports *Finite-Set*
begin

A preliminary: classical rules for reasoning on predicates

declare *predicate1I* [*Pure.intro*!, *intro*!]

declare *predicate1D* [*Pure.dest*, *dest*]
declare *predicate2I* [*Pure.intro*!, *intro*!]
declare *predicate2D* [*Pure.dest*, *dest*]
declare *bot1E* [*elim*!]
declare *bot2E* [*elim*!]
declare *top1I* [*intro*!]
declare *top2I* [*intro*!]
declare *inf1I* [*intro*!]
declare *inf2I* [*intro*!]
declare *inf1E* [*elim*!]
declare *inf2E* [*elim*!]
declare *sup1I1* [*intro?*]
declare *sup2I1* [*intro?*]
declare *sup1I2* [*intro?*]
declare *sup2I2* [*intro?*]
declare *sup1E* [*elim*!]
declare *sup2E* [*elim*!]
declare *sup1CI* [*intro*!]
declare *sup2CI* [*intro*!]
declare *Inf1-I* [*intro*!]
declare *INF1-I* [*intro*!]
declare *Inf2-I* [*intro*!]
declare *INF2-I* [*intro*!]
declare *Inf1-D* [*elim*]
declare *INF1-D* [*elim*]
declare *Inf2-D* [*elim*]
declare *INF2-D* [*elim*]
declare *Inf1-E* [*elim*]
declare *INF1-E* [*elim*]
declare *Inf2-E* [*elim*]
declare *INF2-E* [*elim*]
declare *Sup1-I* [*intro*]
declare *SUP1-I* [*intro*]
declare *Sup2-I* [*intro*]
declare *SUP2-I* [*intro*]
declare *Sup1-E* [*elim*!]
declare *SUP1-E* [*elim*!]
declare *Sup2-E* [*elim*!]
declare *SUP2-E* [*elim*!]

19.1 Fundamental

19.1.1 Relations as sets of pairs

type-synonym $'a\ rel = ('a \times 'a)\ set$

lemma *subrelI*: $(\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow (x,\ y) \in s) \Longrightarrow r \subseteq s$
— Version of *subsetI* for binary relations
⟨*proof*⟩

lemma *lfp-induct2*:
$(a, b) \in lfp\ f \Longrightarrow mono\ f \Longrightarrow$
$(\bigwedge a\ b.\ (a, b) \in f\ (lfp\ f \cap \{(x, y).\ P\ x\ y\}) \Longrightarrow P\ a\ b) \Longrightarrow P\ a\ b$
— Version of *lfp-induct* for binary relations
⟨*proof*⟩

19.1.2 Conversions between set and predicate relations

lemma *pred-equals-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) = (\lambda x.\ x \in S) \longleftrightarrow R = S$
⟨*proof*⟩

lemma *pred-equals-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) = (\lambda x\ y.\ (x, y) \in S) \longleftrightarrow R = S$
⟨*proof*⟩

lemma *pred-subset-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) \le (\lambda x.\ x \in S) \longleftrightarrow R \subseteq S$
⟨*proof*⟩

lemma *pred-subset-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) \le (\lambda x\ y.\ (x, y) \in S) \longleftrightarrow R \subseteq S$
⟨*proof*⟩

lemma *bot-empty-eq* [*pred-set-conv*]: $\bot = (\lambda x.\ x \in \{\})$
⟨*proof*⟩

lemma *bot-empty-eq2* [*pred-set-conv*]: $\bot = (\lambda x\ y.\ (x, y) \in \{\})$
⟨*proof*⟩

lemma *top-empty-eq* [*pred-set-conv*]: $\top = (\lambda x.\ x \in UNIV)$
⟨*proof*⟩

lemma *top-empty-eq2* [*pred-set-conv*]: $\top = (\lambda x\ y.\ (x, y) \in UNIV)$
⟨*proof*⟩

lemma *inf-Int-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) \sqcap (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cap S)$
⟨*proof*⟩

lemma *inf-Int-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) \sqcap (\lambda x\ y.\ (x, y) \in S) = (\lambda x\ y.\ (x, y) \in R \cap S)$
⟨*proof*⟩

lemma *sup-Un-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) \sqcup (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
⟨*proof*⟩

lemma *sup-Un-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) \sqcup (\lambda x\ y.\ (x, y) \in S) = (\lambda x\ y.\ (x, y) \in R \cup S)$
⟨*proof*⟩

lemma *INF-INT-eq* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (\bigcap i{\in}S.\ r$

i))
⟨*proof*⟩

lemma *INF-INT-eq2* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (\bigcap i{\in}S.\ r\ i))$
⟨*proof*⟩

lemma *SUP-UN-eq* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (\bigcup i{\in}S.\ r\ i))$
⟨*proof*⟩

lemma *SUP-UN-eq2* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (\bigcup i{\in}S.\ r\ i))$
⟨*proof*⟩

lemma *Inf-INT-eq* [*pred-set-conv*]: $\bigsqcap S = (\lambda x.\ x \in INTER\ S\ Collect)$
⟨*proof*⟩

lemma *INF-Int-eq* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x.\ x \in i)) = (\lambda x.\ x \in \bigcap S)$
⟨*proof*⟩

lemma *Inf-INT-eq2* [*pred-set-conv*]: $\bigsqcap S = (\lambda x\ y.\ (x,\ y) \in INTER\ (case\text{-}prod\ `\ S)\ Collect)$
⟨*proof*⟩

lemma *INF-Int-eq2* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in i)) = (\lambda x\ y.\ (x,\ y) \in \bigcap S)$
⟨*proof*⟩

lemma *Sup-SUP-eq* [*pred-set-conv*]: $\bigsqcup S = (\lambda x.\ x \in UNION\ S\ Collect)$
⟨*proof*⟩

lemma *SUP-Sup-eq* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x.\ x \in i)) = (\lambda x.\ x \in \bigcup S)$
⟨*proof*⟩

lemma *Sup-SUP-eq2* [*pred-set-conv*]: $\bigsqcup S = (\lambda x\ y.\ (x,\ y) \in UNION\ (case\text{-}prod\ `\ S)\ Collect)$
⟨*proof*⟩

lemma *SUP-Sup-eq2* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in i)) = (\lambda x\ y.\ (x,\ y) \in \bigcup S)$
⟨*proof*⟩

19.2 Properties of relations

19.2.1 Reflexivity

definition *refl-on* :: $'a\ set \Rightarrow 'a\ rel \Rightarrow bool$
where *refl-on* $A\ r \longleftrightarrow r \subseteq A \times A \wedge (\forall x{\in}A.\ (x,\ x) \in r)$

abbreviation *refl* :: *'a rel* ⇒ *bool* — reflexivity over a type
where *refl* ≡ *refl-on UNIV*

definition *reflp* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *reflp r* ⟷ (∀ *x*. *r x x*)

lemma *reflp-refl-eq* [*pred-set-conv*]: *reflp* (λ*x y*. (*x*, *y*) ∈ *r*) ⟷ *refl r*
⟨*proof*⟩

lemma *refl-onI* [*intro?*]: *r* ⊆ *A* × *A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ (*x*, *x*) ∈ *r*) ⟹ *refl-on A r*
⟨*proof*⟩

lemma *refl-onD*: *refl-on A r* ⟹ *a* ∈ *A* ⟹ (*a*, *a*) ∈ *r*
⟨*proof*⟩

lemma *refl-onD1*: *refl-on A r* ⟹ (*x*, *y*) ∈ *r* ⟹ *x* ∈ *A*
⟨*proof*⟩

lemma *refl-onD2*: *refl-on A r* ⟹ (*x*, *y*) ∈ *r* ⟹ *y* ∈ *A*
⟨*proof*⟩

lemma *reflpI* [*intro?*]: (⋀*x*. *r x x*) ⟹ *reflp r*
⟨*proof*⟩

lemma *reflpE*:
assumes *reflp r*
obtains *r x x*
⟨*proof*⟩

lemma *reflpD* [*dest?*]:
assumes *reflp r*
shows *r x x*
⟨*proof*⟩

lemma *refl-on-Int*: *refl-on A r* ⟹ *refl-on B s* ⟹ *refl-on* (*A* ∩ *B*) (*r* ∩ *s*)
⟨*proof*⟩

lemma *reflp-inf*: *reflp r* ⟹ *reflp s* ⟹ *reflp* (*r* ⊓ *s*)
⟨*proof*⟩

lemma *refl-on-Un*: *refl-on A r* ⟹ *refl-on B s* ⟹ *refl-on* (*A* ∪ *B*) (*r* ∪ *s*)
⟨*proof*⟩

lemma *reflp-sup*: *reflp r* ⟹ *reflp s* ⟹ *reflp* (*r* ⊔ *s*)
⟨*proof*⟩

lemma *refl-on-INTER*: ∀ *x*∈*S*. *refl-on* (*A x*) (*r x*) ⟹ *refl-on* (*INTER S A*) (*INTER S r*)

⟨*proof*⟩

lemma *refl-on-UNION*: $\forall x{\in}S.\ \mathit{refl\text{-}on}\ (A\ x)\ (r\ x) \Longrightarrow \mathit{refl\text{-}on}\ (\mathit{UNION}\ S\ A)\ (\mathit{UNION}\ S\ r)$
⟨*proof*⟩

lemma *refl-on-empty* [*simp*]: $\mathit{refl\text{-}on}\ \{\}\ \{\}$
⟨*proof*⟩

lemma *refl-on-singleton* [*simp*]: $\mathit{refl\text{-}on}\ \{x\}\ \{(x,\ x)\}$
⟨*proof*⟩

lemma *refl-on-def'* [*nitpick-unfold*, *code*]:
$\mathit{refl\text{-}on}\ A\ r \longleftrightarrow (\forall (x,\ y) \in r.\ x \in A \wedge y \in A) \wedge (\forall x \in A.\ (x,\ x) \in r)$
⟨*proof*⟩

lemma *reflp-equality* [*simp*]: $\mathit{reflp}\ \mathit{op}\ =$
⟨*proof*⟩

lemma *reflp-mono*: $\mathit{reflp}\ R \Longrightarrow (\bigwedge x\ y.\ R\ x\ y \longrightarrow Q\ x\ y) \Longrightarrow \mathit{reflp}\ Q$
⟨*proof*⟩

19.2.2 Irreflexivity

definition $\mathit{irrefl} :: {'a}\ \mathit{rel} \Rightarrow \mathit{bool}$
where $\mathit{irrefl}\ r \longleftrightarrow (\forall a.\ (a,\ a) \notin r)$

definition $\mathit{irreflp} :: ({'a} \Rightarrow {'a} \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
where $\mathit{irreflp}\ R \longleftrightarrow (\forall a.\ \neg\ R\ a\ a)$

lemma *irreflp-irrefl-eq* [*pred-set-conv*]: $\mathit{irreflp}\ (\lambda a\ b.\ (a,\ b) \in R) \longleftrightarrow \mathit{irrefl}\ R$
⟨*proof*⟩

lemma *irreflI* [*intro?*]: $(\bigwedge a.\ (a,\ a) \notin R) \Longrightarrow \mathit{irrefl}\ R$
⟨*proof*⟩

lemma *irreflpI* [*intro?*]: $(\bigwedge a.\ \neg\ R\ a\ a) \Longrightarrow \mathit{irreflp}\ R$
⟨*proof*⟩

lemma *irrefl-distinct* [*code*]: $\mathit{irrefl}\ r \longleftrightarrow (\forall (a,\ b) \in r.\ a \neq b)$
⟨*proof*⟩

19.2.3 Asymmetry

inductive $\mathit{asym} :: {'a}\ \mathit{rel} \Rightarrow \mathit{bool}$
where *asymI*: $\mathit{irrefl}\ R \Longrightarrow (\bigwedge a\ b.\ (a,\ b) \in R \Longrightarrow (b,\ a) \notin R) \Longrightarrow \mathit{asym}\ R$

inductive $\mathit{asymp} :: ({'a} \Rightarrow {'a} \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
where *asympI*: $\mathit{irreflp}\ R \Longrightarrow (\bigwedge a\ b.\ R\ a\ b \Longrightarrow \neg\ R\ b\ a) \Longrightarrow \mathit{asymp}\ R$

lemma *asymp-asym-eq* [*pred-set-conv*]: $asymp\ (\lambda a\ b.\ (a,\ b) \in R) \longleftrightarrow asym\ R$
⟨*proof*⟩

19.2.4 Symmetry

definition $sym :: {}'a\ rel \Rightarrow bool$
where $sym\ r \longleftrightarrow (\forall x\ y.\ (x,\ y) \in r \longrightarrow (y,\ x) \in r)$

definition $symp :: ({}'a \Rightarrow {}'a \Rightarrow bool) \Rightarrow bool$
where $symp\ r \longleftrightarrow (\forall x\ y.\ r\ x\ y \longrightarrow r\ y\ x)$

lemma *symp-sym-eq* [*pred-set-conv*]: $symp\ (\lambda x\ y.\ (x,\ y) \in r) \longleftrightarrow sym\ r$
⟨*proof*⟩

lemma *symI* [*intro?*]: $(\bigwedge a\ b.\ (a,\ b) \in r \Longrightarrow (b,\ a) \in r) \Longrightarrow sym\ r$
⟨*proof*⟩

lemma *sympI* [*intro?*]: $(\bigwedge a\ b.\ r\ a\ b \Longrightarrow r\ b\ a) \Longrightarrow symp\ r$
⟨*proof*⟩

lemma *symE*:
assumes $sym\ r$ **and** $(b,\ a) \in r$
obtains $(a,\ b) \in r$
⟨*proof*⟩

lemma *sympE*:
assumes $symp\ r$ **and** $r\ b\ a$
obtains $r\ a\ b$
⟨*proof*⟩

lemma *symD* [*dest?*]:
assumes $sym\ r$ **and** $(b,\ a) \in r$
shows $(a,\ b) \in r$
⟨*proof*⟩

lemma *sympD* [*dest?*]:
assumes $symp\ r$ **and** $r\ b\ a$
shows $r\ a\ b$
⟨*proof*⟩

lemma *sym-Int*: $sym\ r \Longrightarrow sym\ s \Longrightarrow sym\ (r \cap s)$
⟨*proof*⟩

lemma *symp-inf*: $symp\ r \Longrightarrow symp\ s \Longrightarrow symp\ (r \sqcap s)$
⟨*proof*⟩

lemma *sym-Un*: $sym\ r \Longrightarrow sym\ s \Longrightarrow sym\ (r \cup s)$
⟨*proof*⟩

lemma *symp-sup*: $symp\ r \Longrightarrow symp\ s \Longrightarrow symp\ (r \sqcup s)$
⟨*proof*⟩

lemma *sym-INTER*: $\forall\, x{\in}S.\ sym\ (r\ x) \Longrightarrow sym\ (INTER\ S\ r)$
⟨*proof*⟩

lemma *symp-INF*: $\forall\, x{\in}S.\ symp\ (r\ x) \Longrightarrow symp\ (INFIMUM\ S\ r)$
⟨*proof*⟩

lemma *sym-UNION*: $\forall\, x{\in}S.\ sym\ (r\ x) \Longrightarrow sym\ (UNION\ S\ r)$
⟨*proof*⟩

lemma *symp-SUP*: $\forall\, x{\in}S.\ symp\ (r\ x) \Longrightarrow symp\ (SUPREMUM\ S\ r)$
⟨*proof*⟩

19.2.5 Antisymmetry

definition *antisym* :: $'a\ rel \Rightarrow bool$
where $antisym\ r \longleftrightarrow (\forall\, x\ y.\ (x,\ y) \in r \longrightarrow (y,\ x) \in r \longrightarrow x = y)$

definition *antisymp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $antisymp\ r \longleftrightarrow (\forall\, x\ y.\ r\ x\ y \longrightarrow r\ y\ x \longrightarrow x = y)$

lemma *antisymp-antisym-eq* [*pred-set-conv*]: $antisymp\ (\lambda x\ y.\ (x,\ y) \in r) \longleftrightarrow antisym\ r$
⟨*proof*⟩

lemma *antisymI* [*intro?*]:
$(\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow (y,\ x) \in r \Longrightarrow x = y) \Longrightarrow antisym\ r$
⟨*proof*⟩

lemma *antisympI* [*intro?*]:
$(\bigwedge x\ y.\ r\ x\ y \Longrightarrow r\ y\ x \Longrightarrow x = y) \Longrightarrow antisymp\ r$
⟨*proof*⟩

lemma *antisymD* [*dest?*]:
$antisym\ r \Longrightarrow (a,\ b) \in r \Longrightarrow (b,\ a) \in r \Longrightarrow a = b$
⟨*proof*⟩

lemma *antisympD* [*dest?*]:
$antisymp\ r \Longrightarrow r\ a\ b \Longrightarrow r\ b\ a \Longrightarrow a = b$
⟨*proof*⟩

lemma *antisym-subset*:
$r \subseteq s \Longrightarrow antisym\ s \Longrightarrow antisym\ r$
⟨*proof*⟩

lemma *antisymp-less-eq*:
$r \leq s \Longrightarrow antisymp\ s \Longrightarrow antisymp\ r$

⟨*proof*⟩

lemma *antisym-empty* [*simp*]:
antisym {}
⟨*proof*⟩

lemma *antisym-bot* [*simp*]:
antisymp $\perp$
⟨*proof*⟩

lemma *antisymp-equality* [*simp*]:
antisymp HOL.eq
⟨*proof*⟩

lemma *antisym-singleton* [*simp*]:
antisym $\{x\}$
⟨*proof*⟩

19.2.6 Transitivity

definition *trans* :: $'a$ *rel* $\Rightarrow$ *bool*
where *trans* $r \longleftrightarrow (\forall x\ y\ z.\ (x, y) \in r \longrightarrow (y, z) \in r \longrightarrow (x, z) \in r)$

definition *transp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where *transp* $r \longleftrightarrow (\forall x\ y\ z.\ r\ x\ y \longrightarrow r\ y\ z \longrightarrow r\ x\ z)$

lemma *transp-trans-eq* [*pred-set-conv*]: *transp* $(\lambda x\ y.\ (x, y) \in r) \longleftrightarrow$ *trans* r
⟨*proof*⟩

lemma *transI* [*intro?*]: $(\bigwedge x\ y\ z.\ (x, y) \in r \Longrightarrow (y, z) \in r \Longrightarrow (x, z) \in r) \Longrightarrow$ *trans* r
⟨*proof*⟩

lemma *transpI* [*intro?*]: $(\bigwedge x\ y\ z.\ r\ x\ y \Longrightarrow r\ y\ z \Longrightarrow r\ x\ z) \Longrightarrow$ *transp* r
⟨*proof*⟩

lemma *transE*:
assumes *trans* r **and** $(x, y) \in r$ **and** $(y, z) \in r$
obtains $(x, z) \in r$
⟨*proof*⟩

lemma *transpE*:
assumes *transp* r **and** $r\ x\ y$ **and** $r\ y\ z$
obtains $r\ x\ z$
⟨*proof*⟩

lemma *transD* [*dest?*]:
assumes *trans* r **and** $(x, y) \in r$ **and** $(y, z) \in r$
shows $(x, z) \in r$

⟨proof⟩

lemma *transpD* [*dest?*]:
assumes *transp r* **and** *r x y* **and** *r y z*
shows *r x z*
⟨proof⟩

lemma *trans-Int*: $trans\ r \Longrightarrow trans\ s \Longrightarrow trans\ (r \cap s)$
⟨proof⟩

lemma *transp-inf*: $transp\ r \Longrightarrow transp\ s \Longrightarrow transp\ (r \sqcap s)$
⟨proof⟩

lemma *trans-INTER*: $\forall x{\in}S.\ trans\ (r\ x) \Longrightarrow trans\ (INTER\ S\ r)$
⟨proof⟩

lemma *transp-INF*: $\forall x{\in}S.\ transp\ (r\ x) \Longrightarrow transp\ (INFIMUM\ S\ r)$
⟨proof⟩

lemma *trans-join* [*code*]: $trans\ r \longleftrightarrow (\forall (x,\ y1) \in r.\ \forall (y2,\ z) \in r.\ y1 = y2 \longrightarrow (x,\ z) \in r)$
⟨proof⟩

lemma *transp-trans*: $transp\ r \longleftrightarrow trans\ \{(x,\ y).\ r\ x\ y\}$
⟨proof⟩

lemma *transp-equality* [*simp*]: *transp op* =
⟨proof⟩

lemma *trans-empty* [*simp*]: $trans\ \{\}$
⟨proof⟩

lemma *transp-empty* [*simp*]: $transp\ (\lambda x\ y.\ False)$
⟨proof⟩

lemma *trans-singleton* [*simp*]: $trans\ \{(a,\ a)\}$
⟨proof⟩

lemma *transp-singleton* [*simp*]: $transp\ (\lambda x\ y.\ x = a \wedge y = a)$
⟨proof⟩

context *preorder*
begin

lemma *transp-le*[*simp*]: $transp\ (op \le)$
⟨proof⟩

lemma *transp-less*[*simp*]: $transp\ (op <)$
⟨proof⟩

lemma *transp-ge*[*simp*]: $transp\ (op \geq)$
⟨*proof*⟩

lemma *transp-gr*[*simp*]: $transp\ (op >)$
⟨*proof*⟩

end

19.2.7 Totality

definition *total-on* :: $'a\ set \Rightarrow 'a\ rel \Rightarrow bool$
where $total\text{-}on\ A\ r \longleftrightarrow (\forall x{\in}A.\ \forall y{\in}A.\ x \neq y \longrightarrow (x,\ y) \in r \lor (y,\ x) \in r)$

lemma *total-onI* [*intro?*]:
$(\bigwedge x\ y.\ [\![x \in A;\ y \in A;\ x \neq y]\!] \Longrightarrow (x,\ y) \in r \lor (y,\ x) \in r) \Longrightarrow total\text{-}on\ A\ r$
⟨*proof*⟩

abbreviation $total \equiv total\text{-}on\ UNIV$

lemma *total-on-empty* [*simp*]: $total\text{-}on\ \{\}\ r$
⟨*proof*⟩

lemma *total-on-singleton* [*simp*]: $total\text{-}on\ \{x\}\ \{(x,\ x)\}$
⟨*proof*⟩

19.2.8 Single valued relations

definition *single-valued* :: $('a \times 'b)\ set \Rightarrow bool$
where $single\text{-}valued\ r \longleftrightarrow (\forall x\ y.\ (x,\ y) \in r \longrightarrow (\forall z.\ (x,\ z) \in r \longrightarrow y = z))$

definition *single-valuedp* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where $single\text{-}valuedp\ r \longleftrightarrow (\forall x\ y.\ r\ x\ y \longrightarrow (\forall z.\ r\ x\ z \longrightarrow y = z))$

lemma *single-valuedp-single-valued-eq* [*pred-set-conv*]:
$single\text{-}valuedp\ (\lambda x\ y.\ (x,\ y) \in r) \longleftrightarrow single\text{-}valued\ r$
⟨*proof*⟩

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

lemma *single-valuedpI*:
$(\bigwedge x\ y.\ r\ x\ y \Longrightarrow (\bigwedge z.\ r\ x\ z \Longrightarrow y = z)) \Longrightarrow single\text{-}valuedp\ r$
⟨*proof*⟩

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

lemma *single-valuedpD*:
$single\text{-}valuedp\ r \Longrightarrow r\ x\ y \Longrightarrow r\ x\ z \Longrightarrow y = z$
$\langle proof \rangle$

lemma *single-valued-empty* [*simp*]:
$single\text{-}valued\ \{\}$
$\langle proof \rangle$

lemma *single-valuedp-bot* [*simp*]:
$single\text{-}valuedp\ \bot$
$\langle proof \rangle$

lemma *single-valued-subset*:
$r \subseteq s \Longrightarrow single\text{-}valued\ s \Longrightarrow single\text{-}valued\ r$
$\langle proof \rangle$

lemma *single-valuedp-less-eq*:
$r \leq s \Longrightarrow single\text{-}valuedp\ s \Longrightarrow single\text{-}valuedp\ r$
$\langle proof \rangle$

19.3 Relation operations

19.3.1 The identity relation

definition $Id :: 'a\ rel$
where [*code del*]: $Id = \{p.\ \exists x.\ p = (x, x)\}$

lemma *IdI* [*intro*]: $(a, a) \in Id$
$\langle proof \rangle$

lemma *IdE* [*elim*!]: $p \in Id \Longrightarrow (\bigwedge x.\ p = (x, x) \Longrightarrow P) \Longrightarrow P$
$\langle proof \rangle$

lemma *pair-in-Id-conv* [*iff*]: $(a, b) \in Id \longleftrightarrow a = b$
$\langle proof \rangle$

lemma *refl-Id*: *refl Id*
$\langle proof \rangle$

lemma *antisym-Id*: *antisym Id*
— A strange result, since *Id* is also symmetric.
$\langle proof \rangle$

lemma *sym-Id*: *sym Id*
$\langle proof \rangle$

lemma *trans-Id*: *trans Id*
$\langle proof \rangle$

lemma *single-valued-Id* [*simp*]: *single-valued Id*

⟨*proof*⟩

lemma *irrefl-diff-Id* [*simp*]: *irrefl* $(r - Id)$
⟨*proof*⟩

lemma *trans-diff-Id*: *trans* $r \Longrightarrow$ *antisym* $r \Longrightarrow$ *trans* $(r - Id)$
⟨*proof*⟩

lemma *total-on-diff-Id* [*simp*]: *total-on* A $(r - Id) =$ *total-on* A r
⟨*proof*⟩

lemma *Id-fstsnd-eq*: $Id = \{x.\ fst\ x = snd\ x\}$
⟨*proof*⟩

19.3.2 Diagonal: identity over a set

definition *Id-on* :: $'a\ set \Rightarrow 'a\ rel$
where *Id-on* $A = (\bigcup x{\in}A.\ \{(x, x)\})$

lemma *Id-on-empty* [*simp*]: *Id-on* $\{\} = \{\}$
⟨*proof*⟩

lemma *Id-on-eqI*: $a = b \Longrightarrow a \in A \Longrightarrow (a, b) \in$ *Id-on* A
⟨*proof*⟩

lemma *Id-onI* [*intro*!]: $a \in A \Longrightarrow (a, a) \in$ *Id-on* A
⟨*proof*⟩

lemma *Id-onE* [*elim*!]: $c \in$ *Id-on* $A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow c = (x, x) \Longrightarrow P) \Longrightarrow P$
— The general elimination rule.
⟨*proof*⟩

lemma *Id-on-iff*: $(x, y) \in$ *Id-on* $A \longleftrightarrow x = y \wedge x \in A$
⟨*proof*⟩

lemma *Id-on-def'* [*nitpick-unfold*]: *Id-on* $\{x.\ A\ x\} =$ *Collect* $(\lambda(x, y).\ x = y \wedge A\ x)$
⟨*proof*⟩

lemma *Id-on-subset-Times*: *Id-on* $A \subseteq A \times A$
⟨*proof*⟩

lemma *refl-on-Id-on*: *refl-on* A (*Id-on* A)
⟨*proof*⟩

lemma *antisym-Id-on* [*simp*]: *antisym* (*Id-on* A)
⟨*proof*⟩

lemma *sym-Id-on* [*simp*]: *sym* (*Id-on* A)
⟨*proof*⟩

lemma *trans-Id-on* [*simp*]: *trans* (*Id-on* A)
⟨*proof*⟩

lemma *single-valued-Id-on* [*simp*]: *single-valued* (*Id-on* A)
⟨*proof*⟩

19.3.3 Composition

inductive-set *relcomp* :: $('a \times 'b)$ *set* $\Rightarrow$ $('b \times 'c)$ *set* $\Rightarrow$ $('a \times 'c)$ *set* (**infixr** O *75*)
for r :: $('a \times 'b)$ *set* **and** s :: $('b \times 'c)$ *set*
where *relcompI* [*intro*]: $(a, b) \in r \Longrightarrow (b, c) \in s \Longrightarrow (a, c) \in r\ O\ s$

notation *relcompp* (**infixr** *OO 75*)

lemmas *relcomppI* = *relcompp.intros*

For historic reasons, the elimination rules are not wholly corresponding. Feel free to consolidate this.

inductive-cases *relcompEpair*: $(a, c) \in r\ O\ s$
inductive-cases *relcomppE* [*elim*!]: $(r\ OO\ s)\ a\ c$

lemma *relcompE* [*elim*!]: $xz \in r\ O\ s \Longrightarrow$
$(\bigwedge x\ y\ z.\ xz = (x, z) \Longrightarrow (x, y) \in r \Longrightarrow (y, z) \in s \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

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

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

lemma *relcomp-empty1* [*simp*]: $\{\}\ O\ R = \{\}$
⟨*proof*⟩

lemma *relcompp-bot1* [*simp*]: $\bot\ OO\ R = \bot$
⟨*proof*⟩

lemma *relcomp-empty2* [*simp*]: $R\ O\ \{\} = \{\}$
⟨*proof*⟩

lemma *relcompp-bot2* [*simp*]: $R\ OO\ \bot = \bot$
⟨*proof*⟩

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

lemma *relcompp-assoc*: $(r\ OO\ s)\ OO\ t = r\ OO\ (s\ OO\ t)$
⟨*proof*⟩

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

lemma *transp-relcompp-less-eq*: $transp\ r \Longrightarrow r\ OO\ r \leq r$
⟨*proof*⟩

lemma *relcomp-mono*: $r' \subseteq r \Longrightarrow s' \subseteq s \Longrightarrow r'\ O\ s' \subseteq r\ O\ s$
⟨*proof*⟩

lemma *relcompp-mono*: $r' \leq r \Longrightarrow s' \leq s \Longrightarrow r'\ OO\ s' \leq r\ OO\ s$
⟨*proof*⟩

lemma *relcomp-subset-Sigma*: $r \subseteq A \times B \Longrightarrow s \subseteq B \times C \Longrightarrow r\ O\ s \subseteq A \times C$
⟨*proof*⟩

lemma *relcomp-distrib* [*simp*]: $R\ O\ (S \cup T) = (R\ O\ S) \cup (R\ O\ T)$
⟨*proof*⟩

lemma *relcompp-distrib* [*simp*]: $R\ OO\ (S \sqcup T) = R\ OO\ S \sqcup R\ OO\ T$
⟨*proof*⟩

lemma *relcomp-distrib2* [*simp*]: $(S \cup T)\ O\ R = (S\ O\ R) \cup (T\ O\ R)$
⟨*proof*⟩

lemma *relcompp-distrib2* [*simp*]: $(S \sqcup T)\ OO\ R = S\ OO\ R \sqcup T\ OO\ R$
⟨*proof*⟩

lemma *relcomp-UNION-distrib*: $s\ O\ UNION\ I\ r = (\bigcup i{\in}I.\ s\ O\ r\ i)$
⟨*proof*⟩

lemma *relcompp-SUP-distrib*: $s\ OO\ SUPREMUM\ I\ r = (\bigsqcup i{\in}I.\ s\ OO\ r\ i)$
⟨*proof*⟩

lemma *relcomp-UNION-distrib2*: $UNION\ I\ r\ O\ s = (\bigcup i{\in}I.\ r\ i\ O\ s)$
⟨*proof*⟩

lemma *relcompp-SUP-distrib2*: $SUPREMUM\ I\ r\ OO\ s = (\bigsqcup i{\in}I.\ r\ i\ OO\ s)$
⟨*proof*⟩

lemma *single-valued-relcomp*: *single-valued* $r \Longrightarrow$ *single-valued* $s \Longrightarrow$ *single-valued* $(r\ O\ s)$
⟨*proof*⟩

lemma *relcomp-unfold*: $r\ O\ s = \{(x, z).\ \exists y.\ (x, y) \in r \wedge (y, z) \in s\}$
⟨*proof*⟩

lemma *relcompp-apply*: $(R\ OO\ S)\ a\ c \longleftrightarrow (\exists\, b.\ R\ a\ b \wedge S\ b\ c)$
⟨*proof*⟩

lemma *eq-OO*: $op = OO\ R = R$
⟨*proof*⟩

lemma *OO-eq*: $R\ OO\ op = = R$
⟨*proof*⟩

19.3.4 Converse

inductive-set *converse* :: $('a \times 'b)\ set \Rightarrow ('b \times 'a)\ set$ $((\text{-}^{-1})\ [1000]\ 999)$
for r :: $('a \times 'b)\ set$
where $(a,\ b) \in r \Longrightarrow (b,\ a) \in r^{-1}$

notation *conversep* $((\text{-}^{-1-1})\ [1000]\ 1000)$

notation (*ASCII*)
converse ((-^−*1*) [*1000*] *999*) **and**
conversep ((-^−−*1*) [*1000*] *1000*)

lemma *converseI* [*sym*]: $(a,\ b) \in r \Longrightarrow (b,\ a) \in r^{-1}$
⟨*proof*⟩

lemma *conversepI* : $r\ a\ b \Longrightarrow r^{-1-1}\ b\ a$
⟨*proof*⟩

lemma *converseD* [*sym*]: $(a,\ b) \in r^{-1} \Longrightarrow (b,\ a) \in r$
⟨*proof*⟩

lemma *conversepD* : $r^{-1-1}\ b\ a \Longrightarrow r\ a\ b$
⟨*proof*⟩

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

lemmas *conversepE* [*elim!*] = *conversep.cases*

lemma *converse-iff* [*iff*]: $(a,\ b) \in r^{-1} \longleftrightarrow (b,\ a) \in r$
⟨*proof*⟩

lemma *conversep-iff* [*iff*]: $r^{-1-1}\ a\ b = r\ b\ a$
⟨*proof*⟩

lemma *converse-converse* [*simp*]: $(r^{-1})^{-1} = r$
⟨*proof*⟩

lemma *conversep-conversep* [*simp*]: $(r^{-1-1})^{-1-1} = r$
⟨*proof*⟩

lemma *converse-empty*[*simp*]: $\{\}^{-1} = \{\}$
⟨*proof*⟩

lemma *converse-UNIV*[*simp*]: $UNIV^{-1} = UNIV$
⟨*proof*⟩

lemma *converse-relcomp*: $(r\ O\ s)^{-1} = s^{-1}\ O\ r^{-1}$
⟨*proof*⟩

lemma *converse-relcompp*: $(r\ OO\ s)^{-1-1} = s^{-1-1}\ OO\ r^{-1-1}$
⟨*proof*⟩

lemma *converse-Int*: $(r \cap s)^{-1} = r^{-1} \cap s^{-1}$
⟨*proof*⟩

lemma *converse-meet*: $(r \sqcap s)^{-1-1} = r^{-1-1} \sqcap s^{-1-1}$
⟨*proof*⟩

lemma *converse-Un*: $(r \cup s)^{-1} = r^{-1} \cup s^{-1}$
⟨*proof*⟩

lemma *converse-join*: $(r \sqcup s)^{-1-1} = r^{-1-1} \sqcup s^{-1-1}$
⟨*proof*⟩

lemma *converse-INTER*: $(INTER\ S\ r)^{-1} = (INT\ x{:}S.\ (r\ x)^{-1})$
⟨*proof*⟩

lemma *converse-UNION*: $(UNION\ S\ r)^{-1} = (UN\ x{:}S.\ (r\ x)^{-1})$
⟨*proof*⟩

lemma *converse-mono*[*simp*]: $r^{-1} \subseteq s\ ^{-1} \longleftrightarrow r \subseteq s$
⟨*proof*⟩

lemma *conversep-mono*[*simp*]: $r^{-1-1} \leq s\ ^{-1-1} \longleftrightarrow r \leq s$
⟨*proof*⟩

lemma *converse-inject*[*simp*]: $r^{-1} = s\ ^{-1} \longleftrightarrow r = s$
⟨*proof*⟩

lemma *conversep-inject*[*simp*]: $r^{-1-1} = s\ ^{-1-1} \longleftrightarrow r = s$
⟨*proof*⟩

lemma *converse-subset-swap*: $r \subseteq s\ ^{-1} \longleftrightarrow r\ ^{-1} \subseteq s$
⟨*proof*⟩

lemma *conversep-le-swap*: $r \leq s^{-1-1} \longleftrightarrow r^{-1-1} \leq s$
⟨*proof*⟩

lemma *converse-Id* [*simp*]: $Id^{-1} = Id$
⟨*proof*⟩

lemma *converse-Id-on* [*simp*]: $(Id\text{-}on\ A)^{-1} = Id\text{-}on\ A$
⟨*proof*⟩

lemma *refl-on-converse* [*simp*]: *refl-on A* (*converse r*) = *refl-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*: $sym\ r \longleftrightarrow r^{-1} = r$
⟨*proof*⟩

lemma *sym-Un-converse*: $sym\ (r \cup r^{-1})$
⟨*proof*⟩

lemma *sym-Int-converse*: $sym\ (r \cap r^{-1})$
⟨*proof*⟩

lemma *total-on-converse* [*simp*]: $total\text{-}on\ A\ (r^{-1}) = total\text{-}on\ A\ r$
⟨*proof*⟩

lemma *finite-converse* [*iff*]: $finite\ (r^{-1}) = finite\ r$
⟨*proof*⟩

lemma *conversep-noteq* [*simp*]: $(op \neq)^{-1-1} = op \neq$
⟨*proof*⟩

lemma *conversep-eq* [*simp*]: $(op =)^{-1-1} = op =$
⟨*proof*⟩

lemma *converse-unfold* [*code*]: $r^{-1} = \{(y, x).\ (x, y) \in r\}$
⟨*proof*⟩

19.3.5 Domain, range and field

inductive-set *Domain* :: $('a \times 'b)\ set \Rightarrow 'a\ set$ **for** r :: $('a \times 'b)\ set$
where *DomainI* [*intro*]: $(a, b) \in r \Longrightarrow a \in Domain\ r$

lemmas *DomainPI* = *Domainp.DomainI*

inductive-cases *DomainE* [*elim*!]: $a \in Domain\ r$
inductive-cases *DomainpE* [*elim*!]: *Domainp r a*

inductive-set *Range* :: $('a \times 'b)\ set \Rightarrow 'b\ set$ **for** r :: $('a \times 'b)\ set$
where *RangeI* [*intro*]: $(a, b) \in r \Longrightarrow b \in Range\ r$

lemmas *RangePI* = *Rangep.RangeI*

inductive-cases *RangeE* [*elim*!]: $b \in Range\ r$
inductive-cases *RangepE* [*elim*!]: *Rangep r b*

definition *Field* :: $'a\ rel \Rightarrow 'a\ set$
where $Field\ r = Domain\ r \cup Range\ r$

lemma *FieldI1*: $(i, j) \in R \Longrightarrow i \in Field\ R$
⟨*proof*⟩

lemma *FieldI2*: $(i, j) \in R \Longrightarrow j \in Field\ R$
⟨*proof*⟩

lemma *Domain-fst* [*code*]: *Domain r* = *fst ‘ r*
⟨*proof*⟩

lemma *Range-snd* [*code*]: *Range r* = *snd ‘ r*
⟨*proof*⟩

lemma *fst-eq-Domain*: *fst ‘ R* = *Domain R*
⟨*proof*⟩

lemma *snd-eq-Range*: *snd ‘ R* = *Range R*
⟨*proof*⟩

lemma *range-fst* [*simp*]: *range fst* = *UNIV*
⟨*proof*⟩

lemma *range-snd* [*simp*]: *range snd* = *UNIV*
⟨*proof*⟩

lemma *Domain-empty* [*simp*]: *Domain* {} = {}
⟨*proof*⟩

lemma *Range-empty* [*simp*]: *Range* {} = {}
⟨*proof*⟩

lemma *Field-empty* [*simp*]: *Field* {} = {}
⟨*proof*⟩

lemma *Domain-empty-iff*: $Domain\ r = \{\} \longleftrightarrow r = \{\}$
⟨*proof*⟩

lemma *Range-empty-iff*: $Range\ r = \{\} \longleftrightarrow r = \{\}$
⟨*proof*⟩

lemma *Domain-insert* [*simp*]: $Domain\ (insert\ (a,\ b)\ r) = insert\ a\ (Domain\ r)$
⟨*proof*⟩

lemma *Range-insert* [*simp*]: $Range\ (insert\ (a,\ b)\ r) = insert\ b\ (Range\ r)$
⟨*proof*⟩

lemma *Field-insert* [*simp*]: $Field\ (insert\ (a,\ b)\ r) = \{a,\ b\} \cup Field\ r$
⟨*proof*⟩

lemma *Domain-iff*: $a \in Domain\ r \longleftrightarrow (\exists\, y.\ (a,\ y) \in r)$
⟨*proof*⟩

lemma *Range-iff*: $a \in Range\ r \longleftrightarrow (\exists\, y.\ (y,\ a) \in r)$
⟨*proof*⟩

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

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

lemma *Domain-Id-on* [*simp*]: $Domain\ (Id\text{-}on\ A) = A$
⟨*proof*⟩

lemma *Range-Id-on* [*simp*]: $Range\ (Id\text{-}on\ A) = A$
⟨*proof*⟩

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

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

lemma *Field-Un* [*simp*]: $Field\ (r \cup s) = Field\ r \cup Field\ s$
⟨*proof*⟩

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

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

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

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

lemma *Domain-Union*: $Domain\ (\bigcup S) = (\bigcup A{\in}S.\ Domain\ A)$
⟨*proof*⟩

lemma *Range-Union*: $Range\ (\bigcup S) = (\bigcup A{\in}S.\ Range\ A)$
⟨*proof*⟩

lemma *Field-Union* [*simp*]: $Field\ (\bigcup R) = \bigcup (Field\ `\ R)$
⟨*proof*⟩

lemma *Domain-converse* [*simp*]: $Domain\ (r^{-1}) = Range\ r$
⟨*proof*⟩

lemma *Range-converse* [*simp*]: $Range\ (r^{-1}) = Domain\ r$
⟨*proof*⟩

lemma *Field-converse* [*simp*]: $Field\ (r^{-1}) = Field\ r$
⟨*proof*⟩

lemma *Domain-Collect-case-prod* [*simp*]: $Domain\ \{(x,\ y).\ P\ x\ y\} = \{x.\ \exists\, y.\ P\ x\ y\}$
⟨*proof*⟩

lemma *Range-Collect-case-prod* [*simp*]: $Range\ \{(x,\ y).\ P\ x\ y\} = \{y.\ \exists\, x.\ P\ x\ y\}$
⟨*proof*⟩

lemma *finite-Domain*: $finite\ r \Longrightarrow finite\ (Domain\ r)$
⟨*proof*⟩

lemma *finite-Range*: $finite\ r \Longrightarrow finite\ (Range\ r)$
⟨*proof*⟩

lemma *finite-Field*: $finite\ r \Longrightarrow finite\ (Field\ r)$
⟨*proof*⟩

lemma *Domain-mono*: $r \subseteq s \Longrightarrow Domain\ r \subseteq Domain\ s$
⟨*proof*⟩

lemma *Range-mono*: $r \subseteq s \Longrightarrow Range\ r \subseteq Range\ s$
⟨*proof*⟩

lemma *mono-Field*: $r \subseteq s \Longrightarrow Field\ r \subseteq Field\ s$
⟨*proof*⟩

lemma *Domain-unfold*: $Domain\ r = \{x.\ \exists\, y.\ (x,\ y) \in r\}$
⟨*proof*⟩

lemma *Field-square* [*simp*]: $Field\ (x \times x) = x$
⟨*proof*⟩

19.3.6 Image of a set under a relation

definition *Image* :: $('a \times 'b)\ set \Rightarrow 'a\ set \Rightarrow 'b\ set$ (**infixr** *`` 90*)
where $r\ ``\ s = \{y.\ \exists\, x{\in}s.\ (x,\ y) \in r\}$

lemma *Image-iff*: $b \in r``A \longleftrightarrow (\exists\, x{\in}A.\ (x,\ b) \in r)$
⟨*proof*⟩

lemma *Image-singleton*: $r``\{a\} = \{b.\ (a,\ b) \in r\}$
⟨*proof*⟩

lemma *Image-singleton-iff* [*iff*]: $b \in r``\{a\} \longleftrightarrow (a,\ b) \in r$
⟨*proof*⟩

lemma *ImageI* [*intro*]: $(a,\ b) \in r \Longrightarrow a \in A \Longrightarrow b \in r``A$
⟨*proof*⟩

lemma *ImageE* [*elim!*]: $b \in r\ ``\ A \Longrightarrow (\bigwedge x.\ (x,\ b) \in r \Longrightarrow x \in A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *rev-ImageI*: $a \in A \Longrightarrow (a,\ b) \in r \Longrightarrow b \in r\ ``\ A$
— 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*]: $Id\text{-}on\ A\ ``\ B = A \cap B$
⟨*proof*⟩

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

lemma *Image-Int-eq*: $single\text{-}valued\ (converse\ R) \Longrightarrow R\ ``\ (A \cap B) = R\ ``\ A \cap R\ ``\ B$
⟨*proof*⟩

lemma *Image-Un*: $R\ ``\ (A \cup B) = R\ ``\ A \cup R\ ``\ B$
⟨*proof*⟩

lemma *Un-Image*: $(R \cup S)$ `` $A = R$ `` $A \cup S$ `` A
⟨*proof*⟩

lemma *Image-subset*: $r \subseteq A \times B \Longrightarrow r``C \subseteq B$
⟨*proof*⟩

lemma *Image-eq-UN*: $r``B = (\bigcup y\in B.\ r``\{y\})$
— NOT suitable for rewriting
⟨*proof*⟩

lemma *Image-mono*: $r' \subseteq r \Longrightarrow A' \subseteq A \Longrightarrow (r'$ `` $A') \subseteq (r$ `` $A)$
⟨*proof*⟩

lemma *Image-UN*: $(r$ `` $(UNION\ A\ B)) = (\bigcup x\in A.\ r$ `` $(B\ x))$
⟨*proof*⟩

lemma *UN-Image*: $(\bigcup i\in I.\ X\ i)$ `` $S = (\bigcup i\in I.\ X\ i$ `` $S)$
⟨*proof*⟩

lemma *Image-INT-subset*: $(r$ `` $INTER\ A\ B) \subseteq (\bigcap x\in A.\ r$ `` $(B\ x))$
⟨*proof*⟩

Converse inclusion requires some assumptions

lemma *Image-INT-eq*: *single-valued* $(r^{-1}) \Longrightarrow A \neq \{\} \Longrightarrow r$ `` $INTER\ A\ B = (\bigcap x\in A.\ r$ `` $B\ x)$
⟨*proof*⟩

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

lemma *Image-Collect-case-prod* [*simp*]: $\{(x,\ y).\ P\ x\ y\}$ `` $A = \{y.\ \exists x\in A.\ P\ x\ y\}$
⟨*proof*⟩

lemma *Sigma-Image*: $(SIGMA\ x{:}A.\ B\ x)$ `` $X = (\bigcup x\in X \cap A.\ B\ x)$
⟨*proof*⟩

lemma *relcomp-Image*: $(X\ O\ Y)$ `` $Z = Y$ `` $(X$ `` $Z)$
⟨*proof*⟩

19.3.7 Inverse image

definition *inv-image* :: $'b\ rel \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ rel$
where *inv-image* $r\ f = \{(x,\ y).\ (f\ x,\ f\ y) \in r\}$

definition *inv-imagep* :: $('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where *inv-imagep* $r\ f = (\lambda x\ y.\ r\ (f\ x)\ (f\ y))$

lemma [*pred-set-conv*]: *inv-imagep* $(\lambda x\ y.\ (x,\ y) \in r)\ f = (\lambda x\ y.\ (x,\ y) \in$ *inv-image* $r\ f)$

⟨*proof*⟩

lemma *sym-inv-image*: $sym\ r \Longrightarrow sym\ (inv\text{-}image\ r\ f)$
⟨*proof*⟩

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

lemma *in-inv-image*[*simp*]: $(x,\ y) \in inv\text{-}image\ r\ f \longleftrightarrow (f\ x,\ f\ y) \in r$
⟨*proof*⟩

lemma *converse-inv-image*[*simp*]: $(inv\text{-}image\ R\ f)^{-1} = inv\text{-}image\ (R^{-1})\ f$
⟨*proof*⟩

lemma *in-inv-imagep* [*simp*]: $inv\text{-}imagep\ r\ f\ x\ y = r\ (f\ x)\ (f\ y)$
⟨*proof*⟩

19.3.8 Powerset

definition $Powp :: ('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$
where $Powp\ A = (\lambda B.\ \forall\, x \in B.\ A\ x)$

lemma *Powp-Pow-eq* [*pred-set-conv*]: $Powp\ (\lambda x.\ x \in A) = (\lambda x.\ x \in Pow\ A)$
⟨*proof*⟩

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred*]

19.3.9 Expressing relation operations via *Finite-Set.fold*

lemma *Id-on-fold*:
assumes *finite A*
shows $Id\text{-}on\ A = Finite\text{-}Set.fold\ (\lambda x.\ Set.insert\ (Pair\ x\ x))\ \{\}\ A$
⟨*proof*⟩

lemma *comp-fun-commute-Image-fold*:
$comp\text{-}fun\text{-}commute\ (\lambda(x,y)\ A.\ if\ x \in S\ then\ Set.insert\ y\ A\ else\ A)$
⟨*proof*⟩

lemma *Image-fold*:
assumes *finite R*
shows $R\ ``\ S = Finite\text{-}Set.fold\ (\lambda(x,y)\ A.\ if\ x \in S\ then\ Set.insert\ y\ A\ else\ A)$
$\{\}\ R$
⟨*proof*⟩

lemma *insert-relcomp-union-fold*:
assumes *finite S*
shows $\{x\}\ O\ S \cup X = Finite\text{-}Set.fold\ (\lambda(w,z)\ A'.\ if\ snd\ x = w\ then\ Set.insert$
$(fst\ x,z)\ A'\ else\ A')\ X\ S$
⟨*proof*⟩

lemma *insert-relcomp-fold*:
assumes *finite S*
shows *Set.insert x R O S* =
Finite-Set.fold (λ(*w*,*z*) *A'*. *if snd x* = *w then Set.insert* (*fst x*,*z*) *A' else A'*) (*R O S*) *S*
⟨*proof*⟩

lemma *comp-fun-commute-relcomp-fold*:
assumes *finite S*
shows *comp-fun-commute* (λ(*x*,*y*) *A*.
Finite-Set.fold (λ(*w*,*z*) *A'*. *if y* = *w then Set.insert* (*x*,*z*) *A' else A'*) *A S*)
⟨*proof*⟩

lemma *relcomp-fold*:
assumes *finite R finite S*
shows *R O S* = *Finite-Set.fold*
(λ(*x*,*y*) *A*. *Finite-Set.fold* (λ(*w*,*z*) *A'*. *if y* = *w then Set.insert* (*x*,*z*) *A' else A'*) *A S*) {} *R*
⟨*proof*⟩

end

20 Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Relation*
begin

⟨*ML*⟩

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.

context notes [[*inductive-internals*]]
begin

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*]: $(a, a) \in r^*$
| *rtrancl-into-rtrancl* [*Pure.intro*]: $(a, b) \in r^* \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^*$

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*]: $(a, b) \in r \Longrightarrow (a, b) \in r^+$
| *trancl-into-trancl* [*Pure.intro*]: $(a, b) \in r^+ \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^+$

notation
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-$^{++}$) [*1000*] *1000*)

declare
rtrancl-def [*nitpick-unfold del*]
rtranclp-def [*nitpick-unfold del*]
trancl-def [*nitpick-unfold del*]
tranclp-def [*nitpick-unfold del*]

end

abbreviation *reflcl* :: $('a \times 'a)$ *set* $\Rightarrow$ $('a \times 'a)$ *set* ((-$^{=}$) [*1000*] *999*)
where $r^{=} \equiv r \cup Id$

abbreviation *reflclp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ ((-$^{==}$) [*1000*] *1000*)
where $r^{==} \equiv$ *sup r op* $=$

notation (*ASCII*)
rtrancl ((-ˆ∗) [*1000*] *999*) **and**
trancl ((-ˆ+) [*1000*] *999*) **and**
reflcl ((-ˆ=) [*1000*] *999*) **and**
rtranclp ((-ˆ∗∗) [*1000*] *1000*) **and**
tranclp ((-ˆ++) [*1000*] *1000*) **and**
reflclp ((-ˆ==) [*1000*] *1000*)

20.1 Reflexive closure

lemma *refl-reflcl*[*simp*]: *refl* $(r^{=})$
⟨*proof*⟩

lemma *antisym-reflcl*[*simp*]: *antisym* $(r^{=})$ = *antisym r*
⟨*proof*⟩

lemma *trans-reflclI*[*simp*]: *trans r* $\Longrightarrow$ *trans* $(r^{=})$
⟨*proof*⟩

lemma *reflclp-idemp* [*simp*]: $(P^{==})^{==} = P^{==}$
⟨*proof*⟩

20.2 Reflexive-transitive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: (*sup* $(\lambda x\ y.\ (x, y) \in r)$ *op* $=$) $=$ $(\lambda x\ y.\ (x, y) \in r \cup Id)$
⟨*proof*⟩

lemma *r-into-rtrancl* [*intro*]: $\bigwedge p.\ p \in r \Longrightarrow p \in r^*$
— *rtrancl* of r contains r
⟨*proof*⟩

lemma *r-into-rtranclp* [*intro*]: $r\ x\ y \Longrightarrow r^{**}\ x\ y$
— *rtrancl* of r contains r
⟨*proof*⟩

lemma *rtranclp-mono*: $r \leq s \Longrightarrow r^{**} \leq s^{**}$
— monotonicity of *rtrancl*
⟨*proof*⟩

lemma *mono-rtranclp*[*mono*]: $(\bigwedge a\ b.\ x\ a\ b \longrightarrow y\ a\ b) \Longrightarrow x^{**}\ a\ b \longrightarrow y^{**}\ a\ b$
⟨*proof*⟩

lemmas *rtrancl-mono* = *rtranclp-mono* [*to-set*]

theorem *rtranclp-induct* [*consumes 1*, *case-names base step*, *induct set*: *rtranclp*]:
assumes *a*: $r^{**}\ a\ b$
and *cases*: $P\ a \bigwedge y\ z.\ r^{**}\ a\ y \Longrightarrow r\ y\ z \Longrightarrow P\ y \Longrightarrow P\ z$
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^*)
⟨*proof*⟩

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* (r^*)
⟨*proof*⟩

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

lemma *rtranclp-trans*:
assumes $r^{**}\ x\ y$
and $r^{**}\ y\ z$
shows $r^{**}\ x\ z$
⟨*proof*⟩

lemma *rtranclE* [*cases set*: *rtrancl*]:
fixes $a\ b :: 'a$
assumes *major*: $(a,\ b) \in r^*$
obtains
(*base*) $a = b$

| (*step*) y **where** $(a,\ y) \in r^*$ **and** $(y,\ b) \in r$
— elimination of *rtrancl* – by induction on a special formula
⟨*proof*⟩

lemma *rtrancl-Int-subset*: $Id \subseteq s \Longrightarrow (r^* \cap s)\ O\ r \subseteq s \Longrightarrow r^* \subseteq s$
⟨*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*]: $(r^{**})^{**} = r^{**}$
⟨*proof*⟩

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [*to-set*]

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

lemma *rtrancl-subset-rtrancl*: $r \subseteq s^* \Longrightarrow r^* \subseteq s^*$
⟨*proof*⟩

lemma *rtranclp-subset*: $R \leq S \Longrightarrow S \leq R^{**} \Longrightarrow S^{**} = R^{**}$
⟨*proof*⟩

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(sup\ (R^{**})\ (S^{**}))^{**} = (sup\ R\ S)^{**}$
⟨*proof*⟩

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflclp* [*simp*]: $(R^{==})^{**} = R^{**}$
⟨*proof*⟩

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflclp* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)^* = r^*$
⟨*proof*⟩

lemma *rtranclp-r-diff-Id*: $(inf\ r\ op \neq)^{**} = r^{**}$
⟨*proof*⟩

theorem *rtranclp-converseD*:
assumes $(r^{-1-1})^{**}\ x\ y$
shows $r^{**}\ y\ x$
⟨*proof*⟩

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

theorem *rtranclp-converseI*:
assumes $r^{**}\ y\ x$
shows $(r^{-1-1})^{**}\ x\ y$
⟨*proof*⟩

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

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

lemma *sym-rtrancl*: $sym\ r \Longrightarrow sym\ (r^{*})$
⟨*proof*⟩

theorem *converse-rtranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: $r^{**}\ a\ b$
and *cases*: $P\ b\ \bigwedge y\ z.\ r\ y\ z \Longrightarrow r^{**}\ z\ b \Longrightarrow P\ z \Longrightarrow P\ y$
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*: $r^{**}\ x\ z$
and *cases*: $x = z \Longrightarrow P\ \bigwedge y.\ r\ x\ y \Longrightarrow r^{**}\ y\ z \Longrightarrow P$
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*: $r\ O\ r^{*} = r^{*}\ O\ r$
⟨*proof*⟩

lemma *rtrancl-unfold*: $r^{*} = Id \cup r^{*}\ O\ r$
⟨*proof*⟩

lemma *rtrancl-Un-separatorE*:
$(a, b) \in (P \cup Q)^* \Longrightarrow \forall x\ y.\ (a, x) \in P^* \longrightarrow (x, y) \in Q \longrightarrow x = y \Longrightarrow (a, b) \in P^*$
⟨*proof*⟩

lemma *rtrancl-Un-separator-converseE*:
$(a, b) \in (P \cup Q)^* \Longrightarrow \forall x\ y.\ (x, b) \in P^* \longrightarrow (y, x) \in Q \longrightarrow y = x \Longrightarrow (a, b) \in P^*$
⟨*proof*⟩

lemma *Image-closed-trancl*:
assumes r ‘‘ $X \subseteq X$
shows r^* ‘‘ $X = X$
⟨*proof*⟩

20.3 Transitive closure

lemma *trancl-mono*: $\bigwedge p.\ p \in r^+ \Longrightarrow r \subseteq s \Longrightarrow p \in s^+$
⟨*proof*⟩

lemma *r-into-trancl'*: $\bigwedge p.\ p \in r \Longrightarrow p \in r^+$
⟨*proof*⟩

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: $r^{++}\ a\ b \Longrightarrow r^{**}\ a\ b$
⟨*proof*⟩

lemmas *trancl-into-rtrancl* = *tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*:
assumes $r^{**}\ a\ b$
shows $r\ b\ c \Longrightarrow r^{++}\ a\ c$
⟨*proof*⟩

lemmas *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*: $r\ a\ b \Longrightarrow r^{**}\ b\ c \Longrightarrow r^{++}\ a\ c$
— 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*: $r^{++}\ a\ b$
and *cases*: $\bigwedge y.\ r\ a\ y \Longrightarrow P\ y\ \bigwedge y\ z.\ r^{++}\ a\ y \Longrightarrow r\ y\ z \Longrightarrow P\ y \Longrightarrow P\ z$
shows $P\ b$
⟨*proof*⟩

lemmas *trancl-induct* [*induct set*: *trancl*] = *tranclp-induct* [*to-set*]

lemmas *tranclp-induct2* =
tranclp-induct [*of* - (*ax*, *ay*) (*bx*, *by*), *split-rule*, *consumes 1*, *case-names base step*]

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

lemma *tranclp-trans-induct*:
assumes *major*: $r^{++}\ x\ y$
and *cases*: $\bigwedge x\ y.\ r\ x\ y \Longrightarrow P\ x\ y$ $\bigwedge x\ y\ z.\ r^{++}\ x\ y \Longrightarrow P\ x\ y \Longrightarrow r^{++}\ y\ z \Longrightarrow P\ y\ z \Longrightarrow P\ x\ z$
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 $(a,\ b) \in r^+$
obtains
(*base*) $(a,\ b) \in r$
| (*step*) c **where** $(a,\ c) \in r^+$ **and** $(c,\ b) \in r$
⟨*proof*⟩

lemma *trancl-Int-subset*: $r \subseteq s \Longrightarrow (r^+ \cap s)\ O\ r \subseteq s \Longrightarrow r^+ \subseteq s$
⟨*proof*⟩

lemma *trancl-unfold*: $r^+ = r \cup r^+\ O\ r$
⟨*proof*⟩

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* (r^+)
⟨*proof*⟩

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*]

lemma *tranclp-trans*:
assumes $r^{++}\ x\ y$
and $r^{++}\ y\ z$
shows $r^{++}\ x\ z$
⟨*proof*⟩

lemma *trancl-id* [*simp*]: *trans* $r \Longrightarrow r^+ = r$
⟨*proof*⟩

lemma *rtranclp-tranclp-tranclp*:
assumes r^{**} x y
shows $\bigwedge z.\ r^{++}\ y\ z \Longrightarrow r^{++}\ x\ z$
⟨*proof*⟩

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: $r\ a\ b \Longrightarrow r^{++}\ b\ c \Longrightarrow r^{++}\ a\ c$
⟨*proof*⟩

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *tranclp-converseI*: $(r^{++})^{-1-1}\ x\ y \Longrightarrow (r^{-1-1})^{++}\ x\ y$
⟨*proof*⟩

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*: $(r^{-1-1})^{++}\ x\ y \Longrightarrow (r^{++})^{-1-1}\ x\ y$
⟨*proof*⟩

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

lemma *tranclp-converse*: $(r^{-1-1})^{++} = (r^{++})^{-1-1}$
⟨*proof*⟩

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: *sym* $r \Longrightarrow$ *sym* (r^{+})
⟨*proof*⟩

lemma *converse-tranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: $r^{++}\ a\ b$
and *cases*: $\bigwedge y.\ r\ y\ b \Longrightarrow P\ y$ $\bigwedge y\ z.\ r\ y\ z \Longrightarrow r^{++}\ z\ b \Longrightarrow P\ z \Longrightarrow P\ y$
shows $P\ a$
⟨*proof*⟩

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

lemma *tranclpD*: $R^{++}\ x\ y \Longrightarrow \exists\, z.\ R\ x\ z \wedge R^{**}\ z\ y$
⟨*proof*⟩

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *converse-tranclpE*:
assumes *major*: *tranclp* $r\ x\ z$
and *base*: $r\ x\ z \Longrightarrow P$
and *step*: $\bigwedge y.\ r\ x\ y \Longrightarrow$ *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*: $r^{-1} \cap r^* = \{\} \Longrightarrow (x, x) \notin r^+$
⟨*proof*⟩

lemma *irrefl-trancl-rD*: $\forall x. (x, x) \notin r^+ \Longrightarrow (x, y) \in r \Longrightarrow x \neq y$
⟨*proof*⟩

lemma *trancl-subset-Sigma-aux*: $(a, b) \in r^* \Longrightarrow r \subseteq A \times A \Longrightarrow a = b \vee a \in A$
⟨*proof*⟩

lemma *trancl-subset-Sigma*: $r \subseteq A \times A \Longrightarrow r^+ \subseteq A \times A$
⟨*proof*⟩

lemma *reflclp-tranclp* [*simp*]: $(r^{++})^{==} = r^{**}$
⟨*proof*⟩

lemmas *reflcl-trancl* [*simp*] = *reflclp-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r^=)^+ = r^*$
⟨*proof*⟩

lemma *rtrancl-trancl-reflcl* [*code*]: $r^* = (r^+)^=$
⟨*proof*⟩

lemma *trancl-empty* [*simp*]: $\{\}^+ = \{\}$
⟨*proof*⟩

lemma *rtrancl-empty* [*simp*]: $\{\}^* = Id$
⟨*proof*⟩

lemma *rtranclpD*: $R^{**}\ a\ b \Longrightarrow a = b \vee a \neq b \wedge R^{++}\ a\ b$
⟨*proof*⟩

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*: $(x,y) \in R^* \longleftrightarrow x = y \vee x \neq y \wedge (x, y) \in R^+$
⟨*proof*⟩

lemma *trancl-unfold-right*: $r^+ = r^*\ O\ r$
⟨*proof*⟩

lemma *trancl-unfold-left*: $r^+ = r\ O\ r^*$
⟨*proof*⟩

lemma *trancl-insert*: $(insert\ (y,\ x)\ r)^+ = r^+ \cup \{(a,\ b).\ (a,\ y) \in r^* \wedge (x,\ b) \in r^*\}$
— primitive recursion for *trancl* over finite relations
⟨*proof*⟩

lemma *trancl-insert2*:
$(insert\ (a,\ b)\ r)^+ = r^+ \cup \{(x,\ y).\ ((x,\ a) \in r^+ \vee x = a) \wedge ((b,\ y) \in r^+ \vee y = b)\}$
⟨*proof*⟩

lemma *rtrancl-insert*: $(insert\ (a,b)\ r)^* = r^* \cup \{(x,\ y).\ (x,\ a) \in r^* \wedge (b,\ y) \in r^*\}$
⟨*proof*⟩

Simplifying nested closures

lemma *rtrancl-trancl-absorb*[*simp*]: $(R^*)^+ = R^*$
⟨*proof*⟩

lemma *trancl-rtrancl-absorb*[*simp*]: $(R^+)^* = R^*$
⟨*proof*⟩

lemma *rtrancl-reflcl-absorb*[*simp*]: $(R^*)^= = R^*$
⟨*proof*⟩

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: $Domain\ (R^*) = UNIV$
⟨*proof*⟩

lemma *Range-rtrancl* [*simp*]: $Range\ (R^*) = UNIV$
⟨*proof*⟩

lemma *rtrancl-Un-subset*: $(R^* \cup S^*) \subseteq (R \cup S)^*$
⟨*proof*⟩

lemma *in-rtrancl-UnI*: $x \in R^* \vee x \in S^* \Longrightarrow x \in (R \cup S)^*$
⟨*proof*⟩

lemma *trancl-domain* [*simp*]: $Domain\ (r^+) = Domain\ r$
⟨*proof*⟩

lemma *trancl-range* [*simp*]: $Range\ (r^+) = Range\ r$
⟨*proof*⟩

lemma *Not-Domain-rtrancl*: $x \notin Domain\ R \Longrightarrow (x,\ y) \in R^* \longleftrightarrow x = y$
⟨*proof*⟩

lemma *trancl-subset-Field2*: $r^+ \subseteq Field\ r \times Field\ r$
⟨*proof*⟩

lemma *finite-trancl*[*simp*]: $finite\ (r^+) = finite\ r$

⟨*proof*⟩

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
$single\text{-}valued\ r \Longrightarrow (x,\ y) \in r^* \Longrightarrow (x,\ z) \in r^* \Longrightarrow (y,\ z) \in r^* \lor (z,\ y) \in r^*$
⟨*proof*⟩

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

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

lemma *tranclp-rtranclp-tranclp*: $r^{++}\ a\ b \Longrightarrow r^{**}\ b\ c \Longrightarrow r^{++}\ a\ c$
⟨*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*]

20.4 The power operation on relations

R ^^ $n = R\ O\ \ldots\ O\ R$, the n-fold composition of R

overloading
$relpow \equiv compow :: nat \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$
$relpowp \equiv compow :: nat \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
begin

primrec $relpow :: nat \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$
where
relpow 0 R = Id
| *relpow* (*Suc n*) *R* = (*R* ^^ *n*) *O R*

primrec $relpowp :: nat \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
where
relpowp 0 R = HOL.eq
| *relpowp* (*Suc n*) *R* = (*R* ^^ *n*) *OO R*

end

lemma *relpowp-relpow-eq* [*pred-set-conv*]:
$(\lambda x\ y.\ (x,\ y) \in R)\ \hat{}\hat{}\ n = (\lambda x\ y.\ (x,\ y) \in R\ \hat{}\hat{}\ n)$ **for** $R :: 'a\ rel$
⟨*proof*⟩

For code generation:

definition *relpow* :: $nat \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$
where *relpow-code-def* [*code-abbrev*]: *relpow = compow*

definition *relpowp* :: $nat \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
where *relpowp-code-def* [*code-abbrev*]: *relpowp = compow*

lemma [*code*]:
relpow (*Suc n*) R = (*relpow n R*) O R
relpow 0 R = Id
⟨*proof*⟩

lemma [*code*]:
relpowp (*Suc n*) $R = (R\ \hat{}\hat{}\ n)\ OO\ R$
relpowp 0 R = HOL.eq
⟨*proof*⟩

hide-const (**open**) *relpow*
hide-const (**open**) *relpowp*

lemma *relpow-1* [*simp*]: $R\ \hat{}\hat{}\ 1 = R$
for $R :: ('a \times 'a)\ set$
⟨*proof*⟩

lemma *relpowp-1* [*simp*]: $P\ \hat{}\hat{}\ 1 = P$
for $P :: 'a \Rightarrow 'a \Rightarrow bool$
⟨*proof*⟩

lemma *relpow-0-I*: $(x,\ x) \in R\ \hat{}\hat{}\ 0$
⟨*proof*⟩

lemma *relpowp-0-I*: $(P\ \hat{}\hat{}\ 0)\ x\ x$
⟨*proof*⟩

lemma *relpow-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 *relpowp-Suc-I*: $(P\ \hat{}\hat{}\ n)\ x\ y \Longrightarrow P\ y\ z \Longrightarrow (P\ \hat{}\hat{}\ Suc\ n)\ x\ z$
⟨*proof*⟩

lemma *relpow-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 *relpowp-Suc-I2*: $P\ x\ y \Longrightarrow (P\ \hat{}\hat{}\ n)\ y\ z \Longrightarrow (P\ \hat{}\hat{}\ Suc\ n)\ x\ z$
⟨*proof*⟩

lemma *relpow-0-E*: $(x,\ y) \in R\ \hat{}\hat{}\ 0 \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *relpowp-0-E*: $(P\ \hat{}\hat{}\ 0)\ x\ y \Longrightarrow (x = y \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *relpow-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 *relpowp-Suc-E*: $(P\ \hat{}\hat{}\ Suc\ n)\ x\ z \Longrightarrow (\bigwedge y.\ (P\ \hat{}\hat{}\ n)\ x\ y \Longrightarrow P\ y\ z \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *relpow-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 *relpowp-E*:
$(P\ \hat{}\hat{}\ n)\ x\ z \Longrightarrow$
$(n = 0 \Longrightarrow x = z \Longrightarrow Q) \Longrightarrow$
$(\bigwedge y\ m.\ n = Suc\ m \Longrightarrow (P\ \hat{}\hat{}\ m)\ x\ y \Longrightarrow P\ y\ z \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *relpow-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 *relpowp-Suc-D2*: $(P\ \hat{}\hat{}\ Suc\ n)\ x\ z \Longrightarrow \exists\, y.\ P\ x\ y \wedge (P\ \hat{}\hat{}\ n)\ y\ z$
⟨*proof*⟩

lemma *relpow-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 *relpowp-Suc-E2*: $(P\ \hat{}\hat{}\ Suc\ n)\ x\ z \Longrightarrow (\bigwedge y.\ P\ x\ y \Longrightarrow (P\ \hat{}\hat{}\ n)\ y\ z \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *relpow-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 *relpowp-Suc-D2'*: $\forall x\ y\ z.\ (P\ \hat{}\hat{}\ n)\ x\ y \wedge P\ y\ z \longrightarrow (\exists w.\ P\ x\ w \wedge (P\ \hat{}\hat{}\ n)\ w\ z)$
⟨*proof*⟩

lemma *relpow-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 *relpowp-E2*:
$(P\ \hat{}\hat{}\ n)\ x\ z \Longrightarrow$
$(n = 0 \Longrightarrow x = z \Longrightarrow Q) \Longrightarrow$
$(\bigwedge y\ m.\ n = Suc\ m \Longrightarrow P\ x\ y \Longrightarrow (P\ \hat{}\hat{}\ m)\ y\ z \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *relpow-add*: $R\ \hat{}\hat{}\ (m + n) = R\hat{}\hat{}m\ O\ R\hat{}\hat{}n$
⟨*proof*⟩

lemma *relpowp-add*: $P\ \hat{}\hat{}\ (m + n) = P\ \hat{}\hat{}\ m\ OO\ P\ \hat{}\hat{}\ n$
⟨*proof*⟩

lemma *relpow-commute*: $R\ O\ R\ \hat{}\hat{}\ n = R\ \hat{}\hat{}\ n\ O\ R$
⟨*proof*⟩

lemma *relpowp-commute*: $P\ OO\ P\ \hat{}\hat{}\ n = P\ \hat{}\hat{}\ n\ OO\ P$
⟨*proof*⟩

lemma *relpow-empty*: $0 < n \Longrightarrow (\{\} :: ('a \times 'a)\ set)\ \hat{}\hat{}\ n = \{\}$
⟨*proof*⟩

lemma *relpowp-bot*: $0 < n \Longrightarrow (\bot :: 'a \Rightarrow 'a \Rightarrow bool)\ \hat{}\hat{}\ n = \bot$
⟨*proof*⟩

lemma *rtrancl-imp-UN-relpow*:
assumes $p \in R^*$
shows $p \in (\bigcup n.\ R\ \hat{}\hat{}\ n)$
⟨*proof*⟩

lemma *rtranclp-imp-Sup-relpowp*:
assumes $(P^{**})\ x\ y$
shows $(\bigsqcup n.\ P\ \hat{}\hat{}\ n)\ x\ y$
⟨*proof*⟩

lemma *relpow-imp-rtrancl*:
assumes $p \in R\ \hat{}\hat{}\ n$
shows $p \in R^*$
⟨*proof*⟩

lemma *relpowp-imp-rtranclp*: $(P \text{ ^^ } n)\ x\ y \Longrightarrow (P^{**})\ x\ y$
⟨*proof*⟩

lemma *rtrancl-is-UN-relpow*: $R^* = (\bigcup n.\ R \text{ ^^ } n)$
⟨*proof*⟩

lemma *rtranclp-is-Sup-relpowp*: $P^{**} = (\bigsqcup n.\ P \text{ ^^ } n)$
⟨*proof*⟩

lemma *rtrancl-power*: $p \in R^* \longleftrightarrow (\exists n.\ p \in R \text{ ^^ } n)$
⟨*proof*⟩

lemma *rtranclp-power*: $(P^{**})\ x\ y \longleftrightarrow (\exists n.\ (P \text{ ^^ } n)\ x\ y)$
⟨*proof*⟩

lemma *trancl-power*: $p \in R^+ \longleftrightarrow (\exists n > 0.\ p \in R \text{ ^^ } n)$
⟨*proof*⟩

lemma *tranclp-power*: $(P^{++})\ x\ y \longleftrightarrow (\exists n > 0.\ (P \text{ ^^ } n)\ x\ y)$
⟨*proof*⟩

lemma *rtrancl-imp-relpow*: $p \in R^* \Longrightarrow \exists n.\ p \in R \text{ ^^ } n$
⟨*proof*⟩

lemma *rtranclp-imp-relpowp*: $(P^{**})\ x\ y \Longrightarrow \exists n.\ (P \text{ ^^ } n)\ x\ y$
⟨*proof*⟩

By Sternagel/Thiemann:

lemma *relpow-fun-conv*: $(a,\ b) \in R \text{ ^^ } n \longleftrightarrow (\exists f.\ f\ 0 = a \wedge f\ n = b \wedge (\forall i<n.\ (f\ i,\ f\ (Suc\ i)) \in R))$
⟨*proof*⟩

lemma *relpowp-fun-conv*: $(P \text{ ^^ } n)\ x\ y \longleftrightarrow (\exists f.\ f\ 0 = x \wedge f\ n = y \wedge (\forall i<n.\ P\ (f\ i)\ (f\ (Suc\ i))))$
⟨*proof*⟩

lemma *relpow-finite-bounded1*:
fixes R :: $('a \times 'a)$ *set*
assumes *finite* R **and** $k > 0$
shows $R\text{^^}k \subseteq (\bigcup n\in\{n.\ 0 < n \wedge n \le card\ R\}.\ R\text{^^}n)$
(**is** - $\subseteq$ *?r*)
⟨*proof*⟩

lemma *relpow-finite-bounded*:
fixes R :: $('a \times 'a)$ *set*
assumes *finite* R
shows $R\text{^^}k \subseteq (UN\ n{:}\{n.\ n \le card\ R\}.\ R\text{^^}n)$
⟨*proof*⟩

lemma *rtrancl-finite-eq-relpow*: $finite\ R \Longrightarrow R^* = (\bigcup n\in\{n.\ n \le card\ R\}.\ R\,\hat{}\hat{}\,n)$
⟨*proof*⟩

lemma *trancl-finite-eq-relpow*: $finite\ R \Longrightarrow R^+ = (\bigcup n\in\{n.\ 0 < n \wedge n \le card\ R\}.\ R\,\hat{}\hat{}\,n)$
⟨*proof*⟩

lemma *finite-relcomp*[*simp*,*intro*]:
assumes *finite R* **and** *finite S*
shows *finite* $(R\ O\ S)$
⟨*proof*⟩

lemma *finite-relpow* [*simp*, *intro*]:
fixes $R :: ('a \times 'a)\ set$
assumes *finite R*
shows $n > 0 \Longrightarrow finite\ (R\,\hat{}\hat{}\,n)$
⟨*proof*⟩

lemma *single-valued-relpow*:
fixes $R :: ('a \times 'a)\ set$
shows $single\text{-}valued\ R \Longrightarrow single\text{-}valued\ (R\ \hat{}\hat{}\ n)$
⟨*proof*⟩

20.5 Bounded transitive closure

definition $ntrancl :: nat \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$
where $ntrancl\ n\ R = (\bigcup i\in\{i.\ 0 < i \wedge i \le Suc\ n\}.\ R\ \hat{}\hat{}\ i)$

lemma *ntrancl-Zero* [*simp*, *code*]: $ntrancl\ 0\ R = R$
⟨*proof*⟩

lemma *ntrancl-Suc* [*simp*]: $ntrancl\ (Suc\ n)\ R = ntrancl\ n\ R\ O\ (Id \cup R)$
⟨*proof*⟩

lemma [*code*]: $ntrancl\ (Suc\ n)\ r = (let\ r' = ntrancl\ n\ r\ in\ r' \cup r'\ O\ r)$
⟨*proof*⟩

lemma *finite-trancl-ntranl*: $finite\ R \Longrightarrow trancl\ R = ntrancl\ (card\ R - 1)\ R$
⟨*proof*⟩

20.6 Acyclic relations

definition $acyclic :: ('a \times 'a)\ set \Rightarrow bool$
where $acyclic\ r \longleftrightarrow (\forall x.\ (x,x) \notin r^+)$

abbreviation $acyclicP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $acyclicP\ r \equiv acyclic\ \{(x,\ y).\ r\ x\ y\}$

lemma *acyclic-irrefl* [*code*]: $acyclic\ r \longleftrightarrow irrefl\ (r^+)$
⟨*proof*⟩

lemma *acyclicI*: $\forall x.\ (x, x) \notin r^+ \Longrightarrow acyclic\ r$
⟨*proof*⟩

lemma (**in** *order*) *acyclicI-order*:
assumes $*$: $\bigwedge a\ b.\ (a, b) \in r \Longrightarrow f\ b < f\ a$
shows *acyclic r*
⟨*proof*⟩

lemma *acyclic-insert* [*iff*]: $acyclic\ (insert\ (y, x)\ r) \longleftrightarrow acyclic\ r \wedge (x, y) \notin r^*$
⟨*proof*⟩

lemma *acyclic-converse* [*iff*]: $acyclic\ (r^{-1}) \longleftrightarrow acyclic\ r$
⟨*proof*⟩

lemmas *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

lemma *acyclic-impl-antisym-rtrancl*: $acyclic\ r \Longrightarrow antisym\ (r^*)$
⟨*proof*⟩

lemma *acyclic-subset*: $acyclic\ s \Longrightarrow r \subseteq s \Longrightarrow acyclic\ r$
⟨*proof*⟩

20.7 Setup of transitivity reasoner

⟨*ML*⟩

Optional methods.

⟨*ML*⟩

end

21 Well-founded Recursion

theory *Wellfounded*
imports *Transitive-Closure*
begin

21.1 Basic Definitions

definition *wf* :: $('a \times 'a)\ set \Rightarrow bool$
where $wf\ r \longleftrightarrow (\forall P.\ (\forall x.\ (\forall y.\ (y, x) \in r \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall x.\ P\ x))$

definition *wfP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $wfP\ r \longleftrightarrow wf\ \{(x, y).\ r\ x\ y\}$

lemma *wfP-wf-eq* [*pred-set-conv*]: $wfP\ (\lambda x\ y.\ (x, y) \in r) = wf\ r$

⟨*proof*⟩

lemma *wfUNIVI*: $(\bigwedge P\ x.\ (\forall x.\ (\forall y.\ (y,\ x) \in r \longrightarrow P\ y) \longrightarrow P\ x) \Longrightarrow P\ x) \Longrightarrow$ *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*:
assumes $r \subseteq A \times B$
and $\bigwedge x\ P.\ [\![\forall x.\ (\forall y.\ (y,\ x) \in r \longrightarrow P\ y) \longrightarrow P\ x;\ \ x \in A;\ x \in B]\!] \Longrightarrow P\ x$
shows *wf r*
⟨*proof*⟩

lemma *wf-induct*:
assumes *wf r*
and $\bigwedge x.\ \forall y.\ (y,\ x) \in r \longrightarrow P\ y \Longrightarrow P\ x$
shows $P\ a$
⟨*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* $\Longrightarrow (a,\ x) \in r \Longrightarrow (x,\ a) \notin r$
⟨*proof*⟩

lemma *wf-asym*:
assumes *wf r* $(a,\ x) \in r$
obtains $(x,\ a) \notin r$
⟨*proof*⟩

lemma *wf-not-refl* [*simp*]: *wf r* $\Longrightarrow (a,\ a) \notin r$
⟨*proof*⟩

lemma *wf-irrefl*:
assumes *wf r*
obtains $(a,\ a) \notin r$
⟨*proof*⟩

lemma *wf-wellorderI*:
assumes *wf*: *wf* $\{(x::'a::ord,\ y).\ x < y\}$
and *lin*: *OFCLASS*('*a*::*ord*, *linorder-class*)
shows *OFCLASS*('*a*::*ord*, *wellorder-class*)

⟨*proof*⟩

lemma (**in** *wellorder*) *wf*: $wf\ \{(x,\ y).\ x < y\}$
⟨*proof*⟩

21.2 Basic Results

Point-free characterization of well-foundedness

lemma *wfE-pf*:
assumes *wf*: $wf\ R$
and *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*⟩

21.2.1 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 *wfE-min'*:
$wf\ R \Longrightarrow Q \neq \{\} \Longrightarrow (\bigwedge z.\ z \in Q \Longrightarrow (\bigwedge y.\ (y,\ z) \in R \Longrightarrow y \notin Q) \Longrightarrow thesis) \Longrightarrow thesis$
⟨*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*: $wf\ r \longleftrightarrow (\forall Q\ x.\ x \in Q \longrightarrow (\exists z \in Q.\ \forall y.\ (y,\ z) \in r \longrightarrow y \notin Q))$
⟨*proof*⟩

lemmas *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

21.2.2 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* $(r^{-1}) \Longrightarrow$ *wf* $((r^{+})^{-1})$
⟨*proof*⟩

Well-foundedness of subsets

lemma *wf-subset*: *wf* $r \Longrightarrow p \subseteq r \Longrightarrow$ *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* ($\lambda x\ y.$ *False*)
⟨*proof*⟩

lemma *wf-Int1*: *wf* $r \Longrightarrow$ *wf* $(r \cap r')$
⟨*proof*⟩

lemma *wf-Int2*: *wf* $r \Longrightarrow$ *wf* $(r' \cap r)$
⟨*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) $\longleftrightarrow$ *wf* $r \wedge (x, y) \notin r^*$
⟨*proof*⟩

21.2.3 Well-foundedness of image

lemma *wf-map-prod-image*: *wf* $r \Longrightarrow$ *inj* $f \Longrightarrow$ *wf* (*map-prod* f f ' r)
⟨*proof*⟩

21.3 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*:

assumes $\forall\, i{\in}I.\ wf\ (r\ i)$
and $\forall\, i{\in}I.\ \forall\, j{\in}I.\ r\ i \neq r\ j \longrightarrow Domain\ (r\ i) \cap Range\ (r\ j) = \{\}$
shows $wf\ (\bigcup i{\in}I.\ r\ i)$
$\langle proof \rangle$

lemma *wfP-SUP*:
$\forall\, i.\ wfP\ (r\ i) \Longrightarrow \forall\, i\ j.\ r\ i \neq r\ j \longrightarrow inf\ (Domainp\ (r\ i))\ (Rangep\ (r\ j)) = bot \Longrightarrow$
$wfP\ (SUPREMUM\ UNIV\ r)$
$\langle proof \rangle$

lemma *wf-Union*:
assumes $\forall\, r{\in}R.\ wf\ r$
and $\forall\, r{\in}R.\ \forall\, s{\in}R.\ r \neq s \longrightarrow Domain\ r \cap Range\ s = \{\}$
shows $wf\ (\bigcup R)$
$\langle proof \rangle$

Intuition: We find an $R \cup S$-min element of a nonempty subset A by case distinction.

1. There is a step $a\ {-R\rightarrow}\ b$ with $a,\, b \in A$. Pick an R-min element z of the (nonempty) set $\{a{\in}A \mid \exists\, b{\in}A.\ a\ {-R\rightarrow}\ b\}$. By definition, there is $z' \in A$ s.t. $z\ {-R\rightarrow}\ z'$. Because z is R-min in the subset, z' must be R-min in A. Because z' has an R-predecessor, it cannot have an S-successor and is thus S-min in A as well.

2. There is no such step. Pick an S-min element of A. In this case it must be an R-min element of A as well.

lemma *wf-Un*: $wf\ r \Longrightarrow wf\ s \Longrightarrow Domain\ r \cap Range\ s = \{\} \Longrightarrow wf\ (r \cup s)$
$\langle proof \rangle$

lemma *wf-union-merge*: $wf\ (R \cup S) = wf\ (R\ O\ R \cup S\ O\ R \cup S)$
(**is** $wf\ ?A = wf\ ?B$)
$\langle proof \rangle$

lemma *wf-comp-self*: $wf\ R \longleftrightarrow wf\ (R\ O\ R)$ — special case
$\langle proof \rangle$

21.4 Well-Foundedness of Composition

Bachmair and Dershowitz 1986, Lemma 2. [Provided by Tjark Weber]

lemma *qc-wf-relto-iff*:
assumes $R\ O\ S \subseteq (R \cup S)^*\ O\ R$ — R quasi-commutes over S
shows $wf\ (S^*\ O\ R\ O\ S^*) \longleftrightarrow wf\ R$
(**is** $wf\ ?S \longleftrightarrow$ -)
$\langle proof \rangle$

corollary *wf-relcomp-compatible*:
assumes *wf R* **and** $R\ O\ S \subseteq S\ O\ R$
shows *wf* $(S\ O\ R)$
⟨*proof*⟩

21.5 Acyclic relations

lemma *wf-acyclic*: *wf r* $\Longrightarrow$ *acyclic r*
⟨*proof*⟩

lemmas *wfP-acyclicP* = *wf-acyclic* [*to-pred*]

21.5.1 Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf* [*rule-format*]: *finite r* $\Longrightarrow$ *acyclic r* $\longrightarrow$ *wf r*
⟨*proof*⟩

lemma *finite-acyclic-wf-converse*: *finite r* $\Longrightarrow$ *acyclic r* $\Longrightarrow$ *wf* (r^{-1})
⟨*proof*⟩

Observe that the converse of an irreflexive, transitive, and finite relation is again well-founded. Thus, we may employ it for well-founded induction.

lemma *wf-converse*:
assumes *irrefl r* **and** *trans r* **and** *finite r*
shows *wf* (r^{-1})
⟨*proof*⟩

lemma *wf-iff-acyclic-if-finite*: *finite r* $\Longrightarrow$ *wf r* = *acyclic r*
⟨*proof*⟩

21.6 *nat* is well-founded

lemma *less-nat-rel*: *op* $<$ = $(\lambda m\ n.\ n = Suc\ m)^{++}$
⟨*proof*⟩

definition *pred-nat* :: $(nat \times nat)$ *set*
where *pred-nat* = $\{(m, n).\ n = Suc\ m\}$

definition *less-than* :: $(nat \times nat)$ *set*
where *less-than* = $pred\text{-}nat^{+}$

lemma *less-eq*: $(m, n) \in pred\text{-}nat^{+} \longleftrightarrow m < n$
⟨*proof*⟩

lemma *pred-nat-trancl-eq-le*: $(m, n) \in pred\text{-}nat^{*} \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *wf-pred-nat*: *wf pred-nat*
⟨*proof*⟩

lemma *wf-less-than* [*iff*]: *wf less-than*
⟨*proof*⟩

lemma *trans-less-than* [*iff*]: *trans less-than*
⟨*proof*⟩

lemma *less-than-iff* [*iff*]: $((x,y) \in less\text{-}than) = (x<y)$
⟨*proof*⟩

lemma *wf-less*: $wf\ \{(x,\ y{::}nat).\ x < y\}$
⟨*proof*⟩

21.7 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [3].

inductive-set $acc :: ('a \times 'a)\ set \Rightarrow 'a\ set$ **for** $r :: ('a \times 'a)\ set$
where *accI*: $(\bigwedge y.\ (y,\ x) \in r \Longrightarrow y \in acc\ r) \Longrightarrow x \in acc\ r$

abbreviation $termip :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$
where $termip\ r \equiv accp\ (r^{-1-1})$

abbreviation $termi :: ('a \times 'a)\ set \Rightarrow 'a\ set$
where $termi\ r \equiv acc\ (r^{-1})$

lemmas $accpI = accp.accI$

lemma *accp-eq-acc* [*code*]: $accp\ r = (\lambda x.\ x \in Wellfounded.acc\ \{(x,\ y).\ r\ x\ y\})$
⟨*proof*⟩

Induction rules

theorem *accp-induct*:
assumes *major*: $accp\ r\ a$
assumes *hyp*: $\bigwedge x.\ accp\ r\ x \Longrightarrow \forall y.\ r\ y\ x \longrightarrow P\ y \Longrightarrow P\ x$
shows $P\ a$
⟨*proof*⟩

lemmas *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: $accp\ r\ b \Longrightarrow r\ a\ b \Longrightarrow accp\ r\ a$
⟨*proof*⟩

lemma *not-accp-down*:
assumes *na*: $\neg\ accp\ R\ x$
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 $R1 \le R2$
shows $accp\ R2 \le accp\ R1$
⟨*proof*⟩

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
assumes *subset*: $D \le accp\ R$
and *dcl*: $\bigwedge x\ z.\ D\ x \Longrightarrow R\ z\ x \Longrightarrow D\ z$
and $D\ x$
and *istep*: $\bigwedge x.\ D\ x \Longrightarrow (\bigwedge z.\ R\ z\ x \Longrightarrow P\ z) \Longrightarrow P\ x$
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*]
lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set*]

21.8 Tools for building wellfounded relations

Inverse Image

lemma *wf-inv-image* [*simp*,*intro*!]: $wf\ r \Longrightarrow wf\ (inv\text{-}image\ r\ f)$
for $f :: 'a \Rightarrow 'b$
⟨*proof*⟩

Measure functions into *nat*

definition *measure* :: $('a \Rightarrow nat) \Rightarrow ('a \times 'a)$ *set*
  where *measure* = *inv-image less-than*

lemma *in-measure*[*simp*, *code-unfold*]: $(x, y) \in$ *measure* $f \longleftrightarrow f\ x < f\ y$
  ⟨*proof*⟩

lemma *wf-measure* [*iff*]: *wf* (*measure* f)
  ⟨*proof*⟩

lemma *wf-if-measure*: $(\bigwedge x.\ P\ x \Longrightarrow f(g\ x) < f\ x) \Longrightarrow wf\ \{(y,x).\ P\ x \wedge y = g\ x\}$
  for f :: $'a \Rightarrow nat$
  ⟨*proof*⟩

21.8.1 Lexicographic combinations

definition *lex-prod* :: $('a \times 'a)$ *set* $\Rightarrow ('b \times 'b)$ *set* $\Rightarrow (('a \times 'b) \times ('a \times 'b))$ *set*
   (**infixr** $<*lex*>$ *80*)
  where $ra <*lex*> rb = \{((a, b), (a', b')).\ (a, a') \in ra \vee a = a' \wedge (b, b') \in rb\}$

lemma *wf-lex-prod* [*intro*!]: $wf\ ra \Longrightarrow wf\ rb \Longrightarrow wf\ (ra <*lex*> rb)$
  ⟨*proof*⟩

lemma *in-lex-prod*[*simp*]: $((a, b), (a', b')) \in r <*lex*> s \longleftrightarrow (a, a') \in r \vee a = a' \wedge (b, b') \in s$
  ⟨*proof*⟩

$<*lex*>$ preserves transitivity

lemma *trans-lex-prod* [*intro*!]: $trans\ R1 \Longrightarrow trans\ R2 \Longrightarrow trans\ (R1 <*lex*> R2)$
  ⟨*proof*⟩

lexicographic combinations with measure functions

definition *mlex-prod* :: $('a \Rightarrow nat) \Rightarrow ('a \times 'a)$ *set* $\Rightarrow ('a \times 'a)$ *set* (**infixr** $<*mlex*>$ *80*)
  where $f <*mlex*> R = inv\text{-}image\ (less\text{-}than <*lex*> R)\ (\lambda x.\ (f\ x, x))$

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* :: $('a\ set \times 'a\ set)\ set$

where *finite-psubset* = $\{(A, B).\ A \subset B \wedge$ *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* $\longleftrightarrow A \subset B \wedge$ *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 *min-ext* $r = \{(X, Y) \mid X\ Y.\ X \neq \{\} \wedge (\forall y \in Y.\ (\exists x \in X.\ (x, y) \in r))\}$

lemma *min-ext-wf*:
assumes *wf* r
shows *wf* (*min-ext* r)
⟨*proof*⟩

21.8.2 Bounded increase must terminate

lemma *wf-bounded-measure*:
fixes *ub* :: $'a \Rightarrow nat$
and f :: $'a \Rightarrow nat$
assumes $\bigwedge a\ b.\ (b, a) \in r \Longrightarrow ub\ b \leq ub\ a \wedge ub\ a \geq f\ b \wedge f\ b > f\ a$
shows *wf* r
⟨*proof*⟩

lemma *wf-bounded-set*:
fixes *ub* :: $'a \Rightarrow 'b$ *set*
and f :: $'a \Rightarrow 'b$ *set*
assumes $\bigwedge a\ b.\ (b,a) \in r \Longrightarrow$ *finite* $(ub\ a) \wedge ub\ b \subseteq ub\ a \wedge ub\ a \supseteq f\ b \wedge f\ b \supset$

$f\ a$
shows $wf\ r$
⟨*proof*⟩

lemma *finite-subset-wf*:
assumes *finite A*
shows $wf\ \{(X,Y).\ X \subset Y \wedge Y \subseteq A\}$
⟨*proof*⟩

hide-const (**open**) *acc accp*

end

22 Well-Founded Recursion Combinator

theory *Wfrec*
imports *Wellfounded*
begin

inductive *wfrec-rel* :: $('a \times 'a)\ set \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
for $R\ F$
where *wfrecI*: $(\bigwedge z.\ (z,\ x) \in R \Longrightarrow wfrec\text{-}rel\ R\ F\ z\ (g\ z)) \Longrightarrow wfrec\text{-}rel\ R\ F\ x\ (F\ g\ x)$

definition *cut* :: $('a \Rightarrow 'b) \Rightarrow ('a \times 'a)\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b$
where $cut\ f\ R\ x = (\lambda y.\ if\ (y,\ x) \in R\ then\ f\ y\ else\ undefined)$

definition *adm-wf* :: $('a \times 'a)\ set \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow bool$
where $adm\text{-}wf\ R\ F \longleftrightarrow (\forall f\ g\ x.\ (\forall z.\ (z,\ x) \in R \longrightarrow f\ z = g\ z) \longrightarrow F\ f\ x = F\ g\ x)$

definition *wfrec* :: $('a \times 'a)\ set \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow ('a \Rightarrow 'b)$
where $wfrec\ R\ F = (\lambda x.\ THE\ y.\ wfrec\text{-}rel\ R\ (\lambda f\ x.\ F\ (cut\ f\ R\ x)\ x)\ x\ y)$

lemma *cuts-eq*: $(cut\ f\ R\ x = cut\ g\ R\ x) \longleftrightarrow (\forall y.\ (y,\ x) \in R \longrightarrow f\ y = g\ y)$
⟨*proof*⟩

lemma *cut-apply*: $(x,\ a) \in R \Longrightarrow cut\ f\ R\ a\ x = f\ x$
⟨*proof*⟩

Inductive characterization of *wfrec* combinator; for details see: John Harrison, "Inductive definitions: automation and application".

lemma *theI-unique*: $\exists!x.\ P\ x \Longrightarrow P\ x \longleftrightarrow x = The\ P$
⟨*proof*⟩

lemma *wfrec-unique*:
assumes *adm-wf R F wf R*
shows $\exists!y.\ wfrec\text{-}rel\ R\ F\ x\ y$
⟨*proof*⟩

lemma *adm-lemma*: $adm\text{-}wf\ R\ (\lambda f\ x.\ F\ (cut\ f\ R\ x)\ x)$
⟨*proof*⟩

lemma *wfrec*: $wf\ R \Longrightarrow wfrec\ R\ F\ a = F\ (cut\ (wfrec\ R\ F)\ R\ a)\ a$
⟨*proof*⟩

This form avoids giant explosions in proofs. NOTE USE OF ≡.

lemma *def-wfrec*: $f \equiv wfrec\ R\ F \Longrightarrow wf\ R \Longrightarrow f\ a = F\ (cut\ f\ R\ a)\ a$
⟨*proof*⟩

22.0.1 Well-founded recursion via genuine fixpoints

lemma *wfrec-fixpoint*:
assumes *wf*: $wf\ R$
and *adm*: $adm\text{-}wf\ R\ F$
shows $wfrec\ R\ F = F\ (wfrec\ R\ F)$
⟨*proof*⟩

22.1 Wellfoundedness of *same-fst*

definition *same-fst* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow ('b \times 'b)\ set) \Rightarrow (('a \times 'b) \times ('a \times 'b))\ set$
where $same\text{-}fst\ P\ R = \{((x', y'), (x, y))\ .\ x' = x \wedge P\ x \wedge (y',y) \in R\ x\}$
— For *wfrec* declarations where the first n parameters stay unchanged in the recursive call.

lemma *same-fstI* [*intro*!]: $P\ x \Longrightarrow (y', y) \in R\ x \Longrightarrow ((x, y'), (x, y)) \in same\text{-}fst\ P\ R$
⟨*proof*⟩

lemma *wf-same-fst*:
assumes *prem*: $\bigwedge x.\ P\ x \Longrightarrow wf\ (R\ x)$
shows $wf\ (same\text{-}fst\ P\ R)$
⟨*proof*⟩

end

23 Orders as Relations

theory *Order-Relation*
imports *Wfrec*
begin

23.1 Orders on a set

definition $preorder\text{-}on\ A\ r \equiv refl\text{-}on\ A\ r \wedge trans\ r$

definition $partial\text{-}order\text{-}on\ A\ r \equiv preorder\text{-}on\ A\ r \wedge antisym\ r$

definition *linear-order-on* A $r \equiv$ *partial-order-on* A $r \wedge$ *total-on* A r

definition *strict-linear-order-on* A $r \equiv$ *trans* $r \wedge$ *irrefl* $r \wedge$ *total-on* A r

definition *well-order-on* A $r \equiv$ *linear-order-on* A $r \wedge$ *wf*$(r - Id)$

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def
strict-linear-order-on-def well-order-on-def

lemma *preorder-on-empty*[*simp*]: *preorder-on* {} {}
⟨*proof*⟩

lemma *partial-order-on-empty*[*simp*]: *partial-order-on* {} {}
⟨*proof*⟩

lemma *lnear-order-on-empty*[*simp*]: *linear-order-on* {} {}
⟨*proof*⟩

lemma *well-order-on-empty*[*simp*]: *well-order-on* {} {}
⟨*proof*⟩

lemma *preorder-on-converse*[*simp*]: *preorder-on* A (r^{-1}) = *preorder-on* A r
⟨*proof*⟩

lemma *partial-order-on-converse*[*simp*]: *partial-order-on* A (r^{-1}) = *partial-order-on* A r
⟨*proof*⟩

lemma *linear-order-on-converse*[*simp*]: *linear-order-on* A (r^{-1}) = *linear-order-on* A r
⟨*proof*⟩

lemma *strict-linear-order-on-diff-Id*: *linear-order-on* A $r \Longrightarrow$ *strict-linear-order-on* A $(r - Id)$
⟨*proof*⟩

lemma *linear-order-on-singleton* [*simp*]: *linear-order-on* $\{x\}$ $\{(x, x)\}$
⟨*proof*⟩

lemma *linear-order-on-acyclic*:
assumes *linear-order-on* A r
shows *acyclic* $(r - Id)$
⟨*proof*⟩

lemma *linear-order-on-well-order-on*:
  assumes *finite r*
  shows *linear-order-on A r* $\longleftrightarrow$ *well-order-on A r*
  ⟨*proof*⟩

23.2 Orders on the field

abbreviation *Refl r* $\equiv$ *refl-on* (*Field r*) *r*

abbreviation *Preorder r* $\equiv$ *preorder-on* (*Field r*) *r*

abbreviation *Partial-order r* $\equiv$ *partial-order-on* (*Field r*) *r*

abbreviation *Total r* $\equiv$ *total-on* (*Field r*) *r*

abbreviation *Linear-order r* $\equiv$ *linear-order-on* (*Field r*) *r*

abbreviation *Well-order r* $\equiv$ *well-order-on* (*Field r*) *r*

lemma *subset-Image-Image-iff*:
  $Preorder\ r \Longrightarrow A \subseteq Field\ r \Longrightarrow B \subseteq Field\ r \Longrightarrow$
    $r `` A \subseteq r `` B \longleftrightarrow (\forall a \in A. \exists b \in B. (b, a) \in r)$
  ⟨*proof*⟩

lemma *subset-Image1-Image1-iff*:
  $Preorder\ r \Longrightarrow a \in Field\ r \Longrightarrow b \in Field\ r \Longrightarrow r `` \{a\} \subseteq r `` \{b\} \longleftrightarrow (b, a) \in r$
  ⟨*proof*⟩

lemma *Refl-antisym-eq-Image1-Image1-iff*:
  assumes *Refl r*
    and *as*: *antisym r*
    and *abf*: $a \in Field\ r\ b \in Field\ r$
  shows $r `` \{a\} = r `` \{b\} \longleftrightarrow a = b$
    (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *Partial-order-eq-Image1-Image1-iff*:
  $Partial\text{-}order\ r \Longrightarrow a \in Field\ r \Longrightarrow b \in Field\ r \Longrightarrow r `` \{a\} = r `` \{b\} \longleftrightarrow a = b$
  ⟨*proof*⟩

lemma *Total-Id-Field*:
  assumes *Total r*
    and *not-Id*: $\neg\ r \subseteq Id$
  shows $Field\ r = Field\ (r - Id)$
  ⟨*proof*⟩

23.3 Orders on a type

abbreviation *strict-linear-order* ≡ *strict-linear-order-on UNIV*

abbreviation *linear-order* ≡ *linear-order-on UNIV*

abbreviation *well-order* ≡ *well-order-on UNIV*

23.4 Order-like relations

In this subsection, we develop basic concepts and results pertaining to order-like relations, i.e., to reflexive and/or transitive and/or symmetric and/or total relations. We also further define upper and lower bounds operators.

23.4.1 Auxiliaries

lemma *refl-on-domain*: *refl-on* $A\ r \Longrightarrow (a,\ b) \in r \Longrightarrow a \in A \wedge b \in A$
⟨*proof*⟩

corollary *well-order-on-domain*: *well-order-on* $A\ r \Longrightarrow (a,\ b) \in r \Longrightarrow a \in A \wedge b \in A$
⟨*proof*⟩

lemma *well-order-on-Field*: *well-order-on* $A\ r \Longrightarrow A =$ *Field* r
⟨*proof*⟩

lemma *well-order-on-Well-order*: *well-order-on* $A\ r \Longrightarrow A =$ *Field* $r \wedge$ *Well-order* r
⟨*proof*⟩

lemma *Total-subset-Id*:
assumes *Total* r
and $r \subseteq Id$
shows $r = \{\} \vee (\exists\, a.\ r = \{(a,\ a)\})$
⟨*proof*⟩

lemma *Linear-order-in-diff-Id*:
assumes *Linear-order* r
and $a \in$ *Field* r
and $b \in$ *Field* r
shows $(a,\ b) \in r \longleftrightarrow (b,\ a) \notin r - Id$
⟨*proof*⟩

23.4.2 The upper and lower bounds operators

Here we define upper ("above") and lower ("below") bounds operators. We think of r as a *non-strict* relation. The suffix S at the names of some operators indicates that the bounds are strict – e.g., *underS* a is the set of all strict lower bounds of a (w.r.t. r). Capitalization of the first letter in the name reminds that the operator acts on sets, rather than on individual elements.

definition *under* :: $'a\ rel \Rightarrow 'a \Rightarrow 'a\ set$
where $under\ r\ a \equiv \{b.\ (b,\ a) \in r\}$

definition *underS* :: $'a\ rel \Rightarrow 'a \Rightarrow 'a\ set$
where $underS\ r\ a \equiv \{b.\ b \neq a \wedge (b,\ a) \in r\}$

definition *Under* :: $'a\ rel \Rightarrow 'a\ set \Rightarrow 'a\ set$
where $Under\ r\ A \equiv \{b \in Field\ r.\ \forall\, a \in A.\ (b,\ a) \in r\}$

definition *UnderS* :: $'a\ rel \Rightarrow 'a\ set \Rightarrow 'a\ set$
where $UnderS\ r\ A \equiv \{b \in Field\ r.\ \forall\, a \in A.\ b \neq a \wedge (b,\ a) \in r\}$

definition *above* :: $'a\ rel \Rightarrow 'a \Rightarrow 'a\ set$
where $above\ r\ a \equiv \{b.\ (a,\ b) \in r\}$

definition *aboveS* :: $'a\ rel \Rightarrow 'a \Rightarrow 'a\ set$
where $aboveS\ r\ a \equiv \{b.\ b \neq a \wedge (a,\ b) \in r\}$

definition *Above* :: $'a\ rel \Rightarrow 'a\ set \Rightarrow 'a\ set$
where $Above\ r\ A \equiv \{b \in Field\ r.\ \forall\, a \in A.\ (a,\ b) \in r\}$

definition *AboveS* :: $'a\ rel \Rightarrow 'a\ set \Rightarrow 'a\ set$
where $AboveS\ r\ A \equiv \{b \in Field\ r.\ \forall\, a \in A.\ b \neq a \wedge (a,\ b) \in r\}$

definition *ofilter* :: $'a\ rel \Rightarrow 'a\ set \Rightarrow bool$
where $ofilter\ r\ A \equiv A \subseteq Field\ r \wedge (\forall\, a \in A.\ under\ r\ a \subseteq A)$

Note: In the definitions of $Above[S]$ and $Under[S]$, we bounded comprehension by $Field\ r$ in order to properly cover the case of A being empty.

lemma *underS-subset-under*: $underS\ r\ a \subseteq under\ r\ a$
⟨*proof*⟩

lemma *underS-notIn*: $a \notin underS\ r\ a$
⟨*proof*⟩

lemma *Refl-under-in*: $Refl\ r \Longrightarrow a \in Field\ r \Longrightarrow a \in under\ r\ a$
⟨*proof*⟩

lemma *AboveS-disjoint*: $A \cap (AboveS\ r\ A) = \{\}$
⟨*proof*⟩

lemma *in-AboveS-underS*: $a \in Field\ r \Longrightarrow a \in AboveS\ r\ (underS\ r\ a)$
⟨*proof*⟩

lemma *Refl-under-underS*: $Refl\ r \Longrightarrow a \in Field\ r \Longrightarrow under\ r\ a = underS\ r\ a \cup \{a\}$
⟨*proof*⟩

lemma *underS-empty*: $a \notin \mathit{Field}\ r \Longrightarrow \mathit{underS}\ r\ a = \{\}$
⟨*proof*⟩

lemma *under-Field*: $\mathit{under}\ r\ a \subseteq \mathit{Field}\ r$
⟨*proof*⟩

lemma *underS-Field*: $\mathit{underS}\ r\ a \subseteq \mathit{Field}\ r$
⟨*proof*⟩

lemma *underS-Field2*: $a \in \mathit{Field}\ r \Longrightarrow \mathit{underS}\ r\ a \subset \mathit{Field}\ r$
⟨*proof*⟩

lemma *underS-Field3*: $\mathit{Field}\ r \neq \{\} \Longrightarrow \mathit{underS}\ r\ a \subset \mathit{Field}\ r$
⟨*proof*⟩

lemma *AboveS-Field*: $\mathit{AboveS}\ r\ A \subseteq \mathit{Field}\ r$
⟨*proof*⟩

lemma *under-incr*:
assumes *trans r*
and $(a,\ b) \in r$
shows $\mathit{under}\ r\ a \subseteq \mathit{under}\ r\ b$
⟨*proof*⟩

lemma *underS-incr*:
assumes *trans r*
and *antisym r*
and *ab*: $(a,\ b) \in r$
shows $\mathit{underS}\ r\ a \subseteq \mathit{underS}\ r\ b$
⟨*proof*⟩

lemma *underS-incl-iff*:
assumes *LO*: *Linear-order r*
and *INa*: $a \in \mathit{Field}\ r$
and *INb*: $b \in \mathit{Field}\ r$
shows $\mathit{underS}\ r\ a \subseteq \mathit{underS}\ r\ b \longleftrightarrow (a,\ b) \in r$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *finite-Linear-order-induct*[*consumes 3*, *case-names step*]:
assumes *Linear-order r*
and $x \in \mathit{Field}\ r$
and *finite r*
and *step*: $\bigwedge x.\ x \in \mathit{Field}\ r \Longrightarrow (\bigwedge y.\ y \in \mathit{aboveS}\ r\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows $P\ x$
⟨*proof*⟩

23.5 Variations on Well-Founded Relations

This subsection contains some variations of the results from *Wellfounded*:

- means for slightly more direct definitions by well-founded recursion;
- variations of well-founded induction;
- means for proving a linear order to be a well-order.

23.5.1 Characterizations of well-foundedness

A transitive relation is well-founded iff it is "locally" well-founded, i.e., iff its restriction to the lower bounds of of any element is well-founded.

lemma *trans-wf-iff*:
assumes *trans r*
shows $wf\ r \longleftrightarrow (\forall a.\ wf\ (r \cap (r^{-1}\text{``}\{a\} \times r^{-1}\text{``}\{a\})))$
⟨*proof*⟩

A transitive relation is well-founded if all initial segments are finite.

corollary *wf-finite-segments*:
assumes *irrefl r* **and** *trans r* **and** $\bigwedge x.\ finite\ \{y.\ (y,\ x) \in r\}$
shows $wf\ (r)$
⟨*proof*⟩

The next lemma is a variation of *wf-eq-minimal* from Wellfounded, allowing one to assume the set included in the field.

lemma *wf-eq-minimal2*: $wf\ r \longleftrightarrow (\forall A.\ A \subseteq Field\ r \wedge A \neq \{\} \longrightarrow (\exists a \in A.\ \forall a' \in A.\ (a',\ a) \notin r))$
⟨*proof*⟩

23.5.2 Characterizations of well-foundedness

The next lemma and its corollary enable one to prove that a linear order is a well-order in a way which is more standard than via well-foundedness of the strict version of the relation.

lemma *Linear-order-wf-diff-Id*:
assumes *Linear-order r*
shows $wf\ (r - Id) \longleftrightarrow (\forall A \subseteq Field\ r.\ A \neq \{\} \longrightarrow (\exists a \in A.\ \forall a' \in A.\ (a,\ a') \in r))$
⟨*proof*⟩

corollary *Linear-order-Well-order-iff*:
$Linear\text{-}order\ r \Longrightarrow$
$Well\text{-}order\ r \longleftrightarrow (\forall A \subseteq Field\ r.\ A \neq \{\} \longrightarrow (\exists a \in A.\ \forall a' \in A.\ (a,\ a') \in r))$
⟨*proof*⟩

end

24 Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *Wellfounded*
keywords *specification* :: *thy-goal*
begin

24.1 Hilbert's epsilon

axiomatization $Eps :: ('a \Rightarrow bool) \Rightarrow 'a$
where *someI*: $P\ x \Longrightarrow P\ (Eps\ P)$

syntax (*epsilon*)
-Eps :: $pttrn \Rightarrow bool \Rightarrow 'a$ $((3\epsilon\ \text{-./ -})\ [0,\ 10]\ 10)$
syntax (*input*)
-Eps :: $pttrn \Rightarrow bool \Rightarrow 'a$ $((3@\ \text{-./ -})\ [0,\ 10]\ 10)$
syntax
-Eps :: $pttrn \Rightarrow bool \Rightarrow 'a$ $((3SOME\ \text{-./ -})\ [0,\ 10]\ 10)$
translations
$SOME\ x.\ P \rightleftharpoons CONST\ Eps\ (\lambda x.\ P)$

⟨*ML*⟩

definition *inv-into* :: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$ **where**
$inv\text{-}into\ A\ f = (\lambda x.\ SOME\ y.\ y \in A \wedge f\ y = x)$

lemma *inv-into-def2*: $inv\text{-}into\ A\ f\ x = (SOME\ y.\ y \in A \wedge f\ y = x)$
⟨*proof*⟩

abbreviation *inv* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$ **where**
$inv \equiv inv\text{-}into\ UNIV$

24.2 Hilbert's Epsilon-operator

Easier to apply than *someI* if the witness comes from an existential formula.

lemma *someI-ex* [*elim?*]: $\exists x.\ P\ x \Longrightarrow P\ (SOME\ x.\ P\ x)$
⟨*proof*⟩

Easier to apply than *someI* because the conclusion has only one occurrence of P.

lemma *someI2*: $P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x.\ P\ x)$
⟨*proof*⟩

Easier to apply than *someI2* if the witness comes from an existential formula.

lemma *someI2-ex*: $\exists a.\ P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x.\ P\ x)$
⟨*proof*⟩

lemma *someI2-bex*: $\exists\, a{\in}A.\ P\ a \Longrightarrow (\bigwedge x.\ x \in A \wedge P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x.\ x \in A \wedge P\ x)$
⟨*proof*⟩

lemma *some-equality* [*intro*]: $P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow x = a) \Longrightarrow (SOME\ x.\ P\ x) = a$
⟨*proof*⟩

lemma *some1-equality*: $\exists !x.\ P\ x \Longrightarrow P\ a \Longrightarrow (SOME\ x.\ P\ x) = a$
⟨*proof*⟩

lemma *some-eq-ex*: $P\ (SOME\ x.\ P\ x) \longleftrightarrow (\exists\, x.\ P\ x)$
⟨*proof*⟩

lemma *some-in-eq*: $(SOME\ x.\ x \in A) \in A \longleftrightarrow A \neq \{\}$
⟨*proof*⟩

lemma *some-eq-trivial* [*simp*]: $(SOME\ y.\ y = x) = x$
⟨*proof*⟩

lemma *some-sym-eq-trivial* [*simp*]: $(SOME\ y.\ x = y) = x$
⟨*proof*⟩

24.3 Axiom of Choice, Proved Using the Description Operator

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*⟩

lemma *choice-iff*: $(\forall\, x.\ \exists\, y.\ Q\ x\ y) \longleftrightarrow (\exists\, f.\ \forall\, x.\ Q\ x\ (f\ x))$
⟨*proof*⟩

lemma *choice-iff'*: $(\forall\, x.\ P\ x \longrightarrow (\exists\, y.\ Q\ x\ y)) \longleftrightarrow (\exists\, f.\ \forall\, x.\ P\ x \longrightarrow Q\ x\ (f\ x))$
⟨*proof*⟩

lemma *bchoice-iff*: $(\forall\, x{\in}S.\ \exists\, y.\ Q\ x\ y) \longleftrightarrow (\exists\, f.\ \forall\, x{\in}S.\ Q\ x\ (f\ x))$
⟨*proof*⟩

lemma *bchoice-iff'*: $(\forall\, x{\in}S.\ P\ x \longrightarrow (\exists\, y.\ Q\ x\ y)) \longleftrightarrow (\exists\, f.\ \forall\, x{\in}S.\ P\ x \longrightarrow Q\ x\ (f\ x))$
⟨*proof*⟩

lemma *dependent-nat-choice*:
assumes *1*: $\exists\, x.\ P\ 0\ x$
and *2*: $\bigwedge x\ n.\ P\ n\ x \Longrightarrow \exists\, y.\ P\ (Suc\ n)\ y \wedge Q\ n\ x\ y$
shows $\exists\, f.\ \forall\, n.\ P\ n\ (f\ n) \wedge Q\ n\ (f\ n)\ (f\ (Suc\ n))$

⟨*proof*⟩

24.4 Function Inverse

lemma *inv-def*: $inv\ f = (\lambda y.\ SOME\ x.\ f\ x = y)$
⟨*proof*⟩

lemma *inv-into-into*: $x \in f\ `\ A \Longrightarrow inv\text{-}into\ A\ f\ x \in A$
⟨*proof*⟩

lemma *inv-identity* [*simp*]: $inv\ (\lambda a.\ a) = (\lambda a.\ a)$
⟨*proof*⟩

lemma *inv-id* [*simp*]: $inv\ id = id$
⟨*proof*⟩

lemma *inv-into-f-f* [*simp*]: $inj\text{-}on\ f\ A \Longrightarrow x \in A \Longrightarrow inv\text{-}into\ A\ f\ (f\ x) = x$
⟨*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*: $inj\text{-}on\ f\ A \Longrightarrow x \in A \Longrightarrow f\ x = y \Longrightarrow inv\text{-}into\ A\ f\ y = x$
⟨*proof*⟩

lemma *inv-f-eq*: $inj\ f \Longrightarrow f\ x = y \Longrightarrow inv\ f\ y = x$
⟨*proof*⟩

lemma *inj-imp-inv-eq*: $inj\ f \Longrightarrow \forall\, x.\ f\ (g\ x) = x \Longrightarrow inv\ f = g$
⟨*proof*⟩

But is it useful?

lemma *inj-transfer*:
assumes *inj*: $inj\ f$
and *minor*: $\bigwedge y.\ y \in range\ f \Longrightarrow P\ (inv\ f\ y)$
shows $P\ x$
⟨*proof*⟩

lemma *inj-iff*: $inj\ f \longleftrightarrow inv\ f \circ f = id$
⟨*proof*⟩

lemma *inv-o-cancel*[*simp*]: $inj\ f \Longrightarrow inv\ f \circ f = id$
⟨*proof*⟩

lemma *o-inv-o-cancel*[*simp*]: $inj\ f \Longrightarrow g \circ inv\ f \circ f = g$
⟨*proof*⟩

lemma *inv-into-image-cancel*[*simp*]: $inj\text{-}on\ f\ A \Longrightarrow S \subseteq A \Longrightarrow inv\text{-}into\ A\ f\ `\ f\ `\ S = S$
⟨*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*: $x \in f`A$
and *y*: $y \in f`A$
shows $x = y$
⟨*proof*⟩

lemma *inj-on-inv-into*: $B \subseteq f`A \Longrightarrow inj\text{-}on\ (inv\text{-}into\ A\ f)\ B$
⟨*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*: $surj\ f \longleftrightarrow f \circ inv\ f = id$
⟨*proof*⟩

lemma *surj-iff-all*: $surj\ f \longleftrightarrow (\forall x.\ f\ (inv\ f\ x) = x)$
⟨*proof*⟩

lemma *surj-imp-inv-eq*: $surj\ f \Longrightarrow \forall x.\ g\ (f\ x) = x \Longrightarrow inv\ f = g$
⟨*proof*⟩

lemma *bij-imp-bij-inv*: $bij\ f \Longrightarrow bij\ (inv\ f)$
⟨*proof*⟩

lemma *inv-equality*: $(\bigwedge x.\ g\ (f\ x) = x) \Longrightarrow (\bigwedge y.\ f\ (g\ y) = y) \Longrightarrow inv\ f = g$
⟨*proof*⟩

lemma *inv-inv-eq*: $bij\ f \Longrightarrow inv\ (inv\ f) = f$
⟨*proof*⟩

$bij\ (inv\ f)$ implies little about f. Consider $f :: bool \Rightarrow bool$ such that $f\ True = f\ False = True$. Then it ia consistent with axiom *someI* that $inv\ f$ could be any function at all, including the identity function. If $inv\ f = id$ then $inv\ f$ is a bijection, but $inj\ f$, $surj\ f$ and $inv\ (inv\ f) = f$ all fail.

lemma *inv-into-comp*:
$inj\text{-}on\ f\ (g\ `\ A) \Longrightarrow inj\text{-}on\ g\ A \Longrightarrow x \in f\ `\ g\ `\ A \Longrightarrow$
$inv\text{-}into\ A\ (f \circ g)\ x = (inv\text{-}into\ A\ g \circ inv\text{-}into\ (g\ `\ A)\ f)\ x$
⟨*proof*⟩

lemma *o-inv-distrib*: $bij\ f \Longrightarrow bij\ g \Longrightarrow inv\ (f \circ g) = inv\ g \circ inv\ f$
⟨*proof*⟩

lemma *image-f-inv-f*: $surj\ f \Longrightarrow f\ `\ (inv\ f\ `\ A) = A$
⟨*proof*⟩

lemma *image-inv-f-f*: $inj\ f \Longrightarrow inv\ f\ `\ (f\ `\ A) = A$
⟨*proof*⟩

lemma *bij-image-Collect-eq*: $bij\ f \Longrightarrow f\ `\ Collect\ P = \{y.\ P\ (inv\ f\ y)\}$
⟨*proof*⟩

lemma *bij-vimage-eq-inv-image*: $bij\ f \Longrightarrow f\ -`\ A = inv\ f\ `\ A$
⟨*proof*⟩

lemma *finite-fun-UNIVD1*:
assumes *fin*: $finite\ (UNIV :: ('a \Rightarrow 'b)\ set)$
and *card*: $card\ (UNIV :: 'b\ set) \neq Suc\ 0$
shows $finite\ (UNIV :: 'a\ set)$
⟨*proof*⟩

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *infinite-countable-subset*:
assumes *inf*: $\neg\ finite\ S$
shows $\exists f{::}nat \Rightarrow 'a.\ inj\ f \land range\ f \subseteq S$
— Courtesy of Stephan Merz
⟨*proof*⟩

lemma *infinite-iff-countable-subset*: $\neg\ finite\ S \longleftrightarrow (\exists f{::}nat \Rightarrow 'a.\ inj\ f \land range\ f \subseteq S)$
— Courtesy of Stephan Merz
⟨*proof*⟩

lemma *image-inv-into-cancel*:
assumes *surj*: $f`A = A'$
and *sub*: $B' \subseteq A'$
shows $f\ `((inv\text{-}into\ A\ f)`B') = B'$

⟨*proof*⟩

lemma *inv-into-inv-into-eq*:
assumes *bij-betw f A A'*
and *a*: $a \in A$
shows *inv-into* A' (*inv-into* A f) $a = f\ a$
⟨*proof*⟩

lemma *inj-on-iff-surj*:
assumes $A \neq \{\}$
shows $(\exists f.\ \textit{inj-on}\ f\ A \wedge f\ `\ A \subseteq A') \longleftrightarrow (\exists g.\ g\ `\ A' = A)$
⟨*proof*⟩

lemma *Ex-inj-on-UNION-Sigma*:
$\exists f.\ (\textit{inj-on}\ f\ (\bigcup i \in I.\ A\ i) \wedge f\ `\ (\bigcup i \in I.\ A\ i) \subseteq (\textit{SIGMA}\ i : I.\ A\ i))$
⟨*proof*⟩

lemma *inv-unique-comp*:
assumes *fg*: $f \circ g = id$
and *gf*: $g \circ f = id$
shows *inv* $f = g$
⟨*proof*⟩

24.5 Other Consequences of Hilbert's Epsilon

Hilbert's Epsilon and the *split* Operator

Looping simprule!

lemma *split-paired-Eps*: $(\textit{SOME}\ x.\ P\ x) = (\textit{SOME}\ (a, b).\ P\ (a, b))$
⟨*proof*⟩

lemma *Eps-case-prod*: $\textit{Eps}\ (\textit{case-prod}\ P) = (\textit{SOME}\ xy.\ P\ (\textit{fst}\ xy)\ (\textit{snd}\ xy))$
⟨*proof*⟩

lemma *Eps-case-prod-eq* [*simp*]: $(\textit{SOME}\ (x', y').\ x = x' \wedge y = y') = (x, y)$
⟨*proof*⟩

A relation is wellfounded iff it has no infinite descending chain.

lemma *wf-iff-no-infinite-down-chain*: $\textit{wf}\ r \longleftrightarrow (\nexists f.\ \forall i.\ (f\ (\textit{Suc}\ i), f\ i) \in r)$
(**is** - $\longleftrightarrow \neg$ *?ex*)
⟨*proof*⟩

lemma *wf-no-infinite-down-chainE*:
assumes *wf r*
obtains *k* **where** $(f\ (\textit{Suc}\ k), f\ k) \notin r$
⟨*proof*⟩

A dynamically-scoped fact for TFL

lemma *tfl-some*: $\forall P\ x.\ P\ x \longrightarrow P\ (\textit{Eps}\ P)$

⟨*proof*⟩

24.6 An aside: bounded accessible part

Finite monotone eventually stable sequences

lemma *finite-mono-remains-stable-implies-strict-prefix*:
fixes $f :: nat \Rightarrow 'a::order$
assumes S: *finite* (*range* f) *mono* f
and eq: $\forall n.\ f\ n = f\ (Suc\ n) \longrightarrow f\ (Suc\ n) = f\ (Suc\ (Suc\ n))$
shows $\exists N.\ (\forall n{\le}N.\ \forall m{\le}N.\ m < n \longrightarrow f\ m < f\ n) \wedge (\forall n{\ge}N.\ f\ N = f\ n)$
⟨*proof*⟩

lemma *finite-mono-strict-prefix-implies-finite-fixpoint*:
fixes $f :: nat \Rightarrow 'a\ set$
assumes S: $\bigwedge i.\ f\ i \subseteq S$ *finite* S
and ex: $\exists N.\ (\forall n{\le}N.\ \forall m{\le}N.\ m < n \longrightarrow f\ m \subset f\ n) \wedge (\forall n{\ge}N.\ f\ N = f\ n)$
shows $f\ (card\ S) = (\bigcup n.\ f\ n)$
⟨*proof*⟩

24.7 More on injections, bijections, and inverses

locale *bijection* =
fixes $f :: 'a \Rightarrow 'a$
assumes *bij*: *bij* f
begin

lemma *bij-inv*: *bij* (*inv* f)
⟨*proof*⟩

lemma *surj* [*simp*]: *surj* f
⟨*proof*⟩

lemma *inj*: *inj* f
⟨*proof*⟩

lemma *surj-inv* [*simp*]: *surj* (*inv* f)
⟨*proof*⟩

lemma *inj-inv*: *inj* (*inv* f)
⟨*proof*⟩

lemma *eqI*: $f\ a = f\ b \Longrightarrow a = b$
⟨*proof*⟩

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

lemma *eq-invI*: $inv\ f\ a = inv\ f\ b \Longrightarrow a = b$
⟨*proof*⟩

lemma *eq-inv-iff* [*simp*]: $inv\ f\ a = inv\ f\ b \longleftrightarrow a = b$
⟨*proof*⟩

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

lemma *inv-comp-left* [*simp*]: $inv\ f \circ f = id$
⟨*proof*⟩

lemma *inv-right* [*simp*]: $f\ (inv\ f\ a) = a$
⟨*proof*⟩

lemma *inv-comp-right* [*simp*]: $f \circ inv\ f = id$
⟨*proof*⟩

lemma *inv-left-eq-iff* [*simp*]: $inv\ f\ a = b \longleftrightarrow f\ b = a$
⟨*proof*⟩

lemma *inv-right-eq-iff* [*simp*]: $b = inv\ f\ a \longleftrightarrow f\ b = a$
⟨*proof*⟩

end

lemma *infinite-imp-bij-betw*:
assumes *infinite*: $\neg$ *finite* A
shows $\exists\, h.\ bij\text{-}betw\ h\ A\ (A - \{a\})$
⟨*proof*⟩

lemma *infinite-imp-bij-betw2*:
assumes $\neg$ *finite* A
shows $\exists\, h.\ bij\text{-}betw\ h\ A\ (A \cup \{a\})$
⟨*proof*⟩

lemma *bij-betw-inv-into-left*: $bij\text{-}betw\ f\ A\ A' \Longrightarrow a \in A \Longrightarrow inv\text{-}into\ A\ f\ (f\ a) = a$
⟨*proof*⟩

lemma *bij-betw-inv-into-right*: $bij\text{-}betw\ f\ A\ A' \Longrightarrow a' \in A' \Longrightarrow f\ (inv\text{-}into\ A\ f\ a') = a'$
⟨*proof*⟩

lemma *bij-betw-inv-into-subset*:
$bij\text{-}betw\ f\ A\ A' \Longrightarrow B \subseteq A \Longrightarrow f\ `\ B = B' \Longrightarrow bij\text{-}betw\ (inv\text{-}into\ A\ f)\ B'\ B$
⟨*proof*⟩

24.8 Specification package – Hilbertized version

lemma *exE-some*: $Ex\ P \Longrightarrow c \equiv Eps\ P \Longrightarrow P\ c$

⟨*proof*⟩

⟨*ML*⟩

end

25 Zorn's Lemma

theory *Zorn*
imports *Order-Relation Hilbert-Choice*
begin

25.1 Zorn's Lemma for the Subset Relation

25.1.1 Results that do not require an order

Let P be a binary predicate on the set A.

locale *pred-on* =
fixes A :: *'a set*
and P :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ⊏ *50*)
begin

abbreviation *Peq* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ⊑ *50*)
where $x \sqsubseteq y \equiv P^{==}\ x\ y$

A chain is a totally ordered subset of A.

definition *chain* :: *'a set* ⇒ *bool*
where *chain* $C \longleftrightarrow C \subseteq A \wedge (\forall x{\in}C.\ \forall y{\in}C.\ x \sqsubseteq y \vee y \sqsubseteq x)$

We call a chain that is a proper superset of some set X, but not necessarily a chain itself, a superchain of X.

abbreviation *superchain* :: *'a set* ⇒ *'a set* ⇒ *bool* (**infix** <*c* *50*)
where $X <c\ C \equiv$ *chain* $C \wedge X \subset C$

A maximal chain is a chain that does not have a superchain.

definition *maxchain* :: *'a set* ⇒ *bool*
where *maxchain* $C \longleftrightarrow$ *chain* $C \wedge (\nexists S.\ C <c\ S)$

We define the successor of a set to be an arbitrary superchain, if such exists, or the set itself, otherwise.

definition *suc* :: *'a set* ⇒ *'a set*
where *suc* C = (*if* ¬ *chain* C ∨ *maxchain* C *then* C *else* (*SOME* D. $C <c\ D$))

lemma *chainI* [*Pure.intro?*]: $C \subseteq A \Longrightarrow (\bigwedge x\ y.\ x \in C \Longrightarrow y \in C \Longrightarrow x \sqsubseteq y \vee y \sqsubseteq x) \Longrightarrow$ *chain* C
⟨*proof*⟩

lemma *chain-total*: $chain\ C \Longrightarrow x \in C \Longrightarrow y \in C \Longrightarrow x \sqsubseteq y \vee y \sqsubseteq x$
⟨*proof*⟩

lemma *not-chain-suc* [*simp*]: $\neg\ chain\ X \Longrightarrow suc\ X = X$
⟨*proof*⟩

lemma *maxchain-suc* [*simp*]: $maxchain\ X \Longrightarrow suc\ X = X$
⟨*proof*⟩

lemma *suc-subset*: $X \subseteq suc\ X$
⟨*proof*⟩

lemma *chain-empty* [*simp*]: $chain\ \{\}$
⟨*proof*⟩

lemma *not-maxchain-Some*: $chain\ C \Longrightarrow \neg\ maxchain\ C \Longrightarrow C <c\ (SOME\ D.\ C <c\ D)$
⟨*proof*⟩

lemma *suc-not-equals*: $chain\ C \Longrightarrow \neg\ maxchain\ C \Longrightarrow suc\ C \neq C$
⟨*proof*⟩

lemma *subset-suc*:
assumes $X \subseteq Y$
shows $X \subseteq suc\ Y$
⟨*proof*⟩

We build a set $\mathcal{C}$ that is closed under applications of *suc* and contains the union of all its subsets.

inductive-set *suc-Union-closed* ($\mathcal{C}$)
where
suc: $X \in \mathcal{C} \Longrightarrow suc\ X \in \mathcal{C}$
| *Union* [*unfolded Pow-iff*]: $X \in Pow\ \mathcal{C} \Longrightarrow \bigcup X \in \mathcal{C}$

Since the empty set as well as the set itself is a subset of every set, $\mathcal{C}$ contains at least $\{\} \in \mathcal{C}$ and $\bigcup \mathcal{C} \in \mathcal{C}$.

lemma *suc-Union-closed-empty*: $\{\} \in \mathcal{C}$
and *suc-Union-closed-Union*: $\bigcup \mathcal{C} \in \mathcal{C}$
⟨*proof*⟩

Thus closure under *suc* will hit a maximal chain eventually, as is shown below.

lemma *suc-Union-closed-induct* [*consumes 1*, *case-names suc Union*, *induct pred*: *suc-Union-closed*]:
assumes $X \in \mathcal{C}$
and $\bigwedge X.\ X \in \mathcal{C} \Longrightarrow Q\ X \Longrightarrow Q\ (suc\ X)$
and $\bigwedge X.\ X \subseteq \mathcal{C} \Longrightarrow \forall x{\in}X.\ Q\ x \Longrightarrow Q\ (\bigcup X)$
shows $Q\ X$

⟨*proof*⟩

lemma *suc-Union-closed-cases* [*consumes 1*, *case-names suc Union*, *cases pred*: *suc-Union-closed*]:
assumes $X \in \mathcal{C}$
and $\bigwedge Y.\ X = suc\ Y \Longrightarrow Y \in \mathcal{C} \Longrightarrow Q$
and $\bigwedge Y.\ X = \bigcup Y \Longrightarrow Y \subseteq \mathcal{C} \Longrightarrow Q$
shows Q
⟨*proof*⟩

On chains, *suc* yields a chain.

lemma *chain-suc*:
assumes *chain X*
shows *chain* (*suc X*)
⟨*proof*⟩

lemma *chain-sucD*:
assumes *chain X*
shows $suc\ X \subseteq A \wedge chain\ (suc\ X)$
⟨*proof*⟩

lemma *suc-Union-closed-total'*:
assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
and $*$: $\bigwedge Z.\ Z \in \mathcal{C} \Longrightarrow Z \subseteq Y \Longrightarrow Z = Y \vee suc\ Z \subseteq Y$
shows $X \subseteq Y \vee suc\ Y \subseteq X$
⟨*proof*⟩

lemma *suc-Union-closed-subsetD*:
assumes $Y \subseteq X$ **and** $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
shows $X = Y \vee suc\ Y \subseteq X$
⟨*proof*⟩

The elements of $\mathcal{C}$ are totally ordered by the subset relation.

lemma *suc-Union-closed-total*:
assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
shows $X \subseteq Y \vee Y \subseteq X$
⟨*proof*⟩

Once we hit a fixed point w.r.t. *suc*, all other elements of $\mathcal{C}$ are subsets of this fixed point.

lemma *suc-Union-closed-suc*:
assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$ **and** $suc\ Y = Y$
shows $X \subseteq Y$
⟨*proof*⟩

lemma *eq-suc-Union*:
assumes $X \in \mathcal{C}$
shows $suc\ X = X \longleftrightarrow X = \bigcup \mathcal{C}$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)

⟨*proof*⟩

lemma *suc-in-carrier*:
assumes $X \subseteq A$
shows *suc* $X \subseteq A$
⟨*proof*⟩

lemma *suc-Union-closed-in-carrier*:
assumes $X \in \mathcal{C}$
shows $X \subseteq A$
⟨*proof*⟩

All elements of $\mathcal{C}$ are chains.

lemma *suc-Union-closed-chain*:
assumes $X \in \mathcal{C}$
shows *chain* X
⟨*proof*⟩

25.1.2 Hausdorff's Maximum Principle

There exists a maximal totally ordered subset of A. (Note that we do not require A to be partially ordered.)

theorem *Hausdorff*: $\exists\, C.$ *maxchain* C
⟨*proof*⟩

Make notation $\mathcal{C}$ available again.

no-notation *suc-Union-closed* ($\mathcal{C}$)

lemma *chain-extend*: *chain* $C \Longrightarrow z \in A \Longrightarrow \forall\, x \in C.\ x \sqsubseteq z \Longrightarrow$ *chain* $(\{z\} \cup C)$
⟨*proof*⟩

lemma *maxchain-imp-chain*: *maxchain* $C \Longrightarrow$ *chain* C
⟨*proof*⟩

end

Hide constant *pred-on.suc-Union-closed*, which was just needed for the proof of Hausforff's maximum principle.

hide-const *pred-on.suc-Union-closed*

lemma *chain-mono*:
assumes $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
and *pred-on.chain* $A\ P\ C$
shows *pred-on.chain* $A\ Q\ C$
⟨*proof*⟩

25.1.3 Results for the proper subset relation

interpretation *subset*: *pred-on* A *op* $\subset$ **for** A ⟨*proof*⟩

lemma *subset-maxchain-max*:
assumes $subset.maxchain\ A\ C$
and $X \in A$
and $\bigcup C \subseteq X$
shows $\bigcup C = X$
$\langle proof \rangle$

25.1.4 Zorn's lemma

If every chain has an upper bound, then there is a maximal set.

lemma *subset-Zorn*:
assumes $\bigwedge C.\ subset.chain\ A\ C \Longrightarrow \exists\, U{\in}A.\ \forall\, X{\in}C.\ X \subseteq U$
shows $\exists\, M{\in}A.\ \forall\, X{\in}A.\ M \subseteq X \longrightarrow X = M$
$\langle proof \rangle$

Alternative version of Zorn's lemma for the subset relation.

lemma *subset-Zorn'*:
assumes $\bigwedge C.\ subset.chain\ A\ C \Longrightarrow \bigcup C \in A$
shows $\exists\, M{\in}A.\ \forall\, X{\in}A.\ M \subseteq X \longrightarrow X = M$
$\langle proof \rangle$

25.2 Zorn's Lemma for Partial Orders

Relate old to new definitions.

definition *chain-subset* :: $'a\ set\ set \Rightarrow bool$ $(chain_{\subseteq})$
where $chain_{\subseteq}\ C \longleftrightarrow (\forall\, A{\in}C.\ \forall\, B{\in}C.\ A \subseteq B \vee B \subseteq A)$

definition *chains* :: $'a\ set\ set \Rightarrow\ 'a\ set\ set\ set$
where $chains\ A = \{C.\ C \subseteq A \wedge chain_{\subseteq}\ C\}$

definition *Chains* :: $('a \times\ 'a)\ set \Rightarrow\ 'a\ set\ set$
where $Chains\ r = \{C.\ \forall\, a{\in}C.\ \forall\, b{\in}C.\ (a,\ b) \in r \vee (b,\ a) \in r\}$

lemma *chains-extend*: $c \in chains\ S \Longrightarrow z \in S \Longrightarrow \forall\, x \in c.\ x \subseteq z \Longrightarrow \{z\} \cup c \in chains\ S$
for z :: $'a\ set$
$\langle proof \rangle$

lemma *mono-Chains*: $r \subseteq s \Longrightarrow Chains\ r \subseteq Chains\ s$
$\langle proof \rangle$

lemma *chain-subset-alt-def*: $chain_{\subseteq}\ C = subset.chain\ UNIV\ C$
$\langle proof \rangle$

lemma *chains-alt-def*: $chains\ A = \{C.\ subset.chain\ A\ C\}$
$\langle proof \rangle$

lemma *Chains-subset*: $Chains\ r \subseteq \{C.\ pred\text{-}on.chain\ UNIV\ (\lambda x\ y.\ (x,\ y) \in r)\ C\}$
  $\langle proof \rangle$

lemma *Chains-subset'*:
  assumes *refl r*
  shows $\{C.\ pred\text{-}on.chain\ UNIV\ (\lambda x\ y.\ (x,\ y) \in r)\ C\} \subseteq Chains\ r$
  $\langle proof \rangle$

lemma *Chains-alt-def*:
  assumes *refl r*
  shows $Chains\ r = \{C.\ pred\text{-}on.chain\ UNIV\ (\lambda x\ y.\ (x,\ y) \in r)\ C\}$
  $\langle proof \rangle$

lemma *Zorn-Lemma*: $\forall\, C{\in}chains\ A.\ \bigcup C \in A \Longrightarrow \exists\, M{\in}A.\ \forall\, X{\in}A.\ M \subseteq X \longrightarrow X = M$
  $\langle proof \rangle$

lemma *Zorn-Lemma2*: $\forall\, C{\in}chains\ A.\ \exists\, U{\in}A.\ \forall\, X{\in}C.\ X \subseteq U \Longrightarrow \exists\, M{\in}A.\ \forall\, X{\in}A.\ M \subseteq X \longrightarrow X = M$
  $\langle proof \rangle$

Various other lemmas

lemma *chainsD*: $c \in chains\ S \Longrightarrow x \in c \Longrightarrow y \in c \Longrightarrow x \subseteq y \lor y \subseteq x$
  $\langle proof \rangle$

lemma *chainsD2*: $c \in chains\ S \Longrightarrow c \subseteq S$
  $\langle proof \rangle$

lemma *Zorns-po-lemma*:
  assumes *po*: *Partial-order r*
    and *u*: $\forall\, C{\in}Chains\ r.\ \exists\, u{\in}Field\ r.\ \forall\, a{\in}C.\ (a,\ u) \in r$
  shows $\exists\, m{\in}Field\ r.\ \forall\, a{\in}Field\ r.\ (m,\ a) \in r \longrightarrow a = m$
$\langle proof \rangle$

25.3 The Well Ordering Theorem

definition *init-seg-of* :: $(('a \times 'a)\ set \times ('a \times 'a)\ set)\ set$
  where $init\text{-}seg\text{-}of = \{(r,\ s).\ r \subseteq s \land (\forall\, a\ b\ c.\ (a,\ b) \in s \land (b,\ c) \in r \longrightarrow (a,\ b) \in r)\}$

abbreviation *initial-segment-of-syntax* :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set \Rightarrow bool$
    (**infix** *initial'-segment'-of 55*)
  where r *initial-segment-of* $s \equiv (r,\ s) \in init\text{-}seg\text{-}of$

lemma *refl-on-init-seg-of* [*simp*]: *r initial-segment-of r*
  $\langle proof \rangle$

lemma *trans-init-seg-of*:

r initial-segment-of s $\Longrightarrow$ *s initial-segment-of t* $\Longrightarrow$ *r initial-segment-of t*
⟨*proof*⟩

lemma *antisym-init-seg-of*: *r initial-segment-of s* $\Longrightarrow$ *s initial-segment-of r* $\Longrightarrow$ *r* = *s*
⟨*proof*⟩

lemma *Chains-init-seg-of-Union*: $R \in$ *Chains init-seg-of* $\Longrightarrow r{\in}R \Longrightarrow$ *r initial-segment-of* $\bigcup R$
⟨*proof*⟩

lemma *chain-subset-trans-Union*:
assumes $chain_\subseteq\ R\ \forall r{\in}R.\ trans\ r$
shows *trans* $(\bigcup R)$
⟨*proof*⟩

lemma *chain-subset-antisym-Union*:
assumes $chain_\subseteq\ R\ \forall r{\in}R.\ antisym\ r$
shows *antisym* $(\bigcup R)$
⟨*proof*⟩

lemma *chain-subset-Total-Union*:
assumes $chain_\subseteq\ R$ **and** $\forall r{\in}R.\ Total\ r$
shows *Total* $(\bigcup R)$
⟨*proof*⟩

lemma *wf-Union-wf-init-segs*:
assumes $R \in$ *Chains init-seg-of*
and $\forall r{\in}R.\ wf\ r$
shows *wf* $(\bigcup R)$
⟨*proof*⟩

lemma *initial-segment-of-Diff*: *p initial-segment-of q* $\Longrightarrow p - s$ *initial-segment-of* $q - s$
⟨*proof*⟩

lemma *Chains-inits-DiffI*: $R \in$ *Chains init-seg-of* $\Longrightarrow \{r - s\ |r.\ r \in R\} \in$ *Chains init-seg-of*
⟨*proof*⟩

theorem *well-ordering*: $\exists r{::}'a\ rel.$ *Well-order r* $\wedge$ *Field r* = *UNIV*
⟨*proof*⟩

corollary *well-order-on*: $\exists r{::}'a\ rel.$ *well-order-on A r*
⟨*proof*⟩

lemma *wfrec-def-adm*: $f \equiv$ *wfrec R F* $\Longrightarrow$ *wf R* $\Longrightarrow$ *adm-wf R F* $\Longrightarrow f = F\ f$

⟨*proof*⟩

lemma *dependent-wf-choice*:
fixes $P :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
assumes *wf R*
and *adm*: $\bigwedge f\ g\ x\ r.\ (\bigwedge z.\ (z,\ x) \in R \Longrightarrow f\ z = g\ z) \Longrightarrow P\ f\ x\ r = P\ g\ x\ r$
and *P*: $\bigwedge x\ f.\ (\bigwedge y.\ (y,\ x) \in R \Longrightarrow P\ f\ y\ (f\ y)) \Longrightarrow \exists\, r.\ P\ f\ x\ r$
shows $\exists f.\ \forall x.\ P\ f\ x\ (f\ x)$
⟨*proof*⟩

lemma (**in** *wellorder*) *dependent-wellorder-choice*:
assumes $\bigwedge r\ f\ g\ x.\ (\bigwedge y.\ y < x \Longrightarrow f\ y = g\ y) \Longrightarrow P\ f\ x\ r = P\ g\ x\ r$
and *P*: $\bigwedge x\ f.\ (\bigwedge y.\ y < x \Longrightarrow P\ f\ y\ (f\ y)) \Longrightarrow \exists\, r.\ P\ f\ x\ r$
shows $\exists f.\ \forall x.\ P\ f\ x\ (f\ x)$
⟨*proof*⟩

end

26 Well-Order Relations as Needed by Bounded Natural Functors

theory *BNF-Wellorder-Relation*
imports *Order-Relation*
begin

In this section, we develop basic concepts and results pertaining to well-order relations. Note that we consider well-order relations as *non-strict relations*, i.e., as containing the diagonals of their fields.

locale *wo-rel* =
fixes $r :: 'a\ rel$
assumes *WELL*: *Well-order r*
begin

The following context encompasses all this section. In other words, for the whole section, we consider a fixed well-order relation *r*.

abbreviation *under* **where** *under* ≡ *Order-Relation.under r*
abbreviation *underS* **where** *underS* ≡ *Order-Relation.underS r*
abbreviation *Under* **where** *Under* ≡ *Order-Relation.Under r*
abbreviation *UnderS* **where** *UnderS* ≡ *Order-Relation.UnderS r*
abbreviation *above* **where** *above* ≡ *Order-Relation.above r*
abbreviation *aboveS* **where** *aboveS* ≡ *Order-Relation.aboveS r*
abbreviation *Above* **where** *Above* ≡ *Order-Relation.Above r*
abbreviation *AboveS* **where** *AboveS* ≡ *Order-Relation.AboveS r*
abbreviation *ofilter* **where** *ofilter* ≡ *Order-Relation.ofilter r*
lemmas *ofilter-def* = *Order-Relation.ofilter-def*[*of r*]

26.1 Auxiliaries

lemma *REFL*: *Refl r*
⟨*proof*⟩

lemma *TRANS*: *trans r*
⟨*proof*⟩

lemma *ANTISYM*: *antisym r*
⟨*proof*⟩

lemma *TOTAL*: *Total r*
⟨*proof*⟩

lemma *TOTALS*: $\forall a \in Field\ r.\ \forall b \in Field\ r.\ (a,b) \in r \vee (b,a) \in r$
⟨*proof*⟩

lemma *LIN*: *Linear-order r*
⟨*proof*⟩

lemma *WF*: $wf\ (r - Id)$
⟨*proof*⟩

lemma *cases-Total*:
$\bigwedge phi\ a\ b.\ [\![\{a,b\} <= Field\ r;\ ((a,b) \in r \Longrightarrow phi\ a\ b);\ ((b,a) \in r \Longrightarrow phi\ a\ b)]\!]$
$\Longrightarrow phi\ a\ b$
⟨*proof*⟩

lemma *cases-Total3*:
$\bigwedge phi\ a\ b.\ [\![\{a,b\} \leq Field\ r;\ ((a,b) \in r - Id \vee (b,a) \in r - Id \Longrightarrow phi\ a\ b);$
$(a = b \Longrightarrow phi\ a\ b)]\!] \Longrightarrow phi\ a\ b$
⟨*proof*⟩

26.2 Well-founded induction and recursion adapted to non-strict well-order relations

Here we provide induction and recursion principles specific to *non-strict* well-order relations. Although minor variations of those for well-founded relations, they will be useful for doing away with the tediousness of having to take out the diagonal each time in order to switch to a well-founded relation.

lemma *well-order-induct*:
assumes *IND*: $\bigwedge x.\ \forall y.\ y \neq x \wedge (y,\ x) \in r \longrightarrow P\ y \Longrightarrow P\ x$
shows $P\ a$
⟨*proof*⟩

definition
$worec :: (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where

worec F $\equiv$ *wfrec* (*r* $-$ *Id*) *F*

definition
adm-wo :: (($'a \Rightarrow 'b$) $\Rightarrow$ $'a \Rightarrow 'b$) $\Rightarrow$ *bool*
where
adm-wo H $\equiv$ $\forall f\ g\ x.$ ($\forall y \in$ *underS x*. $f\ y = g\ y$) $\longrightarrow$ $H\ f\ x = H\ g\ x$

lemma *worec-fixpoint*:
assumes *ADM*: *adm-wo H*
shows *worec H* = *H* (*worec H*)
⟨*proof*⟩

26.3 The notions of maximum, minimum, supremum, successor and order filter

We define the successor *of a set*, and not of an element (the latter is of course a particular case). Also, we define the maximum *of two elements*, *max2*, and the minimum *of a set*, *minim* – we chose these variants since we consider them the most useful for well-orders. The minimum is defined in terms of the auxiliary relational operator *isMinim*. Then, supremum and successor are defined in terms of minimum as expected. The minimum is only meaningful for non-empty sets, and the successor is only meaningful for sets for which strict upper bounds exist. Order filters for well-orders are also known as "initial segments".

definition *max2* :: $'a \Rightarrow 'a \Rightarrow 'a$
where *max2 a b* $\equiv$ *if* (a,b) $\in$ *r then b else a*

definition *isMinim* :: $'a$ *set* $\Rightarrow$ $'a \Rightarrow$ *bool*
where *isMinim A b* $\equiv$ $b \in A \land (\forall a \in A.\ (b,a) \in r)$

definition *minim* :: $'a$ *set* $\Rightarrow$ $'a$
where *minim A* $\equiv$ *THE b. isMinim A b*

definition *supr* :: $'a$ *set* $\Rightarrow$ $'a$
where *supr A* $\equiv$ *minim* (*Above A*)

definition *suc* :: $'a$ *set* $\Rightarrow$ $'a$
where *suc A* $\equiv$ *minim* (*AboveS A*)

26.3.1 Properties of max2

lemma *max2-greater-among*:
assumes $a \in$ *Field r* **and** $b \in$ *Field r*
shows (a, *max2 a b*) $\in r \land$ (b, *max2 a b*) $\in r \land$ *max2 a b* $\in \{a,b\}$
⟨*proof*⟩

lemma *max2-greater*:
assumes $a \in$ *Field r* **and** $b \in$ *Field r*

shows $(a,\ max2\ a\ b) \in r \wedge (b,\ max2\ a\ b) \in r$
$\langle proof \rangle$

lemma *max2-among*:
assumes $a \in Field\ r$ **and** $b \in Field\ r$
shows $max2\ a\ b \in \{a,\ b\}$
$\langle proof \rangle$

lemma *max2-equals1*:
assumes $a \in Field\ r$ **and** $b \in Field\ r$
shows $(max2\ a\ b = a) = ((b,a) \in r)$
$\langle proof \rangle$

lemma *max2-equals2*:
assumes $a \in Field\ r$ **and** $b \in Field\ r$
shows $(max2\ a\ b = b) = ((a,b) \in r)$
$\langle proof \rangle$

26.3.2 Existence and uniqueness for isMinim and well-definedness of minim

lemma *isMinim-unique*:
assumes *MINIM*: $isMinim\ B\ a$ **and** *MINIM'*: $isMinim\ B\ a'$
shows $a = a'$
$\langle proof \rangle$

lemma *Well-order-isMinim-exists*:
assumes *SUB*: $B \leq Field\ r$ **and** *NE*: $B \neq \{\}$
shows $\exists\, b.\ isMinim\ B\ b$
$\langle proof \rangle$

lemma *minim-isMinim*:
assumes *SUB*: $B \leq Field\ r$ **and** *NE*: $B \neq \{\}$
shows $isMinim\ B\ (minim\ B)$
$\langle proof \rangle$

26.3.3 Properties of minim

lemma *minim-in*:
assumes $B \leq Field\ r$ **and** $B \neq \{\}$
shows $minim\ B \in B$
$\langle proof \rangle$

lemma *minim-inField*:
assumes $B \leq Field\ r$ **and** $B \neq \{\}$
shows $minim\ B \in Field\ r$
$\langle proof \rangle$

lemma *minim-least*:
assumes *SUB*: $B \leq Field\ r$ **and** *IN*: $b \in B$

shows $(minim\ B,\ b) \in r$
$\langle proof \rangle$

lemma *equals-minim*:
assumes *SUB*: $B \leq Field\ r$ **and** *IN*: $a \in B$ **and**
LEAST: $\bigwedge b.\ b \in B \Longrightarrow (a,b) \in r$
shows $a = minim\ B$
$\langle proof \rangle$

26.3.4 Properties of successor

lemma *suc-AboveS*:
assumes *SUB*: $B \leq Field\ r$ **and** *ABOVES*: $AboveS\ B \neq \{\}$
shows $suc\ B \in AboveS\ B$
$\langle proof \rangle$

lemma *suc-greater*:
assumes *SUB*: $B \leq Field\ r$ **and** *ABOVES*: $AboveS\ B \neq \{\}$ **and**
IN: $b \in B$
shows $suc\ B \neq b \wedge (b,suc\ B) \in r$
$\langle proof \rangle$

lemma *suc-least-AboveS*:
assumes *ABOVES*: $a \in AboveS\ B$
shows $(suc\ B,a) \in r$
$\langle proof \rangle$

lemma *suc-inField*:
assumes $B \leq Field\ r$ **and** $AboveS\ B \neq \{\}$
shows $suc\ B \in Field\ r$
$\langle proof \rangle$

lemma *equals-suc-AboveS*:
assumes *SUB*: $B \leq Field\ r$ **and** *ABV*: $a \in AboveS\ B$ **and**
MINIM: $\bigwedge a'.\ a' \in AboveS\ B \Longrightarrow (a,a') \in r$
shows $a = suc\ B$
$\langle proof \rangle$

lemma *suc-underS*:
assumes *IN*: $a \in Field\ r$
shows $a = suc\ (underS\ a)$
$\langle proof \rangle$

26.3.5 Properties of order filters

lemma *under-ofilter*:
ofilter $(under\ a)$
$\langle proof \rangle$

lemma *underS-ofilter*:

ofilter (*underS* a)
⟨*proof*⟩

lemma *Field-ofilter*:
ofilter (*Field* r)
⟨*proof*⟩

lemma *ofilter-underS-Field*:
ofilter $A = ((\exists\, a \in \mathit{Field}\ r.\ A = \mathit{underS}\ a) \vee (A = \mathit{Field}\ r))$
⟨*proof*⟩

lemma *ofilter-UNION*:
$(\bigwedge\ i.\ i \in I \Longrightarrow \mathit{ofilter}(A\ i)) \Longrightarrow \mathit{ofilter}\ (\bigcup i \in I.\ A\ i)$
⟨*proof*⟩

lemma *ofilter-under-UNION*:
assumes *ofilter* A
shows $A = (\bigcup a \in A.\ \mathit{under}\ a)$
⟨*proof*⟩

26.3.6 Other properties

lemma *ofilter-linord*:
assumes *OF1*: *ofilter* A **and** *OF2*: *ofilter* B
shows $A \leq B \vee B \leq A$
⟨*proof*⟩

lemma *ofilter-AboveS-Field*:
assumes *ofilter* A
shows $A \cup (\mathit{AboveS}\ A) = \mathit{Field}\ r$
⟨*proof*⟩

lemma *suc-ofilter-in*:
assumes *OF*: *ofilter* A **and** *ABOVE-NE*: $\mathit{AboveS}\ A \neq \{\}$ **and**
REL: $(b,\mathit{suc}\ A) \in r$ **and** *DIFF*: $b \neq \mathit{suc}\ A$
shows $b \in A$
⟨*proof*⟩

end

end

27 Well-Order Embeddings as Needed by Bounded Natural Functors

theory *BNF-Wellorder-Embedding*
imports *Hilbert-Choice BNF-Wellorder-Relation*
begin

In this section, we introduce well-order *embeddings* and *isomorphisms* and prove their basic properties. The notion of embedding is considered from the point of view of the theory of ordinals, and therefore requires the source to be injected as an *initial segment* (i.e., *order filter*) of the target. A main result of this section is the existence of embeddings (in one direction or another) between any two well-orders, having as a consequence the fact that, given any two sets on any two types, one is smaller than (i.e., can be injected into) the other.

27.1 Auxiliaries

lemma *UNION-inj-on-ofilter*:
assumes *WELL*: *Well-order r* **and**
OF: $\bigwedge i.\ i \in I \Longrightarrow$ *wo-rel.ofilter r* $(A\ i)$ **and**
INJ: $\bigwedge i.\ i \in I \Longrightarrow$ *inj-on f* $(A\ i)$
shows *inj-on f* $(\bigcup i \in I.\ A\ i)$
⟨*proof*⟩

lemma *under-underS-bij-betw*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
IN: $a \in$ *Field r* **and** *IN'*: $f\ a \in$ *Field r'* **and**
BIJ: *bij-betw f* (*underS r a*) (*underS r'* $(f\ a)$)
shows *bij-betw f* (*under r a*) (*under r'* $(f\ a)$)
⟨*proof*⟩

27.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions

Standardly, a function is an embedding of a well-order in another if it injectively and order-compatibly maps the former into an order filter of the latter. Here we opt for a more succinct definition (operator *embed*), asking that, for any element in the source, the function should be a bijection between the set of strict lower bounds of that element and the set of strict lower bounds of its image. (Later we prove equivalence with the standard definition – lemma *embed-iff-compat-inj-on-ofilter*.) A *strict embedding* (operator *embedS*) is a non-bijective embedding and an isomorphism (operator *iso*) is a bijective embedding.

definition *embed* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where
embed r r' f $\equiv \forall a \in$ *Field r*. *bij-betw f* (*under r a*) (*under r'* $(f\ a)$)

lemmas *embed-defs* = *embed-def embed-def*[*abs-def*]

Strict embeddings:

definition *embedS* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where

embedS r r' f $\equiv$ *embed* r r' f $\wedge$ $\neg$ *bij-betw* f (*Field* r) (*Field* r')

lemmas *embedS-defs* = *embedS-def embedS-def*[*abs-def*]

definition *iso* :: $'a$ *rel* $\Rightarrow$ $'a'$ *rel* $\Rightarrow$ $('a \Rightarrow 'a')$ $\Rightarrow$ *bool*
where
iso r r' f $\equiv$ *embed* r r' f $\wedge$ *bij-betw* f (*Field* r) (*Field* r')

lemmas *iso-defs* = *iso-def iso-def*[*abs-def*]

definition *compat* :: $'a$ *rel* $\Rightarrow$ $'a'$ *rel* $\Rightarrow$ $('a \Rightarrow 'a')$ $\Rightarrow$ *bool*
where
compat r r' f $\equiv$ $\forall$ a b. $(a,b) \in r \longrightarrow (f\ a,\ f\ b) \in r'$

lemma *compat-wf*:
assumes *CMP*: *compat* r r' f **and** *WF*: *wf* r'
shows *wf* r
⟨*proof*⟩

lemma *id-embed*: *embed* r r *id*
⟨*proof*⟩

lemma *id-iso*: *iso* r r *id*
⟨*proof*⟩

lemma *embed-in-Field*:
assumes *WELL*: *Well-order* r **and**
EMB: *embed* r r' f **and** *IN*: $a \in$ *Field* r
shows $f\ a \in$ *Field* r'
⟨*proof*⟩

lemma *comp-embed*:
assumes *WELL*: *Well-order* r **and**
EMB: *embed* r r' f **and** *EMB'*: *embed* r' r'' f'
shows *embed* r r'' $(f'\ o\ f)$
⟨*proof*⟩

lemma *comp-iso*:
assumes *WELL*: *Well-order* r **and**
EMB: *iso* r r' f **and** *EMB'*: *iso* r' r'' f'
shows *iso* r r'' $(f'\ o\ f)$
⟨*proof*⟩

That *embedS* is also preserved by function composition shall be proved only later.

lemma *embed-Field*:
⟦*Well-order* r; *embed* r r' f⟧ $\Longrightarrow$ f‘(*Field* r) $\leq$ *Field* r'
⟨*proof*⟩

lemma *embed-preserves-ofilter*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
EMB: *embed r r' f* **and** *OF*: *wo-rel.ofilter r A*
shows *wo-rel.ofilter r'* (*f'A*)
⟨*proof*⟩

lemma *embed-Field-ofilter*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
EMB: *embed r r' f*
shows *wo-rel.ofilter r'* (*f'*(*Field r*))
⟨*proof*⟩

lemma *embed-compat*:
assumes *EMB*: *embed r r' f*
shows *compat r r' f*
⟨*proof*⟩

lemma *embed-inj-on*:
assumes *WELL*: *Well-order r* **and** *EMB*: *embed r r' f*
shows *inj-on f* (*Field r*)
⟨*proof*⟩

lemma *embed-underS*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
EMB: *embed r r' f* **and** *IN*: $a \in$ *Field r*
shows *bij-betw f* (*underS r a*) (*underS r'* (*f a*))
⟨*proof*⟩

lemma *embed-iff-compat-inj-on-ofilter*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
shows *embed r r' f* = (*compat r r' f* $\wedge$ *inj-on f* (*Field r*) $\wedge$ *wo-rel.ofilter r'* (*f'*(*Field r*)))
⟨*proof*⟩

lemma *inv-into-ofilter-embed*:
assumes *WELL*: *Well-order r* **and** *OF*: *wo-rel.ofilter r A* **and**
BIJ: $\forall$ *b* $\in$ *A*. *bij-betw f* (*under r b*) (*under r'* (*f b*)) **and**
IMAGE: *f* ' *A* = *Field r'*
shows *embed r' r* (*inv-into A f*)
⟨*proof*⟩

lemma *inv-into-underS-embed*:
assumes *WELL*: *Well-order r* **and**
BIJ: $\forall$ *b* $\in$ *underS r a*. *bij-betw f* (*under r b*) (*under r'* (*f b*)) **and**
IN: $a \in$ *Field r* **and**
IMAGE: *f* ' (*underS r a*) = *Field r'*
shows *embed r' r* (*inv-into* (*underS r a*) *f*)
⟨*proof*⟩

lemma *inv-into-Field-embed*:
assumes *WELL*: *Well-order r* **and** *EMB*: *embed r r' f* **and**
IMAGE: *Field r'* $\leq$ *f ' (Field r)*
shows *embed r' r (inv-into (Field r) f)*
⟨*proof*⟩

lemma *inv-into-Field-embed-bij-betw*:
assumes *WELL*: *Well-order r* **and**
EMB: *embed r r' f* **and** *BIJ*: *bij-betw f (Field r) (Field r')*
shows *embed r' r (inv-into (Field r) f)*
⟨*proof*⟩

27.3 Given any two well-orders, one can be embedded in the other

Here is an overview of the proof of of this fact, stated in theorem *wellorders-totally-ordered*:

Fix the well-orders $r{::}'a\ rel$ and $r'{::}'a'\ rel$. Attempt to define an embedding $f{::}'a \Rightarrow 'a'$ from r to r' in the natural way by well-order recursion ("hoping" that *Field r* turns out to be smaller than *Field r'*), but also record, at the recursive step, in a function $g{::}'a \Rightarrow bool$, the extra information of whether *Field r'* gets exhausted or not.

If *Field r'* does not get exhausted, then *Field r* is indeed smaller and f is the desired embedding from r to r' (lemma *wellorders-totally-ordered-aux*).

Otherwise, it means that *Field r'* is the smaller one, and the inverse of (the "good" segment of) f is the desired embedding from r' to r (lemma *wellorders-totally-ordered-aux2*).

lemma *wellorders-totally-ordered-aux*:
fixes $r\ {::}'a\ rel$ **and** $r'{::}'a'\ rel$ **and**
$f\ {::}\ 'a \Rightarrow 'a'$ **and** $a{::}'a$
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and** *IN*: $a \in$ *Field r*
and
IH: $\forall b \in$ *underS r a. bij-betw f (under r b) (under r' (f b))* **and**
NOT: *f ' (underS r a)* $\neq$ *Field r'* **and** *SUC*: *f a = wo-rel.suc r' (f'(underS r a))*
shows *bij-betw f (under r a) (under r' (f a))*
⟨*proof*⟩

lemma *wellorders-totally-ordered-aux2*:
fixes $r\ {::}'a\ rel$ **and** $r'{::}'a'\ rel$ **and**
$f\ {::}\ 'a \Rightarrow 'a'$ **and** $g\ {::}\ 'a \Rightarrow bool$ **and** $a{::}'a$
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
MAIN1:
$\bigwedge a.\ (False \notin g`(underS\ r\ a) \wedge f`(underS\ r\ a) \neq Field\ r'$
$\longrightarrow f\ a = wo\text{-}rel.suc\ r'\ (f`(underS\ r\ a)) \wedge g\ a = True)$
$\wedge$
$(\neg(False \notin (g`(underS\ r\ a)) \wedge f`(underS\ r\ a) \neq Field\ r')$
$\longrightarrow g\ a = False)$ **and**

MAIN2: $\bigwedge$ *a*. $a \in$ *Field r* $\wedge$ *False* $\notin$ *g*‘(*under r a*) $\longrightarrow$
bij-betw f (*under r a*) (*under r*$'$ (*f a*)) **and**
Case: $a \in$ *Field r* $\wedge$ *False* $\in$ *g*‘(*under r a*)
shows $\exists f'$. *embed r*$'$ *r f*$'$
⟨*proof*⟩

theorem *wellorders-totally-ordered*:
fixes *r* ::$'$*a rel* **and** *r*$'$::$'$*a*$'$ *rel*
assumes *WELL*: *Well-order r* **and** *WELL*$'$: *Well-order r*$'$
shows ($\exists f$. *embed r r*$'$ *f*) $\vee$ ($\exists f'$. *embed r*$'$ *r f*$'$)
⟨*proof*⟩

27.4 Uniqueness of embeddings

Here we show a fact complementary to the one from the previous subsection – namely, that between any two well-orders there is *at most* one embedding, and is the one definable by the expected well-order recursive equation. As a consequence, any two embeddings of opposite directions are mutually inverse.

lemma *embed-determined*:
assumes *WELL*: *Well-order r* **and** *WELL*$'$: *Well-order r*$'$ **and**
EMB: *embed r r*$'$ *f* **and** *IN*: $a \in$ *Field r*
shows *f a* = *wo-rel.suc r*$'$ (*f*‘(*underS r a*))
⟨*proof*⟩

lemma *embed-unique*:
assumes *WELL*: *Well-order r* **and** *WELL*$'$: *Well-order r*$'$ **and**
EMBf: *embed r r*$'$ *f* **and** *EMBg*: *embed r r*$'$ *g*
shows $a \in$ *Field r* $\longrightarrow$ *f a* = *g a*
⟨*proof*⟩

lemma *embed-bothWays-inverse*:
assumes *WELL*: *Well-order r* **and** *WELL*$'$: *Well-order r*$'$ **and**
EMB: *embed r r*$'$ *f* **and** *EMB*$'$: *embed r*$'$ *r f*$'$
shows ($\forall a \in$ *Field r*. $f'(f\ a) = a$) $\wedge$ ($\forall a' \in$ *Field r*$'$. $f(f'\ a') = a'$)
⟨*proof*⟩

lemma *embed-bothWays-bij-betw*:
assumes *WELL*: *Well-order r* **and** *WELL*$'$: *Well-order r*$'$ **and**
EMB: *embed r r*$'$ *f* **and** *EMB*$'$: *embed r*$'$ *r g*
shows *bij-betw f* (*Field r*) (*Field r*$'$)
⟨*proof*⟩

lemma *embed-bothWays-iso*:
assumes *WELL*: *Well-order r* **and** *WELL*$'$: *Well-order r*$'$ **and**
EMB: *embed r r*$'$ *f* **and** *EMB*$'$: *embed r*$'$ *r g*
shows *iso r r*$'$ *f*
⟨*proof*⟩

27.5 More properties of embeddings, strict embeddings and isomorphisms

lemma *embed-bothWays-Field-bij-betw*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
EMB: *embed r r' f* **and** *EMB'*: *embed r' r f'*
shows *bij-betw f (Field r) (Field r')*
⟨*proof*⟩

lemma *embedS-comp-embed*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and** *WELL''*: *Well-order r''*
and *EMB*: *embedS r r' f* **and** *EMB'*: *embed r' r'' f'*
shows *embedS r r'' (f' o f)*
⟨*proof*⟩

lemma *embed-comp-embedS*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and** *WELL''*: *Well-order r''*
and *EMB*: *embed r r' f* **and** *EMB'*: *embedS r' r'' f'*
shows *embedS r r'' (f' o f)*
⟨*proof*⟩

lemma *embed-comp-iso*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and** *WELL''*: *Well-order r''*
and *EMB*: *embed r r' f* **and** *EMB'*: *iso r' r'' f'*
shows *embed r r'' (f' o f)*
⟨*proof*⟩

lemma *iso-comp-embed*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and** *WELL''*: *Well-order r''*
and *EMB*: *iso r r' f* **and** *EMB'*: *embed r' r'' f'*
shows *embed r r'' (f' o f)*
⟨*proof*⟩

lemma *embedS-comp-iso*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and** *WELL''*: *Well-order r''*
and *EMB*: *embedS r r' f* **and** *EMB'*: *iso r' r'' f'*
shows *embedS r r'' (f' o f)*
⟨*proof*⟩

lemma *iso-comp-embedS*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and** *WELL''*: *Well-order r''*
and *EMB*: *iso r r' f* **and** *EMB'*: *embedS r' r'' f'*
shows *embedS r r'' (f' o f)*
⟨*proof*⟩

lemma *embedS-Field*:
assumes *WELL*: *Well-order r* **and** *EMB*: *embedS r r′ f*
shows f ‘ (*Field r*) < *Field r′*
⟨*proof*⟩

lemma *embedS-iff*:
assumes *WELL*: *Well-order r* **and** *ISO*: *embed r r′ f*
shows *embedS r r′ f* = (f ‘ (*Field r*) < *Field r′*)
⟨*proof*⟩

lemma *iso-Field*:
iso r r′ f ⟹ f ‘ (*Field r*) = *Field r′*
⟨*proof*⟩

lemma *iso-iff*:
assumes *Well-order r*
shows *iso r r′ f* = (*embed r r′ f* ∧ f ‘ (*Field r*) = *Field r′*)
⟨*proof*⟩

lemma *iso-iff2*:
assumes *Well-order r*
shows *iso r r′ f* = (*bij-betw f* (*Field r*) (*Field r′*) ∧
(∀ *a* ∈ *Field r*. ∀ *b* ∈ *Field r*.
(((*a*,*b*) ∈ *r*) = ((*f a*, *f b*) ∈ *r′*))))
⟨*proof*⟩

lemma *iso-iff3*:
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′*
shows *iso r r′ f* = (*bij-betw f* (*Field r*) (*Field r′*) ∧ *compat r r′ f*)
⟨*proof*⟩

end

28 Constructions on Wellorders as Needed by Bounded Natural Functors

theory *BNF-Wellorder-Constructions*
imports *BNF-Wellorder-Embedding*
begin

In this section, we study basic constructions on well-orders, such as restriction to a set/order filter, copy via direct images, ordinal-like sum of disjoint well-orders, and bounded square. We also define between well-orders the relations *ordLeq*, of being embedded (abbreviated $\leq o$), *ordLess*, of being strictly embedded (abbreviated $<o$), and *ordIso*, of being isomorphic (abbreviated $=o$). We study the connections between these relations, order filters, and the aforementioned constructions. A main result of this section

is that $<o$ is well-founded.

28.1 Restriction to a set

abbreviation *Restr* :: $'a\ rel \Rightarrow 'a\ set \Rightarrow 'a\ rel$
where $Restr\ r\ A \equiv r\ Int\ (A \times A)$

lemma *Restr-subset*:
$A \leq B \Longrightarrow Restr\ (Restr\ r\ B)\ A = Restr\ r\ A$
⟨*proof*⟩

lemma *Restr-Field*: $Restr\ r\ (Field\ r) = r$
⟨*proof*⟩

lemma *Refl-Restr*: $Refl\ r \Longrightarrow Refl(Restr\ r\ A)$
⟨*proof*⟩

lemma *linear-order-on-Restr*:
$linear\text{-}order\text{-}on\ A\ r \Longrightarrow linear\text{-}order\text{-}on\ (A \cap above\ r\ x)\ (Restr\ r\ (above\ r\ x))$
⟨*proof*⟩

lemma *antisym-Restr*:
$antisym\ r \Longrightarrow antisym(Restr\ r\ A)$
⟨*proof*⟩

lemma *Total-Restr*:
$Total\ r \Longrightarrow Total(Restr\ r\ A)$
⟨*proof*⟩

lemma *trans-Restr*:
$trans\ r \Longrightarrow trans(Restr\ r\ A)$
⟨*proof*⟩

lemma *Preorder-Restr*:
$Preorder\ r \Longrightarrow Preorder(Restr\ r\ A)$
⟨*proof*⟩

lemma *Partial-order-Restr*:
$Partial\text{-}order\ r \Longrightarrow Partial\text{-}order(Restr\ r\ A)$
⟨*proof*⟩

lemma *Linear-order-Restr*:
$Linear\text{-}order\ r \Longrightarrow Linear\text{-}order(Restr\ r\ A)$
⟨*proof*⟩

lemma *Well-order-Restr*:
assumes $Well\text{-}order\ r$
shows $Well\text{-}order(Restr\ r\ A)$
⟨*proof*⟩

lemma *Field-Restr-subset*: $Field(Restr\ r\ A) \leq A$
⟨*proof*⟩

lemma *Refl-Field-Restr*:
$Refl\ r \Longrightarrow Field(Restr\ r\ A) = (Field\ r)\ Int\ A$
⟨*proof*⟩

lemma *Refl-Field-Restr2*:
$[\![Refl\ r;\ A \leq Field\ r]\!] \Longrightarrow Field(Restr\ r\ A) = A$
⟨*proof*⟩

lemma *well-order-on-Restr*:
assumes *WELL*: *Well-order r* **and** *SUB*: $A \leq Field\ r$
shows *well-order-on A* (*Restr r A*)
⟨*proof*⟩

28.2 Order filters versus restrictions and embeddings

lemma *Field-Restr-ofilter*:
$[\![Well\text{-}order\ r;\ wo\text{-}rel.ofilter\ r\ A]\!] \Longrightarrow Field(Restr\ r\ A) = A$
⟨*proof*⟩

lemma *ofilter-Restr-under*:
assumes *WELL*: *Well-order r* **and** *OF*: *wo-rel.ofilter r A* **and** *IN*: $a \in A$
shows *under* (*Restr r A*) *a* = *under r a*
⟨*proof*⟩

lemma *ofilter-embed*:
assumes *Well-order r*
shows *wo-rel.ofilter r A* = $(A \leq Field\ r \wedge embed\ (Restr\ r\ A)\ r\ id)$
⟨*proof*⟩

lemma *ofilter-Restr-Int*:
assumes *WELL*: *Well-order r* **and** *OFA*: *wo-rel.ofilter r A*
shows *wo-rel.ofilter* (*Restr r B*) (*A Int B*)
⟨*proof*⟩

lemma *ofilter-Restr-subset*:
assumes *WELL*: *Well-order r* **and** *OFA*: *wo-rel.ofilter r A* **and** *SUB*: $A \leq B$
shows *wo-rel.ofilter* (*Restr r B*) *A*
⟨*proof*⟩

lemma *ofilter-subset-embed*:
assumes *WELL*: *Well-order r* **and**
OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*
shows $(A \leq B) = (embed\ (Restr\ r\ A)\ (Restr\ r\ B)\ id)$
⟨*proof*⟩

lemma *ofilter-subset-embedS-iso*:
assumes *WELL*: *Well-order r* **and**
OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*
shows $((A < B) = (embedS\ (Restr\ r\ A)\ (Restr\ r\ B)\ id)) \wedge$
$((A = B) = (iso\ (Restr\ r\ A)\ (Restr\ r\ B)\ id))$
⟨*proof*⟩

lemma *ofilter-subset-embedS*:
assumes *WELL*: *Well-order r* **and**
OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*
shows $(A < B) = embedS\ (Restr\ r\ A)\ (Restr\ r\ B)\ id$
⟨*proof*⟩

lemma *embed-implies-iso-Restr*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
EMB: *embed r' r f*
shows *iso r'* $(Restr\ r\ (f\ `\ (Field\ r')))\ f$
⟨*proof*⟩

28.3 The strict inclusion on proper ofilters is well-founded

definition *ofilterIncl* :: *'a rel* ⇒ *'a set rel*
where
ofilterIncl r ≡ {(*A*,*B*). *wo-rel.ofilter r A* ∧ *A* ≠ *Field r* ∧
wo-rel.ofilter r B ∧ *B* ≠ *Field r* ∧ *A* < *B*}

lemma *wf-ofilterIncl*:
assumes *WELL*: *Well-order r*
shows *wf*(*ofilterIncl r*)
⟨*proof*⟩

28.4 Ordering the well-orders by existence of embeddings

We define three relations between well-orders:

- *ordLeq*, of being embedded (abbreviated ≤*o*);
- *ordLess*, of being strictly embedded (abbreviated <*o*);
- *ordIso*, of being isomorphic (abbreviated =*o*).

The prefix "ord" and the index "o" in these names stand for "ordinal-like". These relations shall be proved to be inter-connected in a similar fashion as the trio ≤, <, = associated to a total order on a set.

definition *ordLeq* :: (*'a rel* ∗ *'a' rel*) *set*
where
ordLeq = {(*r*,*r'*). *Well-order r* ∧ *Well-order r'* ∧ (∃*f*. *embed r r' f*)}

abbreviation *ordLeq2* :: *'a rel* ⇒ *'a' rel* ⇒ *bool* (**infix** <=*o 50*)

where $r <=o\ r' \equiv (r,r') \in ordLeq$

abbreviation *ordLeq3* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow bool$ (**infix** $\leq o$ *50*)
where $r \leq o\ r' \equiv r <=o\ r'$

definition *ordLess* :: $('a\ rel * 'a'\ rel)\ set$
where
$ordLess = \{(r,r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ embedS\ r\ r'\ f)\}$

abbreviation *ordLess2* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow bool$ (**infix** $<o$ *50*)
where $r <o\ r' \equiv (r,r') \in ordLess$

definition *ordIso* :: $('a\ rel * 'a'\ rel)\ set$
where
$ordIso = \{(r,r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ iso\ r\ r'\ f)\}$

abbreviation *ordIso2* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow bool$ (**infix** $=o$ *50*)
where $r =o\ r' \equiv (r,r') \in ordIso$

lemmas *ordRels-def* = *ordLeq-def ordLess-def ordIso-def*

lemma *ordLeq-Well-order-simp*:
assumes $r \leq o\ r'$
shows $Well\text{-}order\ r \wedge Well\text{-}order\ r'$
⟨*proof*⟩

Notice that the relations $\leq o$, $<o$, $=o$ connect well-orders on potentially *distinct* types. However, some of the lemmas below, including the next one, restrict implicitly the type of these relations to $(('a\ rel) * ('a\ rel))\ set$, i.e., to $'a\ rel\ rel$.

lemma *ordLeq-reflexive*:
$Well\text{-}order\ r \Longrightarrow r \leq o\ r$
⟨*proof*⟩

lemma *ordLeq-transitive*[*trans*]:
assumes $*$: $r \leq o\ r'$ **and** $**$: $r' \leq o\ r''$
shows $r \leq o\ r''$
⟨*proof*⟩

lemma *ordLeq-total*:
$[\![Well\text{-}order\ r;\ Well\text{-}order\ r']\!] \Longrightarrow r \leq o\ r' \vee r' \leq o\ r$
⟨*proof*⟩

lemma *ordIso-reflexive*:
$Well\text{-}order\ r \Longrightarrow r =o\ r$
⟨*proof*⟩

lemma *ordIso-transitive*[*trans*]:
assumes $*$: $r =o\ r'$ **and** $**$: $r' =o\ r''$

shows $r =o\ r''$
⟨*proof*⟩

lemma *ordIso-symmetric*:
assumes $*$: $r =o\ r'$
shows $r' =o\ r$
⟨*proof*⟩

lemma *ordLeq-ordLess-trans*[*trans*]:
assumes $r \le o\ r'$ **and** $r' <o\ r''$
shows $r <o\ r''$
⟨*proof*⟩

lemma *ordLess-ordLeq-trans*[*trans*]:
assumes $r <o\ r'$ **and** $r' \le o\ r''$
shows $r <o\ r''$
⟨*proof*⟩

lemma *ordLeq-ordIso-trans*[*trans*]:
assumes $r \le o\ r'$ **and** $r' =o\ r''$
shows $r \le o\ r''$
⟨*proof*⟩

lemma *ordIso-ordLeq-trans*[*trans*]:
assumes $r =o\ r'$ **and** $r' \le o\ r''$
shows $r \le o\ r''$
⟨*proof*⟩

lemma *ordLess-ordIso-trans*[*trans*]:
assumes $r <o\ r'$ **and** $r' =o\ r''$
shows $r <o\ r''$
⟨*proof*⟩

lemma *ordIso-ordLess-trans*[*trans*]:
assumes $r =o\ r'$ **and** $r' <o\ r''$
shows $r <o\ r''$
⟨*proof*⟩

lemma *ordLess-not-embed*:
assumes $r <o\ r'$
shows $\neg(\exists f'.\ embed\ r'\ r\ f')$
⟨*proof*⟩

lemma *ordLess-Field*:
assumes *OL*: $r1 <o\ r2$ **and** *EMB*: $embed\ r1\ r2\ f$
shows $\neg\ (f`(Field\ r1) = Field\ r2)$
⟨*proof*⟩

lemma *ordLess-iff*:

$r <o\ r' = (\textit{Well-order}\ r \wedge \textit{Well-order}\ r' \wedge \neg(\exists f'.\ \textit{embed}\ r'\ r\ f'))$
⟨*proof*⟩

lemma *ordLess-irreflexive*: $\neg\ r <o\ r$
⟨*proof*⟩

lemma *ordLeq-iff-ordLess-or-ordIso*:
$r \leq o\ r' = (r <o\ r' \vee r =o\ r')$
⟨*proof*⟩

lemma *ordIso-iff-ordLeq*:
$(r =o\ r') = (r \leq o\ r' \wedge r' \leq o\ r)$
⟨*proof*⟩

lemma *not-ordLess-ordLeq*:
$r <o\ r' \Longrightarrow \neg\ r' \leq o\ r$
⟨*proof*⟩

lemma *ordLess-or-ordLeq*:
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $r <o\ r' \vee r' \leq o\ r$
⟨*proof*⟩

lemma *not-ordLess-ordIso*:
$r <o\ r' \Longrightarrow \neg\ r =o\ r'$
⟨*proof*⟩

lemma *not-ordLeq-iff-ordLess*:
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $(\neg\ r' \leq o\ r) = (r <o\ r')$
⟨*proof*⟩

lemma *not-ordLess-iff-ordLeq*:
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $(\neg\ r' <o\ r) = (r \leq o\ r')$
⟨*proof*⟩

lemma *ordLess-transitive*[*trans*]:
$[\![r <o\ r';\ r' <o\ r'']\!] \Longrightarrow r <o\ r''$
⟨*proof*⟩

corollary *ordLess-trans*: *trans ordLess*
⟨*proof*⟩

lemmas *ordIso-equivalence* = *ordIso-transitive ordIso-reflexive ordIso-symmetric*

lemma *ordIso-imp-ordLeq*:
$r =o\ r' \Longrightarrow r \leq o\ r'$
⟨*proof*⟩

lemma *ordLess-imp-ordLeq*:
$r <o\ r' \Longrightarrow r \leq o\ r'$
⟨*proof*⟩

lemma *ofilter-subset-ordLeq*:
assumes *WELL*: *Well-order r* **and**
OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*
shows $(A \leq B) = (Restr\ r\ A \leq o\ Restr\ r\ B)$
⟨*proof*⟩

lemma *ofilter-subset-ordLess*:
assumes *WELL*: *Well-order r* **and**
OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*
shows $(A < B) = (Restr\ r\ A <o\ Restr\ r\ B)$
⟨*proof*⟩

lemma *ofilter-ordLess*:
$[\![Well\text{-}order\ r;\ wo\text{-}rel.ofilter\ r\ A]\!] \Longrightarrow (A < Field\ r) = (Restr\ r\ A <o\ r)$
⟨*proof*⟩

corollary *underS-Restr-ordLess*:
assumes *Well-order r* **and** $Field\ r \neq \{\}$
shows $Restr\ r\ (underS\ r\ a) <o\ r$
⟨*proof*⟩

lemma *embed-ordLess-ofilterIncl*:
assumes
OL12: $r1 <o\ r2$ **and** *OL23*: $r2 <o\ r3$ **and**
EMB13: *embed r1 r3 f13* **and** *EMB23*: *embed r2 r3 f23*
shows $(f13`(Field\ r1),\ f23`(Field\ r2)) \in (ofilterIncl\ r3)$
⟨*proof*⟩

lemma *ordLess-iff-ordIso-Restr*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
shows $(r' <o\ r) = (\exists\, a \in Field\ r.\ r' =o\ Restr\ r\ (underS\ r\ a))$
⟨*proof*⟩

lemma *internalize-ordLess*:
$(r' <o\ r) = (\exists\, p.\ Field\ p < Field\ r \wedge r' =o\ p \wedge p <o\ r)$
⟨*proof*⟩

lemma *internalize-ordLeq*:
$(r' \leq o\ r) = (\exists\, p.\ Field\ p \leq Field\ r \wedge r' =o\ p \wedge p \leq o\ r)$
⟨*proof*⟩

lemma *ordLeq-iff-ordLess-Restr*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
shows $(r \leq o\ r') = (\forall\, a \in Field\ r.\ Restr\ r\ (underS\ r\ a) <o\ r')$

⟨*proof*⟩

lemma *finite-ordLess-infinite*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
FIN: *finite*(*Field r*) **and** *INF*: ¬*finite*(*Field r'*)
shows $r <o\ r'$
⟨*proof*⟩

lemma *finite-well-order-on-ordIso*:
assumes *FIN*: *finite A* **and**
WELL: *well-order-on A r* **and** *WELL'*: *well-order-on A r'*
shows $r =o\ r'$
⟨*proof*⟩

28.5 $<o$ is well-founded

Of course, it only makes sense to state that the $<o$ is well-founded on the restricted type *'a rel rel*. We prove this by first showing that, for any set of well-orders all embedded in a fixed well-order, the function mapping each well-order in the set to an order filter of the fixed well-order is compatible w.r.t. to $<o$ versus *strict inclusion*; and we already know that strict inclusion of order filters is well-founded.

definition *ord-to-filter* :: *'a rel* ⇒ *'a rel* ⇒ *'a set*
where *ord-to-filter r0 r* ≡ (*SOME f. embed r r0 f*) ‘ (*Field r*)

lemma *ord-to-filter-compat*:
compat (*ordLess Int* (*ordLess*^−*1*‘‘{*r0*} × *ordLess*^−*1*‘‘{*r0*}))
(*ofilterIncl r0*)
(*ord-to-filter r0*)
⟨*proof*⟩

theorem *wf-ordLess*: *wf ordLess*
⟨*proof*⟩

corollary *exists-minim-Well-order*:
assumes *NE*: $R \neq \{\}$ **and** *WELL*: $\forall r \in R.$ *Well-order r*
shows $\exists r \in R.\ \forall r' \in R.\ r \leq o\ r'$
⟨*proof*⟩

28.6 Copy via direct images

The direct image operator is the dual of the inverse image operator *inv-image* from *Relation.thy*. It is useful for transporting a well-order between different types.

definition *dir-image* :: *'a rel* ⇒ (*'a* ⇒ *'a'*) ⇒ *'a' rel*
where
dir-image r f = {(*f a*, *f b*)| *a b*. (*a*,*b*) ∈ *r*}

lemma *dir-image-Field*:
Field(*dir-image r f*) = *f* ‘ (*Field r*)
⟨*proof*⟩

lemma *dir-image-minus-Id*:
inj-on f (*Field r*) ⟹ (*dir-image r f*) − *Id* = *dir-image* (*r* − *Id*) *f*
⟨*proof*⟩

lemma *Refl-dir-image*:
assumes *Refl r*
shows *Refl*(*dir-image r f*)
⟨*proof*⟩

lemma *trans-dir-image*:
assumes *TRANS*: *trans r* **and** *INJ*: *inj-on f* (*Field r*)
shows *trans*(*dir-image r f*)
⟨*proof*⟩

lemma *Preorder-dir-image*:
⟦*Preorder r*; *inj-on f* (*Field r*)⟧ ⟹ *Preorder* (*dir-image r f*)
⟨*proof*⟩

lemma *antisym-dir-image*:
assumes *AN*: *antisym r* **and** *INJ*: *inj-on f* (*Field r*)
shows *antisym*(*dir-image r f*)
⟨*proof*⟩

lemma *Partial-order-dir-image*:
⟦*Partial-order r*; *inj-on f* (*Field r*)⟧ ⟹ *Partial-order* (*dir-image r f*)
⟨*proof*⟩

lemma *Total-dir-image*:
assumes *TOT*: *Total r* **and** *INJ*: *inj-on f* (*Field r*)
shows *Total*(*dir-image r f*)
⟨*proof*⟩

lemma *Linear-order-dir-image*:
⟦*Linear-order r*; *inj-on f* (*Field r*)⟧ ⟹ *Linear-order* (*dir-image r f*)
⟨*proof*⟩

lemma *wf-dir-image*:
assumes *WF*: *wf r* **and** *INJ*: *inj-on f* (*Field r*)
shows *wf*(*dir-image r f*)
⟨*proof*⟩

lemma *Well-order-dir-image*:
⟦*Well-order r*; *inj-on f* (*Field r*)⟧ ⟹ *Well-order* (*dir-image r f*)
⟨*proof*⟩

lemma *dir-image-bij-betw*:
$[\![inj\text{-}on\ f\ (Field\ r)]\!] \Longrightarrow bij\text{-}betw\ f\ (Field\ r)\ (Field\ (dir\text{-}image\ r\ f))$
⟨*proof*⟩

lemma *dir-image-compat*:
$compat\ r\ (dir\text{-}image\ r\ f)\ f$
⟨*proof*⟩

lemma *dir-image-iso*:
$[\![Well\text{-}order\ r;\ inj\text{-}on\ f\ (Field\ r)]\!] \Longrightarrow iso\ r\ (dir\text{-}image\ r\ f)\ f$
⟨*proof*⟩

lemma *dir-image-ordIso*:
$[\![Well\text{-}order\ r;\ inj\text{-}on\ f\ (Field\ r)]\!] \Longrightarrow r =o\ dir\text{-}image\ r\ f$
⟨*proof*⟩

lemma *Well-order-iso-copy*:
assumes *WELL*: $well\text{-}order\text{-}on\ A\ r$ **and** *BIJ*: $bij\text{-}betw\ f\ A\ A'$
shows $\exists\, r'.\ well\text{-}order\text{-}on\ A'\ r' \wedge r =o\ r'$
⟨*proof*⟩

28.7 Bounded square

This construction essentially defines, for an order relation r, a lexicographic order $bsqr\ r$ on $(Field\ r) \times (Field\ r)$, applying the following criteria (in this order):

- compare the maximums;
- compare the first components;
- compare the second components.

The only application of this construction that we are aware of is at proving that the square of an infinite set has the same cardinal as that set. The essential property required there (and which is ensured by this construction) is that any proper order filter of the product order is included in a rectangle, i.e., in a product of proper filters on the original relation (assumed to be a well-order).

definition $bsqr :: 'a\ rel => ('a * 'a)rel$
where
$bsqr\ r = \{((a1,a2),(b1,b2)).$
$\{a1,a2,b1,b2\} \leq Field\ r \wedge$
$(a1 = b1 \wedge a2 = b2 \vee$
$(wo\text{-}rel.max2\ r\ a1\ a2,\ wo\text{-}rel.max2\ r\ b1\ b2) \in r - Id \vee$
$wo\text{-}rel.max2\ r\ a1\ a2 = wo\text{-}rel.max2\ r\ b1\ b2 \wedge (a1,b1) \in r - Id \vee$
$wo\text{-}rel.max2\ r\ a1\ a2 = wo\text{-}rel.max2\ r\ b1\ b2 \wedge a1 = b1\ \wedge (a2,b2) \in r$
$- Id$

)}

lemma *Field-bsqr*:
Field (*bsqr r*) = *Field r* × *Field r*
⟨*proof*⟩

lemma *bsqr-Refl*: *Refl*(*bsqr r*)
⟨*proof*⟩

lemma *bsqr-Trans*:
assumes *Well-order r*
shows *trans* (*bsqr r*)
⟨*proof*⟩

lemma *bsqr-antisym*:
assumes *Well-order r*
shows *antisym* (*bsqr r*)
⟨*proof*⟩

lemma *bsqr-Total*:
assumes *Well-order r*
shows *Total*(*bsqr r*)
⟨*proof*⟩

lemma *bsqr-Linear-order*:
assumes *Well-order r*
shows *Linear-order*(*bsqr r*)
⟨*proof*⟩

lemma *bsqr-Well-order*:
assumes *Well-order r*
shows *Well-order*(*bsqr r*)
⟨*proof*⟩

lemma *bsqr-max2*:
assumes *WELL*: *Well-order r* **and** *LEQ*: ((*a1*,*a2*),(*b1*,*b2*)) ∈ *bsqr r*
shows (*wo-rel.max2 r a1 a2*, *wo-rel.max2 r b1 b2*) ∈ *r*
⟨*proof*⟩

lemma *bsqr-ofilter*:
assumes *WELL*: *Well-order r* **and**
OF: *wo-rel.ofilter* (*bsqr r*) *D* **and** *SUB*: *D* < *Field r* × *Field r* **and**
NE: ¬ (∃ *a*. *Field r* = *under r a*)
shows ∃ *A*. *wo-rel.ofilter r A* ∧ *A* < *Field r* ∧ *D* ≤ *A* × *A*
⟨*proof*⟩

definition *Func* **where**
Func A B = {*f* . (∀ *a* ∈ *A*. *f a* ∈ *B*) ∧ (∀ *a*. *a* ∉ *A* ⟶ *f a* = *undefined*)}

lemma *Func-empty*:
$Func\ \{\}\ B = \{\lambda x.\ undefined\}$
⟨*proof*⟩

lemma *Func-elim*:
assumes $g \in Func\ A\ B$ **and** $a \in A$
shows $\exists\ b.\ b \in B \wedge g\ a = b$
⟨*proof*⟩

definition *curr* **where**
$curr\ A\ f \equiv \lambda\ a.\ if\ a \in A\ then\ \lambda b.\ f\ (a,b)\ else\ undefined$

lemma *curr-in*:
assumes f: $f \in Func\ (A \times B)\ C$
shows $curr\ A\ f \in Func\ A\ (Func\ B\ C)$
⟨*proof*⟩

lemma *curr-inj*:
assumes $f1 \in Func\ (A \times B)\ C$ **and** $f2 \in Func\ (A \times B)\ C$
shows $curr\ A\ f1 = curr\ A\ f2 \longleftrightarrow f1 = f2$
⟨*proof*⟩

lemma *curr-surj*:
assumes $g \in Func\ A\ (Func\ B\ C)$
shows $\exists\ f \in Func\ (A \times B)\ C.\ curr\ A\ f = g$
⟨*proof*⟩

lemma *bij-betw-curr*:
$bij\text{-}betw\ (curr\ A)\ (Func\ (A \times B)\ C)\ (Func\ A\ (Func\ B\ C))$
⟨*proof*⟩

definition *Func-map* **where**
$Func\text{-}map\ B2\ f1\ f2\ g\ b2 \equiv if\ b2 \in B2\ then\ f1\ (g\ (f2\ b2))\ else\ undefined$

lemma *Func-map*:
assumes g: $g \in Func\ A2\ A1$ **and** $f1$: $f1\ `\ A1 \subseteq B1$ **and** $f2$: $f2\ `\ B2 \subseteq A2$
shows $Func\text{-}map\ B2\ f1\ f2\ g \in Func\ B2\ B1$
⟨*proof*⟩

lemma *Func-non-emp*:
assumes $B \neq \{\}$
shows $Func\ A\ B \neq \{\}$
⟨*proof*⟩

lemma *Func-is-emp*:
$Func\ A\ B = \{\} \longleftrightarrow A \neq \{\} \wedge B = \{\}$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *Func-map-surj*:

assumes *B1*: *f1* ' *A1* = *B1* **and** *A2*: *inj-on f2 B2 f2* ' *B2* ⊆ *A2*
and *B2A2*: *B2* = {} ⟹ *A2* = {}
shows *Func B2 B1* = *Func-map B2 f1 f2* ' *Func A2 A1*
⟨*proof*⟩

end

29 Cardinal-Order Relations as Needed by Bounded Natural Functors

theory *BNF-Cardinal-Order-Relation*
imports *Zorn BNF-Wellorder-Constructions*
begin

In this section, we define cardinal-order relations to be minim well-orders on their field. Then we define the cardinal of a set to be *some* cardinal-order relation on that set, which will be unique up to order isomorphism. Then we study the connection between cardinals and:

- standard set-theoretic constructions: products, sums, unions, lists, powersets, set-of finite sets operator;
- finiteness and infiniteness (in particular, with the numeric cardinal operator for finite sets, *card*, from the theory *Finite-Sets.thy*).

On the way, we define the canonical ω cardinal and finite cardinals. We also define (again, up to order isomorphism) the successor of a cardinal, and show that any cardinal admits a successor.
Main results of this section are the existence of cardinal relations and the facts that, in the presence of infiniteness, most of the standard set-theoretic constructions (except for the powerset) *do not increase cardinality*. In particular, e.g., the set of words/lists over any infinite set has the same cardinality (hence, is in bijection) with that set.

29.1 Cardinal orders

A cardinal order in our setting shall be a well-order *minim* w.r.t. the order-embedding relation, $\leq o$ (which is the same as being *minimal* w.r.t. the strict order-embedding relation, $<o$), among all the well-orders on its field.

definition *card-order-on* :: *'a set* ⇒ *'a rel* ⇒ *bool*
where
card-order-on A r ≡ *well-order-on A r* ∧ (∀ *r'*. *well-order-on A r'* ⟶ *r* $\leq o$ *r'*)

abbreviation *Card-order r* ≡ *card-order-on* (*Field r*) *r*
abbreviation *card-order r* ≡ *card-order-on UNIV r*

lemma *card-order-on-well-order-on*:
assumes *card-order-on A r*
shows *well-order-on A r*
⟨*proof*⟩

lemma *card-order-on-Card-order*:
$card\text{-}order\text{-}on\ A\ r \Longrightarrow A = Field\ r \wedge Card\text{-}order\ r$
⟨*proof*⟩

The existence of a cardinal relation on any given set (which will mean that any set has a cardinal) follows from two facts:

- Zermelo's theorem (proved in *Zorn.thy* as theorem *well-order-on*), which states that on any given set there exists a well-order;
- The well-founded-ness of $<o$, ensuring that then there exists a minimal such well-order, i.e., a cardinal order.

theorem *card-order-on*: $\exists\, r.\ card\text{-}order\text{-}on\ A\ r$
⟨*proof*⟩

lemma *card-order-on-ordIso*:
assumes *CO*: *card-order-on A r* **and** CO': $card\text{-}order\text{-}on\ A\ r'$
shows $r =o\ r'$
⟨*proof*⟩

lemma *Card-order-ordIso*:
assumes *CO*: *Card-order r* **and** *ISO*: $r' =o\ r$
shows $Card\text{-}order\ r'$
⟨*proof*⟩

lemma *Card-order-ordIso2*:
assumes *CO*: *Card-order r* **and** *ISO*: $r =o\ r'$
shows $Card\text{-}order\ r'$
⟨*proof*⟩

29.2 Cardinal of a set

We define the cardinal of set to be *some* cardinal order on that set. We shall prove that this notion is unique up to order isomorphism, meaning that order isomorphism shall be the true identity of cardinals.

definition $card\text{-}of :: 'a\ set \Rightarrow 'a\ rel$ (|-|)
where $card\text{-}of\ A = (SOME\ r.\ card\text{-}order\text{-}on\ A\ r)$

lemma *card-of-card-order-on*: $card\text{-}order\text{-}on\ A\ |A|$
⟨*proof*⟩

lemma *card-of-well-order-on*: *well-order-on* A $|A|$
⟨*proof*⟩

lemma *Field-card-of*: *Field* $|A| = A$
⟨*proof*⟩

lemma *card-of-Card-order*: *Card-order* $|A|$
⟨*proof*⟩

corollary *ordIso-card-of-imp-Card-order*:
$r =o |A| \Longrightarrow$ *Card-order* r
⟨*proof*⟩

lemma *card-of-Well-order*: *Well-order* $|A|$
⟨*proof*⟩

lemma *card-of-refl*: $|A| =o |A|$
⟨*proof*⟩

lemma *card-of-least*: *well-order-on* A $r \Longrightarrow |A| \leq o\ r$
⟨*proof*⟩

lemma *card-of-ordIso*:
$(\exists f.$ *bij-betw* $f\ A\ B) = (\ |A| =o |B|\ )$
⟨*proof*⟩

lemma *card-of-ordLeq*:
$(\exists f.$ *inj-on* $f\ A \wedge f\ `\ A \leq B) = (\ |A| \leq o\ |B|\ )$
⟨*proof*⟩

lemma *card-of-ordLeq2*:
$A \neq \{\} \Longrightarrow (\exists g.\ g\ `\ B = A) = (\ |A| \leq o\ |B|\ )$
⟨*proof*⟩

lemma *card-of-ordLess*:
$(\neg(\exists f.$ *inj-on* $f\ A \wedge f\ `\ A \leq B)) = (\ |B| <o\ |A|\ )$
⟨*proof*⟩

lemma *card-of-ordLess2*:
$B \neq \{\} \Longrightarrow (\neg(\exists f.\ f\ `\ A = B)) = (\ |A| <o\ |B|\ )$
⟨*proof*⟩

lemma *card-of-ordIsoI*:
assumes *bij-betw* $f\ A\ B$
shows $|A| =o |B|$
⟨*proof*⟩

lemma *card-of-ordLeqI*:
assumes *inj-on* $f\ A$ **and** $\bigwedge\ a.\ a \in A \Longrightarrow f\ a \in B$

shows $|A| \leq o\ |B|$
⟨*proof*⟩

lemma *card-of-unique*:
card-order-on $A\ r \Longrightarrow r =o\ |A|$
⟨*proof*⟩

lemma *card-of-mono1*:
$A \leq B \Longrightarrow |A| \leq o\ |B|$
⟨*proof*⟩

lemma *card-of-mono2*:
assumes $r \leq o\ r'$
shows $|Field\ r| \leq o\ |Field\ r'|$
⟨*proof*⟩

lemma *card-of-cong*: $r =o\ r' \Longrightarrow |Field\ r| =o\ |Field\ r'|$
⟨*proof*⟩

lemma *card-of-Field-ordLess*: *Well-order* $r \Longrightarrow |Field\ r| \leq o\ r$
⟨*proof*⟩

lemma *card-of-Field-ordIso*:
assumes *Card-order* r
shows $|Field\ r| =o\ r$
⟨*proof*⟩

lemma *Card-order-iff-ordIso-card-of*:
Card-order $r = (r =o\ |Field\ r|\ )$
⟨*proof*⟩

lemma *Card-order-iff-ordLeq-card-of*:
Card-order $r = (r \leq o\ |Field\ r|\ )$
⟨*proof*⟩

lemma *Card-order-iff-Restr-underS*:
assumes *Well-order* r
shows *Card-order* $r = (\forall\ a \in Field\ r.\ Restr\ r\ (underS\ r\ a) <o\ |Field\ r|\ )$
⟨*proof*⟩

lemma *card-of-underS*:
assumes r: *Card-order* r **and** a: $a : Field\ r$
shows $|underS\ r\ a| <o\ r$
⟨*proof*⟩

lemma *ordLess-Field*:
assumes $r <o\ r'$
shows $|Field\ r| <o\ r'$
⟨*proof*⟩

lemma *internalize-card-of-ordLeq*:
$(|A| \leq o\ r) = (\exists B \leq \mathit{Field}\ r.\ |A| =o\ |B| \wedge |B| \leq o\ r)$
$\langle proof \rangle$

lemma *internalize-card-of-ordLeq2*:
$(|A| \leq o\ |C|\ ) = (\exists B \leq C.\ |A| =o\ |B| \wedge |B| \leq o\ |C|\ )$
$\langle proof \rangle$

29.3 Cardinals versus set operations on arbitrary sets

Here we embark in a long journey of simple results showing that the standard set-theoretic operations are well-behaved w.r.t. the notion of cardinal – essentially, this means that they preserve the "cardinal identity" $=o$ and are monotonic w.r.t. $\leq o$.

lemma *card-of-empty*: $|\{\}| \leq o\ |A|$
$\langle proof \rangle$

lemma *card-of-empty1*:
assumes $\mathit{Well\text{-}order}\ r \vee \mathit{Card\text{-}order}\ r$
shows $|\{\}| \leq o\ r$
$\langle proof \rangle$

corollary *Card-order-empty*:
$\mathit{Card\text{-}order}\ r \Longrightarrow |\{\}| \leq o\ r$ $\langle proof \rangle$

lemma *card-of-empty2*:
assumes LEQ: $|A| =o\ |\{\}|$
shows $A = \{\}$
$\langle proof \rangle$

lemma *card-of-empty3*:
assumes LEQ: $|A| \leq o\ |\{\}|$
shows $A = \{\}$
$\langle proof \rangle$

lemma *card-of-empty-ordIso*:
$|\{\}::'a\ set| =o\ |\{\}::'b\ set|$
$\langle proof \rangle$

lemma *card-of-image*:
$|f\ `\ A| <=o\ |A|$
$\langle proof \rangle$

lemma *surj-imp-ordLeq*:
assumes $B \subseteq f\ `\ A$
shows $|B| \leq o\ |A|$
$\langle proof \rangle$

lemma *card-of-singl-ordLeq*:
assumes $A \neq \{\}$
shows $|\{b\}| \leq o\ |A|$
⟨*proof*⟩

corollary *Card-order-singl-ordLeq*:
$[\![Card\text{-}order\ r;\ Field\ r \neq \{\}]\!] \Longrightarrow |\{b\}| \leq o\ r$
⟨*proof*⟩

lemma *card-of-Pow*: $|A| <o\ |Pow\ A|$
⟨*proof*⟩

corollary *Card-order-Pow*:
$Card\text{-}order\ r \Longrightarrow r <o\ |Pow(Field\ r)|$
⟨*proof*⟩

lemma *card-of-Plus1*: $|A| \leq o\ |A <+> B|$
⟨*proof*⟩

corollary *Card-order-Plus1*:
$Card\text{-}order\ r \Longrightarrow r \leq o\ |(Field\ r) <+> B|$
⟨*proof*⟩

lemma *card-of-Plus2*: $|B| \leq o\ |A <+> B|$
⟨*proof*⟩

corollary *Card-order-Plus2*:
$Card\text{-}order\ r \Longrightarrow r \leq o\ |A <+> (Field\ r)|$
⟨*proof*⟩

lemma *card-of-Plus-empty1*: $|A| =o\ |A <+> \{\}|$
⟨*proof*⟩

lemma *card-of-Plus-empty2*: $|A| =o\ |\{\} <+> A|$
⟨*proof*⟩

lemma *card-of-Plus-commute*: $|A <+> B| =o\ |B <+> A|$
⟨*proof*⟩

lemma *card-of-Plus-assoc*:
fixes $A :: 'a\ set$ **and** $B :: 'b\ set$ **and** $C :: 'c\ set$
shows $|(A <+> B) <+> C| =o\ |A <+> B <+> C|$
⟨*proof*⟩

lemma *card-of-Plus-mono1*:
assumes $|A| \leq o\ |B|$
shows $|A <+> C| \leq o\ |B <+> C|$
⟨*proof*⟩

corollary *ordLeq-Plus-mono1*:
assumes $r \leq o\ r'$
shows $|(Field\ r) <+> C| \leq o\ |(Field\ r') <+> C|$
⟨*proof*⟩

lemma *card-of-Plus-mono2*:
assumes $|A| \leq o\ |B|$
shows $|C <+> A| \leq o\ |C <+> B|$
⟨*proof*⟩

corollary *ordLeq-Plus-mono2*:
assumes $r \leq o\ r'$
shows $|A <+> (Field\ r)| \leq o\ |A <+> (Field\ r')|$
⟨*proof*⟩

lemma *card-of-Plus-mono*:
assumes $|A| \leq o\ |B|$ **and** $|C| \leq o\ |D|$
shows $|A <+> C| \leq o\ |B <+> D|$
⟨*proof*⟩

corollary *ordLeq-Plus-mono*:
assumes $r \leq o\ r'$ **and** $p \leq o\ p'$
shows $|(Field\ r) <+> (Field\ p)| \leq o\ |(Field\ r') <+> (Field\ p')|$
⟨*proof*⟩

lemma *card-of-Plus-cong1*:
assumes $|A| =o\ |B|$
shows $|A <+> C| =o\ |B <+> C|$
⟨*proof*⟩

corollary *ordIso-Plus-cong1*:
assumes $r =o\ r'$
shows $|(Field\ r) <+> C| =o\ |(Field\ r') <+> C|$
⟨*proof*⟩

lemma *card-of-Plus-cong2*:
assumes $|A| =o\ |B|$
shows $|C <+> A| =o\ |C <+> B|$
⟨*proof*⟩

corollary *ordIso-Plus-cong2*:
assumes $r =o\ r'$
shows $|A <+> (Field\ r)| =o\ |A <+> (Field\ r')|$
⟨*proof*⟩

lemma *card-of-Plus-cong*:
assumes $|A| =o\ |B|$ **and** $|C| =o\ |D|$
shows $|A <+> C| =o\ |B <+> D|$

⟨*proof*⟩

corollary *ordIso-Plus-cong*:
assumes $r =o\ r'$ **and** $p =o\ p'$
shows $|(\mathit{Field}\ r) <+> (\mathit{Field}\ p)| =o\ |(\mathit{Field}\ r') <+> (\mathit{Field}\ p')|$
⟨*proof*⟩

lemma *card-of-Un-Plus-ordLeq*:
$|A \cup B| \le o\ |A <+> B|$
⟨*proof*⟩

lemma *card-of-Times1*:
assumes $A \neq \{\}$
shows $|B| \le o\ |B \times A|$
⟨*proof*⟩

lemma *card-of-Times-commute*: $|A \times B| =o\ |B \times A|$
⟨*proof*⟩

lemma *card-of-Times2*:
assumes $A \neq \{\}$ **shows** $|B| \le o\ |A \times B|$
⟨*proof*⟩

corollary *Card-order-Times1*:
$[\![\mathit{Card\text{-}order}\ r;\ B \neq \{\}]\!] \Longrightarrow r \le o\ |(\mathit{Field}\ r) \times B|$
⟨*proof*⟩

corollary *Card-order-Times2*:
$[\![\mathit{Card\text{-}order}\ r;\ A \neq \{\}]\!] \Longrightarrow r \le o\ |A \times (\mathit{Field}\ r)|$
⟨*proof*⟩

lemma *card-of-Times3*: $|A| \le o\ |A \times A|$
⟨*proof*⟩

lemma *card-of-Plus-Times-bool*: $|A <+> A| =o\ |A \times (\mathit{UNIV}::\mathit{bool\ set})|$
⟨*proof*⟩

lemma *card-of-Times-mono1*:
assumes $|A| \le o\ |B|$
shows $|A \times C| \le o\ |B \times C|$
⟨*proof*⟩

corollary *ordLeq-Times-mono1*:
assumes $r \le o\ r'$
shows $|(\mathit{Field}\ r) \times C| \le o\ |(\mathit{Field}\ r') \times C|$
⟨*proof*⟩

lemma *card-of-Times-mono2*:
assumes $|A| \le o\ |B|$

shows $|C \times A| \leq o\ |C \times B|$
⟨*proof*⟩

corollary *ordLeq-Times-mono2*:
assumes $r \leq o\ r'$
shows $|A \times (Field\ r)| \leq o\ |A \times (Field\ r')|$
⟨*proof*⟩

lemma *card-of-Sigma-mono1*:
assumes $\forall\, i \in I.\ |A\ i| \leq o\ |B\ i|$
shows $|SIGMA\ i : I.\ A\ i| \leq o\ |SIGMA\ i : I.\ B\ i|$
⟨*proof*⟩

lemma *card-of-UNION-Sigma*:
$|\bigcup i \in I.\ A\ i| \leq o\ |SIGMA\ i : I.\ A\ i|$
⟨*proof*⟩

lemma *card-of-bool*:
assumes $a1 \neq a2$
shows $|UNIV::bool\ set| =o\ |\{a1,a2\}|$
⟨*proof*⟩

lemma *card-of-Plus-Times-aux*:
assumes $A2$: $a1 \neq a2 \wedge \{a1,a2\} \leq A$ **and**
  LEQ: $|A| \leq o\ |B|$
shows $|A <+> B| \leq o\ |A \times B|$
⟨*proof*⟩

lemma *card-of-Plus-Times*:
assumes $A2$: $a1 \neq a2 \wedge \{a1,a2\} \leq A$ **and**
  $B2$: $b1 \neq b2 \wedge \{b1,b2\} \leq B$
shows $|A <+> B| \leq o\ |A \times B|$
⟨*proof*⟩

lemma *card-of-Times-Plus-distrib*:
 $|A \times (B <+> C)| =o\ |A \times B <+> A \times C|$ (**is** $|?RHS| =o\ |?LHS|$)
⟨*proof*⟩

lemma *card-of-ordLeq-finite*:
assumes $|A| \leq o\ |B|$ **and** *finite B*
shows *finite A*
⟨*proof*⟩

lemma *card-of-ordLeq-infinite*:
assumes $|A| \leq o\ |B|$ **and** $\neg$ *finite A*
shows $\neg$ *finite B*
⟨*proof*⟩

lemma *card-of-ordIso-finite*:

assumes $|A| =o |B|$
shows *finite* A = *finite* B
⟨*proof*⟩

lemma *card-of-ordIso-finite-Field*:
assumes *Card-order* r **and** $r =o |A|$
shows *finite*(*Field* r) = *finite* A
⟨*proof*⟩

29.4 Cardinals versus set operations involving infinite sets

Here we show that, for infinite sets, most set-theoretic constructions do not increase the cardinality. The cornerstone for this is theorem *Card-order-Times-same-infinite*, which states that self-product does not increase cardinality – the proof of this fact adapts a standard set-theoretic argument, as presented, e.g., in the proof of theorem 1.5.11 at page 47 in [2]. Then everything else follows fairly easily.

lemma *infinite-iff-card-of-nat*:
$\neg$ *finite* $A \longleftrightarrow$ ($|UNIV::nat\ set| \le o |A|$)
⟨*proof*⟩

The next two results correspond to the ZF fact that all infinite cardinals are limit ordinals:

lemma *Card-order-infinite-not-under*:
assumes *CARD*: *Card-order* r **and** *INF*: $\neg$*finite* (*Field* r)
shows $\neg$ ($\exists\, a$. *Field* r = *under* r a)
⟨*proof*⟩

lemma *infinite-Card-order-limit*:
assumes r: *Card-order* r **and** $\neg$*finite* (*Field* r)
and a: a : *Field* r
shows *EX* b : *Field* r. $a \neq b \wedge (a,b) : r$
⟨*proof*⟩

theorem *Card-order-Times-same-infinite*:
assumes *CO*: *Card-order* r **and** *INF*: $\neg$*finite*(*Field* r)
shows $|Field\ r \times Field\ r| \le o\ r$
⟨*proof*⟩

corollary *card-of-Times-same-infinite*:
assumes $\neg$*finite* A
shows $|A \times A| =o |A|$
⟨*proof*⟩

lemma *card-of-Times-infinite*:
assumes *INF*: $\neg$*finite* A **and** *NE*: $B \neq \{\}$ **and** *LEQ*: $|B| \le o |A|$
shows $|A \times B| =o |A| \wedge |B \times A| =o |A|$
⟨*proof*⟩

corollary *Card-order-Times-infinite*:
assumes *INF*: $\neg$*finite*(*Field r*) **and** *CARD*: *Card-order r* **and**
NE: *Field p* $\neq$ {} **and** *LEQ*: $p \leq o\ r$
shows $|$ (*Field r*) $\times$ (*Field p*) $| =o\ r \wedge |$ (*Field p*) $\times$ (*Field r*) $| =o\ r$
⟨*proof*⟩

lemma *card-of-Sigma-ordLeq-infinite*:
assumes *INF*: $\neg$*finite B* **and**
LEQ-I: $|I| \leq o\ |B|$ **and** *LEQ*: $\forall\, i \in I.\ |A\ i| \leq o\ |B|$
shows $|SIGMA\ i : I.\ A\ i| \leq o\ |B|$
⟨*proof*⟩

lemma *card-of-Sigma-ordLeq-infinite-Field*:
assumes *INF*: $\neg$*finite* (*Field r*) **and** *r*: *Card-order r* **and**
LEQ-I: $|I| \leq o\ r$ **and** *LEQ*: $\forall\, i \in I.\ |A\ i| \leq o\ r$
shows $|SIGMA\ i : I.\ A\ i| \leq o\ r$
⟨*proof*⟩

lemma *card-of-Times-ordLeq-infinite-Field*:
⟦$\neg$*finite* (*Field r*); $|A| \leq o\ r$; $|B| \leq o\ r$; *Card-order r*⟧
$\Longrightarrow |A \times B| \leq o\ r$
⟨*proof*⟩

lemma *card-of-Times-infinite-simps*:
⟦$\neg$*finite A*; $B \neq$ {}; $|B| \leq o\ |A|$⟧ $\Longrightarrow |A \times B| =o\ |A|$
⟦$\neg$*finite A*; $B \neq$ {}; $|B| \leq o\ |A|$⟧ $\Longrightarrow |A| =o\ |A \times B|$
⟦$\neg$*finite A*; $B \neq$ {}; $|B| \leq o\ |A|$⟧ $\Longrightarrow |B \times A| =o\ |A|$
⟦$\neg$*finite A*; $B \neq$ {}; $|B| \leq o\ |A|$⟧ $\Longrightarrow |A| =o\ |B \times A|$
⟨*proof*⟩

lemma *card-of-UNION-ordLeq-infinite*:
assumes *INF*: $\neg$*finite B* **and**
LEQ-I: $|I| \leq o\ |B|$ **and** *LEQ*: $\forall\, i \in I.\ |A\ i| \leq o\ |B|$
shows $|\bigcup i \in I.\ A\ i| \leq o\ |B|$
⟨*proof*⟩

corollary *card-of-UNION-ordLeq-infinite-Field*:
assumes *INF*: $\neg$*finite* (*Field r*) **and** *r*: *Card-order r* **and**
LEQ-I: $|I| \leq o\ r$ **and** *LEQ*: $\forall\, i \in I.\ |A\ i| \leq o\ r$
shows $|\bigcup i \in I.\ A\ i| \leq o\ r$
⟨*proof*⟩

lemma *card-of-Plus-infinite1*:
assumes *INF*: $\neg$*finite A* **and** *LEQ*: $|B| \leq o\ |A|$
shows $|A <+> B| =o\ |A|$
⟨*proof*⟩

lemma *card-of-Plus-infinite2*:

assumes *INF*: $\neg$*finite* A **and** *LEQ*: $|B| \leq o\ |A|$
shows $|B <+> A| =o\ |A|$
⟨*proof*⟩

lemma *card-of-Plus-infinite*:
assumes *INF*: $\neg$*finite* A **and** *LEQ*: $|B| \leq o\ |A|$
shows $|A <+> B| =o\ |A| \wedge |B <+> A| =o\ |A|$
⟨*proof*⟩

corollary *Card-order-Plus-infinite*:
assumes *INF*: $\neg$*finite*(*Field* r) **and** *CARD*: *Card-order* r **and**
LEQ: $p \leq o\ r$
shows $|$ (*Field* r) $<+>$ (*Field* p) $| =o\ r \wedge |$ (*Field* p) $<+>$ (*Field* r) $| =o\ r$
⟨*proof*⟩

29.5 The cardinal ω and the finite cardinals

The cardinal ω, of natural numbers, shall be the standard non-strict order relation on *nat*, that we abbreviate by *natLeq*. The finite cardinals shall be the restrictions of these relations to the numbers smaller than fixed numbers n, that we abbreviate by *natLeq-on* n.

definition (*natLeq*::(*nat* $*$ *nat*) *set*) $\equiv \{(x,y).\ x \leq y\}$
definition (*natLess*::(*nat* $*$ *nat*) *set*) $\equiv \{(x,y).\ x < y\}$

abbreviation *natLeq-on* :: *nat* $\Rightarrow$ (*nat* $*$ *nat*) *set*
where *natLeq-on* $n \equiv \{(x,y).\ x < n \wedge y < n \wedge x \leq y\}$

lemma *infinite-cartesian-product*:
assumes $\neg$*finite* A $\neg$*finite* B
shows $\neg$*finite* $(A \times B)$
⟨*proof*⟩

29.5.1 First as well-orders

lemma *Field-natLeq*: *Field natLeq* = (*UNIV*::*nat set*)
⟨*proof*⟩

lemma *natLeq-Refl*: *Refl natLeq*
⟨*proof*⟩

lemma *natLeq-trans*: *trans natLeq*
⟨*proof*⟩

lemma *natLeq-Preorder*: *Preorder natLeq*
⟨*proof*⟩

lemma *natLeq-antisym*: *antisym natLeq*
⟨*proof*⟩

lemma *natLeq-Partial-order*: *Partial-order natLeq*
⟨*proof*⟩

lemma *natLeq-Total*: *Total natLeq*
⟨*proof*⟩

lemma *natLeq-Linear-order*: *Linear-order natLeq*
⟨*proof*⟩

lemma *natLeq-natLess-Id*: *natLess* = *natLeq* − *Id*
⟨*proof*⟩

lemma *natLeq-Well-order*: *Well-order natLeq*
⟨*proof*⟩

lemma *Field-natLeq-on*: *Field* (*natLeq-on* n) = $\{x.\ x < n\}$
⟨*proof*⟩

lemma *natLeq-underS-less*: *underS natLeq* $n = \{x.\ x < n\}$
⟨*proof*⟩

lemma *Restr-natLeq*: *Restr natLeq* $\{x.\ x < n\}$ = *natLeq-on* n
⟨*proof*⟩

lemma *Restr-natLeq2*:
Restr natLeq (*underS natLeq* n) = *natLeq-on* n
⟨*proof*⟩

lemma *natLeq-on-Well-order*: *Well-order*(*natLeq-on* n)
⟨*proof*⟩

corollary *natLeq-on-well-order-on*: *well-order-on* $\{x.\ x < n\}$ (*natLeq-on* n)
⟨*proof*⟩

lemma *natLeq-on-wo-rel*: *wo-rel*(*natLeq-on* n)
⟨*proof*⟩

29.5.2 Then as cardinals

lemma *natLeq-Card-order*: *Card-order natLeq*
⟨*proof*⟩

corollary *card-of-Field-natLeq*:
|*Field natLeq*| =*o natLeq*
⟨*proof*⟩

corollary *card-of-nat*:
|*UNIV*::*nat set*| =*o natLeq*
⟨*proof*⟩

corollary *infinite-iff-natLeq-ordLeq*:
$\neg finite\ A = (\ natLeq \leq o\ |A|\ )$
$\langle proof \rangle$

corollary *finite-iff-ordLess-natLeq*:
$finite\ A = (\ |A| <o\ natLeq)$
$\langle proof \rangle$

29.6 The successor of a cardinal

First we define *isCardSuc* $r\ r'$, the notion of r' being a successor cardinal of r. Although the definition does not require r to be a cardinal, only this case will be meaningful.

definition $isCardSuc :: 'a\ rel \Rightarrow 'a\ set\ rel \Rightarrow bool$
where
$isCardSuc\ r\ r' \equiv$
$Card\text{-}order\ r' \wedge r <o\ r' \wedge$
$(\forall (r''::'a\ set\ rel).\ Card\text{-}order\ r'' \wedge r <o\ r'' \longrightarrow r' \leq o\ r'')$

Now we introduce the cardinal-successor operator *cardSuc*, by picking *some* cardinal-order relation fulfilling *isCardSuc*. Again, the picked item shall be proved unique up to order-isomorphism.

definition $cardSuc :: 'a\ rel \Rightarrow 'a\ set\ rel$
where
$cardSuc\ r \equiv SOME\ r'.\ isCardSuc\ r\ r'$

lemma *exists-minim-Card-order*:
$[\![R \neq \{\};\ \forall r \in R.\ Card\text{-}order\ r]\!] \Longrightarrow \exists r \in R.\ \forall r' \in R.\ r \leq o\ r'$
$\langle proof \rangle$

lemma *exists-isCardSuc*:
assumes $Card\text{-}order\ r$
shows $\exists r'.\ isCardSuc\ r\ r'$
$\langle proof \rangle$

lemma *cardSuc-isCardSuc*:
assumes $Card\text{-}order\ r$
shows $isCardSuc\ r\ (cardSuc\ r)$
$\langle proof \rangle$

lemma *cardSuc-Card-order*:
$Card\text{-}order\ r \Longrightarrow Card\text{-}order(cardSuc\ r)$
$\langle proof \rangle$

lemma *cardSuc-greater*:
$Card\text{-}order\ r \Longrightarrow r <o\ cardSuc\ r$
$\langle proof \rangle$

lemma *cardSuc-ordLeq*:
$Card\text{-}order\ r \Longrightarrow r \le o\ cardSuc\ r$
⟨*proof*⟩

The minimality property of *cardSuc* originally present in its definition is local to the type $'a\ set\ rel$, i.e., that of $cardSuc\ r$:

lemma *cardSuc-least-aux*:
$[\![Card\text{-}order\ (r{::}'a\ rel);\ Card\text{-}order\ (r'{::}'a\ set\ rel);\ r <o\ r']\!] \Longrightarrow cardSuc\ r \le o\ r'$
⟨*proof*⟩

But from this we can infer general minimality:

lemma *cardSuc-least*:
assumes $CARD$: $Card\text{-}order\ r$ **and** $CARD'$: $Card\text{-}order\ r'$ **and** $LESS$: $r <o\ r'$
shows $cardSuc\ r \le o\ r'$
⟨*proof*⟩

lemma *cardSuc-ordLess-ordLeq*:
assumes $CARD$: $Card\text{-}order\ r$ **and** $CARD'$: $Card\text{-}order\ r'$
shows $(r <o\ r') = (cardSuc\ r \le o\ r')$
⟨*proof*⟩

lemma *cardSuc-ordLeq-ordLess*:
assumes $CARD$: $Card\text{-}order\ r$ **and** $CARD'$: $Card\text{-}order\ r'$
shows $(r' <o\ cardSuc\ r) = (r' \le o\ r)$
⟨*proof*⟩

lemma *cardSuc-mono-ordLeq*:
assumes $CARD$: $Card\text{-}order\ r$ **and** $CARD'$: $Card\text{-}order\ r'$
shows $(cardSuc\ r \le o\ cardSuc\ r') = (r \le o\ r')$
⟨*proof*⟩

lemma *cardSuc-invar-ordIso*:
assumes $CARD$: $Card\text{-}order\ r$ **and** $CARD'$: $Card\text{-}order\ r'$
shows $(cardSuc\ r =o\ cardSuc\ r') = (r =o\ r')$
⟨*proof*⟩

lemma *card-of-cardSuc-finite*:
$finite(Field(cardSuc\ |A|\ )) = finite\ A$
⟨*proof*⟩

lemma *cardSuc-finite*:
assumes $Card\text{-}order\ r$
shows $finite\ (Field\ (cardSuc\ r)) = finite\ (Field\ r)$
⟨*proof*⟩

lemma *card-of-Plus-ordLess-infinite*:
assumes INF: $\neg finite\ C$ **and**
$LESS1$: $|A| <o\ |C|$ **and** $LESS2$: $|B| <o\ |C|$

shows $|A <+> B| <o\ |C|$
⟨*proof*⟩

lemma *card-of-Plus-ordLess-infinite-Field*:
assumes *INF*: $\neg$*finite* (*Field r*) **and** *r*: *Card-order r* **and**
LESS1: $|A| <o\ r$ **and** *LESS2*: $|B| <o\ r$
shows $|A <+> B| <o\ r$
⟨*proof*⟩

lemma *card-of-Plus-ordLeq-infinite-Field*:
assumes *r*: $\neg$*finite* (*Field r*) **and** *A*: $|A| \leq o\ r$ **and** *B*: $|B| \leq o\ r$
and *c*: *Card-order r*
shows $|A <+> B| \leq o\ r$
⟨*proof*⟩

lemma *card-of-Un-ordLeq-infinite-Field*:
assumes *C*: $\neg$*finite* (*Field r*) **and** *A*: $|A| \leq o\ r$ **and** *B*: $|B| \leq o\ r$
and *Card-order r*
shows $|A\ Un\ B| \leq o\ r$
⟨*proof*⟩

29.7 Regular cardinals

definition *cofinal* **where**
cofinal A r $\equiv$
ALL a : Field r. EX b : A. $a \neq b \wedge (a,b) : r$

definition *regularCard* **where**
regularCard r $\equiv$
ALL K. K $\leq$ *Field r* $\wedge$ *cofinal K r* $\longrightarrow |K| =o\ r$

definition *relChain* **where**
relChain r As $\equiv$
ALL i j. $(i,j) \in r \longrightarrow$ *As i* $\leq$ *As j*

lemma *regularCard-UNION*:
assumes *r*: *Card-order r* *regularCard r*
and *As*: *relChain r As*
and *Bsub*: *B* $\leq$ (*UN i : Field r. As i*)
and *cardB*: $|B| <o\ r$
shows *EX i : Field r. B* $\leq$ *As i*
⟨*proof*⟩

lemma *infinite-cardSuc-regularCard*:
assumes *r-inf*: $\neg$*finite* (*Field r*) **and** *r-card*: *Card-order r*
shows *regularCard* (*cardSuc r*)
⟨*proof*⟩

lemma *cardSuc-UNION*:

assumes *r*: *Card-order r* **and** $\neg$*finite* (*Field r*)
and *As*: *relChain* (*cardSuc r*) *As*
and *Bsub*: $B \leq$ (*UN i* : *Field* (*cardSuc r*). *As i*)
and *cardB*: $|B|$ <=o *r*
shows *EX i* : *Field* (*cardSuc r*). $B \leq$ *As i*
⟨*proof*⟩

29.8 Others

lemma *card-of-Func-Times*:
$|Func\ (A \times B)\ C|$ =o $|Func\ A\ (Func\ B\ C)|$
⟨*proof*⟩

lemma *card-of-Pow-Func*:
$|Pow\ A|$ =o $|Func\ A\ (UNIV{::}bool\ set)|$
⟨*proof*⟩

lemma *card-of-Func-UNIV*:
$|Func\ (UNIV{::}'a\ set)\ (B{::}'b\ set)|$ =o $|\{f{::}'a \Rightarrow 'b.\ range\ f \subseteq B\}|$
⟨*proof*⟩

lemma *Func-Times-Range*:
$|Func\ A\ (B \times C)|$ =o $|Func\ A\ B \times Func\ A\ C|$ (**is** $|?LHS|$ =o $|?RHS|$)
⟨*proof*⟩

end

30 Cardinal Arithmetic as Needed by Bounded Natural Functors

theory *BNF-Cardinal-Arithmetic*
imports *BNF-Cardinal-Order-Relation*
begin

lemma *dir-image*: $[\![\bigwedge x\ y.\ (f\ x = f\ y) = (x = y);\ Card\text{-}order\ r]\!] \Longrightarrow r$ =o *dir-image r f*
⟨*proof*⟩

lemma *card-order-dir-image*:
assumes *bij*: *bij f* **and** *co*: *card-order r*
shows *card-order* (*dir-image r f*)
⟨*proof*⟩

lemma *ordIso-refl*: *Card-order r* $\Longrightarrow$ *r* =o *r*
⟨*proof*⟩

lemma *ordLeq-refl*: *Card-order r* $\Longrightarrow$ *r* $\leq$o *r*
⟨*proof*⟩

lemma *card-of-ordIso-subst*: $A = B \Longrightarrow |A| =o\ |B|$
⟨*proof*⟩

lemma *Field-card-order*: $card\text{-}order\ r \Longrightarrow Field\ r = UNIV$
⟨*proof*⟩

30.1 Zero

definition *czero* **where**
$czero = card\text{-}of\ \{\}$

lemma *czero-ordIso*:
$czero =o\ czero$
⟨*proof*⟩

lemma *card-of-ordIso-czero-iff-empty*:
$|A| =o\ (czero :: 'b\ rel) \longleftrightarrow A = (\{\} :: 'a\ set)$
⟨*proof*⟩

abbreviation *Cnotzero* **where**
$Cnotzero\ (r :: 'a\ rel) \equiv \neg(r =o\ (czero :: 'a\ rel)) \wedge Card\text{-}order\ r$

lemma *Cnotzero-imp-not-empty*: $Cnotzero\ r \Longrightarrow Field\ r \neq \{\}$
⟨*proof*⟩

lemma *czeroI*:
$[\![Card\text{-}order\ r;\ Field\ r = \{\}]\!] \Longrightarrow r =o\ czero$
⟨*proof*⟩

lemma *czeroE*:
$r =o\ czero \Longrightarrow Field\ r = \{\}$
⟨*proof*⟩

lemma *Cnotzero-mono*:
$[\![Cnotzero\ r;\ Card\text{-}order\ q;\ r \leq o\ q]\!] \Longrightarrow Cnotzero\ q$
⟨*proof*⟩

30.2 (In)finite cardinals

definition *cinfinite* **where**
$cinfinite\ r = (\neg\ finite\ (Field\ r))$

abbreviation *Cinfinite* **where**
$Cinfinite\ r \equiv cinfinite\ r \wedge Card\text{-}order\ r$

definition *cfinite* **where**
$cfinite\ r = finite\ (Field\ r)$

abbreviation *Cfinite* **where**
Cfinite r $\equiv$ *cfinite r* $\wedge$ *Card-order r*

lemma *Cfinite-ordLess-Cinfinite*: $[\![$*Cfinite r*; *Cinfinite s*$]\!] \Longrightarrow r <o\ s$
$\langle proof \rangle$

lemmas *natLeq-card-order* = *natLeq-Card-order*[*unfolded Field-natLeq*]

lemma *natLeq-cinfinite*: *cinfinite natLeq*
$\langle proof \rangle$

lemma *natLeq-ordLeq-cinfinite*:
assumes *inf*: *Cinfinite r*
shows *natLeq* $\leq o$ *r*
$\langle proof \rangle$

lemma *cinfinite-not-czero*: *cinfinite r* $\Longrightarrow \neg$ (*r* $=o$ (*czero* :: *'a rel*))
$\langle proof \rangle$

lemma *Cinfinite-Cnotzero*: *Cinfinite r* $\Longrightarrow$ *Cnotzero r*
$\langle proof \rangle$

lemma *Cinfinite-cong*: $[\![$*r1* $=o$ *r2*; *Cinfinite r1*$]\!] \Longrightarrow$ *Cinfinite r2*
$\langle proof \rangle$

lemma *cinfinite-mono*: $[\![$*r1* $\leq o$ *r2*; *cinfinite r1*$]\!] \Longrightarrow$ *cinfinite r2*
$\langle proof \rangle$

30.3 Binary sum

definition *csum* (**infixr** $+c$ *65*) **where**
r1 $+c$ *r2* $\equiv$ |*Field r1* <+> *Field r2*|

lemma *Field-csum*: *Field* (*r* $+c$ *s*) = *Inl* ' *Field r* $\cup$ *Inr* ' *Field s*
$\langle proof \rangle$

lemma *Card-order-csum*:
Card-order (*r1* $+c$ *r2*)
$\langle proof \rangle$

lemma *csum-Cnotzero1*:
Cnotzero r1 $\Longrightarrow$ *Cnotzero* (*r1* $+c$ *r2*)
$\langle proof \rangle$

lemma *card-order-csum*:
assumes *card-order r1 card-order r2*
shows *card-order* (*r1* $+c$ *r2*)
$\langle proof \rangle$

lemma *cinfinite-csum*:
cinfinite r1 ∨ *cinfinite r2* ⟹ *cinfinite* (*r1* +*c* *r2*)
⟨*proof*⟩

lemma *Cinfinite-csum1*:
Cinfinite r1 ⟹ *Cinfinite* (*r1* +*c* *r2*)
⟨*proof*⟩

lemma *Cinfinite-csum*:
Cinfinite r1 ∨ *Cinfinite r2* ⟹ *Cinfinite* (*r1* +*c* *r2*)
⟨*proof*⟩

lemma *Cinfinite-csum-weak*:
⟦*Cinfinite r1*; *Cinfinite r2*⟧ ⟹ *Cinfinite* (*r1* +*c* *r2*)
⟨*proof*⟩

lemma *csum-cong*: ⟦*p1* =*o* *r1*; *p2* =*o* *r2*⟧ ⟹ *p1* +*c* *p2* =*o* *r1* +*c* *r2*
⟨*proof*⟩

lemma *csum-cong1*: *p1* =*o* *r1* ⟹ *p1* +*c* *q* =*o* *r1* +*c* *q*
⟨*proof*⟩

lemma *csum-cong2*: *p2* =*o* *r2* ⟹ *q* +*c* *p2* =*o* *q* +*c* *r2*
⟨*proof*⟩

lemma *csum-mono*: ⟦*p1* ≤*o* *r1*; *p2* ≤*o* *r2*⟧ ⟹ *p1* +*c* *p2* ≤*o* *r1* +*c* *r2*
⟨*proof*⟩

lemma *csum-mono1*: *p1* ≤*o* *r1* ⟹ *p1* +*c* *q* ≤*o* *r1* +*c* *q*
⟨*proof*⟩

lemma *csum-mono2*: *p2* ≤*o* *r2* ⟹ *q* +*c* *p2* ≤*o* *q* +*c* *r2*
⟨*proof*⟩

lemma *ordLeq-csum1*: *Card-order p1* ⟹ *p1* ≤*o* *p1* +*c* *p2*
⟨*proof*⟩

lemma *ordLeq-csum2*: *Card-order p2* ⟹ *p2* ≤*o* *p1* +*c* *p2*
⟨*proof*⟩

lemma *csum-com*: *p1* +*c* *p2* =*o* *p2* +*c* *p1*
⟨*proof*⟩

lemma *csum-assoc*: (*p1* +*c* *p2*) +*c* *p3* =*o* *p1* +*c* *p2* +*c* *p3*
⟨*proof*⟩

lemma *Cfinite-csum*: ⟦*Cfinite r*; *Cfinite s*⟧ ⟹ *Cfinite* (*r* +*c* *s*)
⟨*proof*⟩

lemma *csum-csum*: $(r1 +c\ r2) +c\ (r3 +c\ r4) =o\ (r1 +c\ r3) +c\ (r2 +c\ r4)$
⟨*proof*⟩

lemma *Plus-csum*: $|A <+> B| =o\ |A| +c\ |B|$
⟨*proof*⟩

lemma *Un-csum*: $|A \cup B| \leq o\ |A| +c\ |B|$
⟨*proof*⟩

30.4 One

definition *cone* **where**
cone = *card-of* {()}

lemma *Card-order-cone*: *Card-order cone*
⟨*proof*⟩

lemma *Cfinite-cone*: *Cfinite cone*
⟨*proof*⟩

lemma *cone-not-czero*: ¬ (*cone* =*o czero*)
⟨*proof*⟩

lemma *cone-ordLeq-Cnotzero*: *Cnotzero r* ⟹ *cone* ≤*o r*
⟨*proof*⟩

30.5 Two

definition *ctwo* **where**
ctwo = |*UNIV* :: *bool set*|

lemma *Card-order-ctwo*: *Card-order ctwo*
⟨*proof*⟩

lemma *ctwo-not-czero*: ¬ (*ctwo* =*o czero*)
⟨*proof*⟩

lemma *ctwo-Cnotzero*: *Cnotzero ctwo*
⟨*proof*⟩

30.6 Family sum

definition *Csum* **where**
Csum r rs ≡ |*SIGMA i* : *Field r*. *Field* (*rs i*)|

syntax *-Csum* ::
pttrn => ('*a* ∗ '*a*) *set* => '*b* ∗ '*b set* => (('*a* ∗ '*b*) ∗ ('*a* ∗ '*b*)) *set*
((*3CSUM* -:-. -) [*0*, *51*, *10*] *10*)

translations
CSUM i:r. rs == CONST Csum r (%i. rs)

lemma *SIGMA-CSUM*: $|SIGMA\ i : I.\ As\ i| = (CSUM\ i : |I|.\ |As\ i|\ )$
⟨*proof*⟩

30.7 Product

definition *cprod* (**infixr** $*c$ *80*) **where**
$r1\ *c\ r2 = |Field\ r1 \times Field\ r2|$

lemma *card-order-cprod*:
assumes *card-order r1 card-order r2*
shows *card-order* $(r1\ *c\ r2)$
⟨*proof*⟩

lemma *Card-order-cprod*: *Card-order* $(r1\ *c\ r2)$
⟨*proof*⟩

lemma *cprod-mono1*: $p1 \leq o\ r1 \Longrightarrow p1\ *c\ q \leq o\ r1\ *c\ q$
⟨*proof*⟩

lemma *cprod-mono2*: $p2 \leq o\ r2 \Longrightarrow q\ *c\ p2 \leq o\ q\ *c\ r2$
⟨*proof*⟩

lemma *cprod-mono*: $[\![p1 \leq o\ r1;\ p2 \leq o\ r2]\!] \Longrightarrow p1\ *c\ p2 \leq o\ r1\ *c\ r2$
⟨*proof*⟩

lemma *ordLeq-cprod2*: $[\![Cnotzero\ p1;\ Card\text{-}order\ p2]\!] \Longrightarrow p2 \leq o\ p1\ *c\ p2$
⟨*proof*⟩

lemma *cinfinite-cprod*: $[\![cinfinite\ r1;\ cinfinite\ r2]\!] \Longrightarrow cinfinite\ (r1\ *c\ r2)$
⟨*proof*⟩

lemma *cinfinite-cprod2*: $[\![Cnotzero\ r1;\ Cinfinite\ r2]\!] \Longrightarrow cinfinite\ (r1\ *c\ r2)$
⟨*proof*⟩

lemma *Cinfinite-cprod2*: $[\![Cnotzero\ r1;\ Cinfinite\ r2]\!] \Longrightarrow Cinfinite\ (r1\ *c\ r2)$
⟨*proof*⟩

lemma *cprod-cong*: $[\![p1 =o\ r1;\ p2 =o\ r2]\!] \Longrightarrow p1\ *c\ p2 =o\ r1\ *c\ r2$
⟨*proof*⟩

lemma *cprod-cong1*: $[\![p1 =o\ r1]\!] \Longrightarrow p1\ *c\ p2 =o\ r1\ *c\ p2$
⟨*proof*⟩

lemma *cprod-cong2*: $p2 =o\ r2 \Longrightarrow q\ *c\ p2 =o\ q\ *c\ r2$
⟨*proof*⟩

lemma *cprod-com*: $p1 *c\ p2 =o\ p2 *c\ p1$
⟨*proof*⟩

lemma *card-of-Csum-Times*:
$\forall i \in I.\ |A\ i| \le o\ |B| \Longrightarrow (CSUM\ i : |I|.\ |A\ i|\ ) \le o\ |I| *c\ |B|$
⟨*proof*⟩

lemma *card-of-Csum-Times'*:
assumes $Card\text{-}order\ r\ \forall i \in I.\ |A\ i| \le o\ r$
shows $(CSUM\ i : |I|.\ |A\ i|\ ) \le o\ |I| *c\ r$
⟨*proof*⟩

lemma *cprod-csum-distrib1*: $r1 *c\ r2 +c\ r1 *c\ r3 =o\ r1 *c\ (r2 +c\ r3)$
⟨*proof*⟩

lemma *csum-absorb2'*: $[\![Card\text{-}order\ r2;\ r1 \le o\ r2;\ cinfinite\ r1 \lor cinfinite\ r2]\!] \Longrightarrow r1 +c\ r2 =o\ r2$
⟨*proof*⟩

lemma *csum-absorb1'*:
assumes *card*: $Card\text{-}order\ r2$
and *r12*: $r1 \le o\ r2$ **and** *cr12*: $cinfinite\ r1 \lor cinfinite\ r2$
shows $r2 +c\ r1 =o\ r2$
⟨*proof*⟩

lemma *csum-absorb1*: $[\![Cinfinite\ r2;\ r1 \le o\ r2]\!] \Longrightarrow r2 +c\ r1 =o\ r2$
⟨*proof*⟩

30.8 Exponentiation

definition *cexp* (**infixr** *^c 90*) **where**
$r1\ \hat{}c\ r2 \equiv |Func\ (Field\ r2)\ (Field\ r1)|$

lemma *Card-order-cexp*: $Card\text{-}order\ (r1\ \hat{}c\ r2)$
⟨*proof*⟩

lemma *cexp-mono'*:
assumes *1*: $p1 \le o\ r1$ **and** *2*: $p2 \le o\ r2$
and *n*: $Field\ p2 = \{\} \Longrightarrow Field\ r2 = \{\}$
shows $p1\ \hat{}c\ p2 \le o\ r1\ \hat{}c\ r2$
⟨*proof*⟩

lemma *cexp-mono*:
assumes *1*: $p1 \le o\ r1$ **and** *2*: $p2 \le o\ r2$
and *n*: $p2 =o\ czero \Longrightarrow r2 =o\ czero$ **and** *card*: $Card\text{-}order\ p2$
shows $p1\ \hat{}c\ p2 \le o\ r1\ \hat{}c\ r2$
⟨*proof*⟩

lemma *cexp-mono1*:
assumes *1*: *p1* $\leq$*o r1* **and** *q*: *Card-order q*
shows *p1 ^c q* $\leq$*o r1 ^c q*
⟨*proof*⟩

lemma *cexp-mono2'*:
assumes *2*: *p2* $\leq$*o r2* **and** *q*: *Card-order q*
and *n*: *Field p2* = {} $\Longrightarrow$ *Field r2* = {}
shows *q ^c p2* $\leq$*o q ^c r2*
⟨*proof*⟩

lemma *cexp-mono2*:
assumes *2*: *p2* $\leq$*o r2* **and** *q*: *Card-order q*
and *n*: *p2 =o czero* $\Longrightarrow$ *r2 =o czero* **and** *card*: *Card-order p2*
shows *q ^c p2* $\leq$*o q ^c r2*
⟨*proof*⟩

lemma *cexp-mono2-Cnotzero*:
assumes *p2* $\leq$*o r2 Card-order q Cnotzero p2*
shows *q ^c p2* $\leq$*o q ^c r2*
⟨*proof*⟩

lemma *cexp-cong*:
assumes *1*: *p1 =o r1* **and** *2*: *p2 =o r2*
and *Cr*: *Card-order r2*
and *Cp*: *Card-order p2*
shows *p1 ^c p2 =o r1 ^c r2*
⟨*proof*⟩

lemma *cexp-cong1*:
assumes *1*: *p1 =o r1* **and** *q*: *Card-order q*
shows *p1 ^c q =o r1 ^c q*
⟨*proof*⟩

lemma *cexp-cong2*:
assumes *2*: *p2 =o r2* **and** *q*: *Card-order q* **and** *p*: *Card-order p2*
shows *q ^c p2 =o q ^c r2*
⟨*proof*⟩

lemma *cexp-cone*:
assumes *Card-order r*
shows *r ^c cone =o r*
⟨*proof*⟩

lemma *cexp-cprod*:
assumes *r1*: *Card-order r1*
shows (*r1 ^c r2*) *^c r3 =o r1 ^c* (*r2* $*$*c r3*) (**is** *?L =o ?R*)
⟨*proof*⟩

lemma *cprod-infinite1'*: $\llbracket$*Cinfinite r*; *Cnotzero p*; $p \le o\ r\rrbracket \Longrightarrow r *c\ p =o\ r$
⟨*proof*⟩

lemma *cprod-infinite*: *Cinfinite r* $\Longrightarrow r *c\ r =o\ r$
⟨*proof*⟩

lemma *cexp-cprod-ordLeq*:
assumes *r1*: *Card-order r1* **and** *r2*: *Cinfinite r2*
and *r3*: *Cnotzero r3 r3* $\le o$ *r2*
shows (*r1* ^*c r2*) ^*c r3* $=o$ *r1* ^*c r2* (**is** *?L* $=o$ *?R*)
⟨*proof*⟩

lemma *Cnotzero-UNIV*: *Cnotzero* |*UNIV*|
⟨*proof*⟩

lemma *ordLess-ctwo-cexp*:
assumes *Card-order r*
shows *r* $<o$ *ctwo* ^*c r*
⟨*proof*⟩

lemma *ordLeq-cexp1*:
assumes *Cnotzero r Card-order q*
shows *q* $\le o$ *q* ^*c r*
⟨*proof*⟩

lemma *ordLeq-cexp2*:
assumes *ctwo* $\le o$ *q Card-order r*
shows *r* $\le o$ *q* ^*c r*
⟨*proof*⟩

lemma *cinfinite-cexp*: $\llbracket$*ctwo* $\le o$ *q*; *Cinfinite r*$\rrbracket \Longrightarrow$ *cinfinite* (*q* ^*c r*)
⟨*proof*⟩

lemma *Cinfinite-cexp*:
$\llbracket$*ctwo* $\le o$ *q*; *Cinfinite r*$\rrbracket \Longrightarrow$ *Cinfinite* (*q* ^*c r*)
⟨*proof*⟩

lemma *ctwo-ordLess-natLeq*: *ctwo* $<o$ *natLeq*
⟨*proof*⟩

lemma *ctwo-ordLess-Cinfinite*: *Cinfinite r* $\Longrightarrow$ *ctwo* $<o$ *r*
⟨*proof*⟩

lemma *ctwo-ordLeq-Cinfinite*:
assumes *Cinfinite r*
shows *ctwo* $\le o$ *r*
⟨*proof*⟩

lemma *Un-Cinfinite-bound*: $\llbracket |A| \le o\ r; |B| \le o\ r;$ *Cinfinite r*$\rrbracket \Longrightarrow |A \cup B| \le o\ r$

⟨*proof*⟩

lemma *UNION-Cinfinite-bound*: $[\![|I| \leq o\ r;\ \forall\, i \in I.\ |A\ i| \leq o\ r;\ Cinfinite\ r]\!] \Longrightarrow |\bigcup i \in I.\ A\ i| \leq o\ r$
⟨*proof*⟩

lemma *csum-cinfinite-bound*:
assumes $p \leq o\ r\ q \leq o\ r$ *Card-order p Card-order q Cinfinite r*
shows $p +c\ q \leq o\ r$
⟨*proof*⟩

lemma *cprod-cinfinite-bound*:
assumes $p \leq o\ r\ q \leq o\ r$ *Card-order p Card-order q Cinfinite r*
shows $p *c\ q \leq o\ r$
⟨*proof*⟩

lemma *cprod-csum-cexp*:
$r1 *c\ r2 \leq o\ (r1 +c\ r2)\ \hat{}c\ ctwo$
⟨*proof*⟩

lemma *Cfinite-cprod-Cinfinite*: $[\![Cfinite\ r;\ Cinfinite\ s]\!] \Longrightarrow r *c\ s \leq o\ s$
⟨*proof*⟩

lemma *cprod-cexp*: $(r *c\ s)\ \hat{}c\ t =o\ r\ \hat{}c\ t *c\ s\ \hat{}c\ t$
⟨*proof*⟩

lemma *cprod-cexp-csum-cexp-Cinfinite*:
assumes *t*: *Cinfinite t*
shows $(r *c\ s)\ \hat{}c\ t \leq o\ (r +c\ s)\ \hat{}c\ t$
⟨*proof*⟩

lemma *Cfinite-cexp-Cinfinite*:
assumes *s*: *Cfinite s* **and** *t*: *Cinfinite t*
shows $s\ \hat{}c\ t \leq o\ ctwo\ \hat{}c\ t$
⟨*proof*⟩

lemma *csum-Cfinite-cexp-Cinfinite*:
assumes *r*: *Card-order r* **and** *s*: *Cfinite s* **and** *t*: *Cinfinite t*
shows $(r +c\ s)\ \hat{}c\ t \leq o\ (r +c\ ctwo)\ \hat{}c\ t$
⟨*proof*⟩

lemma *Cinfinite-cardSuc*: *Cinfinite r* $\Longrightarrow$ *Cinfinite* (*cardSuc r*)
⟨*proof*⟩

lemma *cardSuc-UNION-Cinfinite*:
assumes *Cinfinite r relChain* (*cardSuc r*) *As* $B \leq$ (*UN i* : *Field* (*cardSuc r*). *As i*) $|B| <\!=o\ r$

shows $EX\ i : Field\ (cardSuc\ r).\ B \leq As\ i$
$\langle proof \rangle$

end

31 Function Definition Base

theory *Fun-Def-Base*
imports *Ctr-Sugar Set Wellfounded*
begin

$\langle ML \rangle$
named-theorems *termination-simp simplification rules for termination proofs*
$\langle ML \rangle$

end

32 Definition of Bounded Natural Functors

theory *BNF-Def*
imports *BNF-Cardinal-Arithmetic Fun-Def-Base*
keywords
print-bnfs :: *diag* **and**
bnf :: *thy-goal*
begin

lemma *Collect-case-prodD*: $x \in Collect\ (case\text{-}prod\ A) \Longrightarrow A\ (fst\ x)\ (snd\ x)$
$\langle proof \rangle$

inductive
rel-sum :: $('a \Rightarrow 'c \Rightarrow bool) \Rightarrow ('b \Rightarrow 'd \Rightarrow bool) \Rightarrow 'a + 'b \Rightarrow 'c + 'd \Rightarrow bool$
for *R1 R2*
where
$R1\ a\ c \Longrightarrow rel\text{-}sum\ R1\ R2\ (Inl\ a)\ (Inl\ c)$
$|\ R2\ b\ d \Longrightarrow rel\text{-}sum\ R1\ R2\ (Inr\ b)\ (Inr\ d)$

definition
rel-fun :: $('a \Rightarrow 'c \Rightarrow bool) \Rightarrow ('b \Rightarrow 'd \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow bool$
where
$rel\text{-}fun\ A\ B = (\lambda f\ g.\ \forall\ x\ y.\ A\ x\ y \longrightarrow B\ (f\ x)\ (g\ y))$

lemma *rel-funI* [*intro*]:
assumes $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ (f\ x)\ (g\ y)$
shows $rel\text{-}fun\ A\ B\ f\ g$
$\langle proof \rangle$

lemma *rel-funD*:

assumes *rel-fun* A B f g **and** A x y
shows B $(f\ x)$ $(g\ y)$
⟨*proof*⟩

lemma *rel-fun-mono*:
$\llbracket$ *rel-fun* X A f g; $\bigwedge x\ y.\ Y\ x\ y \longrightarrow X\ x\ y$; $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ x\ y$ $\rrbracket \Longrightarrow$ *rel-fun* Y B f g
⟨*proof*⟩

lemma *rel-fun-mono'* [*mono*]:
$\llbracket \bigwedge x\ y.\ Y\ x\ y \longrightarrow X\ x\ y$; $\bigwedge x\ y.\ A\ x\ y \longrightarrow B\ x\ y$ $\rrbracket \Longrightarrow$ *rel-fun* X A f g $\longrightarrow$ *rel-fun* Y B f g
⟨*proof*⟩

definition *rel-set* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$
where *rel-set* $R = (\lambda A\ B.\ (\forall x{\in}A.\ \exists y{\in}B.\ R\ x\ y) \wedge (\forall y{\in}B.\ \exists x{\in}A.\ R\ x\ y))$

lemma *rel-setI*:
assumes $\bigwedge x.\ x \in A \Longrightarrow \exists y{\in}B.\ R\ x\ y$
assumes $\bigwedge y.\ y \in B \Longrightarrow \exists x{\in}A.\ R\ x\ y$
shows *rel-set* R A B
⟨*proof*⟩

lemma *predicate2-transferD*:
$\llbracket$*rel-fun* $R1$ (*rel-fun* $R2$ $(op\ =)$) P Q; $a \in A$; $b \in B$; $A \subseteq \{(x,\ y).\ R1\ x\ y\}$; $B \subseteq \{(x,\ y).\ R2\ x\ y\}\rrbracket \Longrightarrow$
$P\ (fst\ a)\ (fst\ b) \longleftrightarrow Q\ (snd\ a)\ (snd\ b)$
⟨*proof*⟩

definition *collect* **where**
collect $F\ x = (\bigcup f \in F.\ f\ x)$

lemma *fstI*: $x = (y,\ z) \Longrightarrow fst\ x = y$
⟨*proof*⟩

lemma *sndI*: $x = (y,\ z) \Longrightarrow snd\ x = z$
⟨*proof*⟩

lemma *bijI'*: $\llbracket\bigwedge x\ y.\ (f\ x = f\ y) = (x = y);\ \bigwedge y.\ \exists x.\ y = f\ x\rrbracket \Longrightarrow bij\ f$
⟨*proof*⟩

definition *Gr* $A\ f = \{(a,\ f\ a) \mid a.\ a \in A\}$

definition *Grp* $A\ f = (\lambda a\ b.\ b = f\ a \wedge a \in A)$

definition *vimage2p* **where**
vimage2p $f\ g\ R = (\lambda x\ y.\ R\ (f\ x)\ (g\ y))$

lemma *collect-comp*: $collect\ F \circ g = collect\ ((\lambda f.\ f \circ g)\ `\ F)$
⟨*proof*⟩

definition *convol* (⟨(-,/ -)⟩) **where**
$\langle f,\ g\rangle \equiv \lambda a.\ (f\ a,\ g\ a)$

lemma *fst-convol*: $fst \circ \langle f,\ g\rangle = f$
⟨*proof*⟩

lemma *snd-convol*: $snd \circ \langle f,\ g\rangle = g$
⟨*proof*⟩

lemma *convol-mem-GrpI*:
$x \in A \Longrightarrow \langle id,\ g\rangle\ x \in (Collect\ (case\text{-}prod\ (Grp\ A\ g)))$
⟨*proof*⟩

definition *csquare* **where**
$csquare\ A\ f1\ f2\ p1\ p2 \longleftrightarrow (\forall\ a \in A.\ f1\ (p1\ a) = f2\ (p2\ a))$

lemma *eq-alt*: $op == Grp\ UNIV\ id$
⟨*proof*⟩

lemma *leq-conversepI*: $R = op = \Longrightarrow R \leq R\hat{\ }{-}{-}1$
⟨*proof*⟩

lemma *leq-OOI*: $R = op = \Longrightarrow R \leq R\ OO\ R$
⟨*proof*⟩

lemma *OO-Grp-alt*: $(Grp\ A\ f)\hat{\ }{-}{-}1\ OO\ Grp\ A\ g = (\lambda x\ y.\ \exists z.\ z \in A \wedge f\ z = x \wedge g\ z = y)$
⟨*proof*⟩

lemma *Grp-UNIV-id*: $f = id \Longrightarrow (Grp\ UNIV\ f)\hat{\ }{-}{-}1\ OO\ Grp\ UNIV\ f = Grp\ UNIV\ f$
⟨*proof*⟩

lemma *Grp-UNIV-idI*: $x = y \Longrightarrow Grp\ UNIV\ id\ x\ y$
⟨*proof*⟩

lemma *Grp-mono*: $A \leq B \Longrightarrow Grp\ A\ f \leq Grp\ B\ f$
⟨*proof*⟩

lemma *GrpI*: $[\![f\ x = y;\ x \in A]\!] \Longrightarrow Grp\ A\ f\ x\ y$
⟨*proof*⟩

lemma *GrpE*: $Grp\ A\ f\ x\ y \Longrightarrow ([\![f\ x = y;\ x \in A]\!] \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *Collect-case-prod-Grp-eqD*: $z \in Collect\ (case\text{-}prod\ (Grp\ A\ f)) \Longrightarrow (f\ \circ$

fst) z = *snd* z
  ⟨*proof*⟩

lemma *Collect-case-prod-Grp-in*: $z \in$ *Collect* (*case-prod* (*Grp* A f)) $\Longrightarrow$ *fst* $z \in A$
  ⟨*proof*⟩

definition *pick-middlep* P Q a c = (*SOME* b. P a b $\wedge$ Q b c)

lemma *pick-middlep*:
  (P *OO* Q) a c $\Longrightarrow$ P a (*pick-middlep* P Q a c) $\wedge$ Q (*pick-middlep* P Q a c) c
  ⟨*proof*⟩

definition *fstOp* **where**
  fstOp P Q *ac* = (*fst ac*, *pick-middlep* P Q (*fst ac*) (*snd ac*))

definition *sndOp* **where**
  sndOp P Q *ac* = (*pick-middlep* P Q (*fst ac*) (*snd ac*), (*snd ac*))

lemma *fstOp-in*: *ac* $\in$ *Collect* (*case-prod* (P *OO* Q)) $\Longrightarrow$ *fstOp* P Q *ac* $\in$ *Collect* (*case-prod* P)
  ⟨*proof*⟩

lemma *fst-fstOp*: *fst bc* = (*fst* $\circ$ *fstOp* P Q) *bc*
  ⟨*proof*⟩

lemma *snd-sndOp*: *snd bc* = (*snd* $\circ$ *sndOp* P Q) *bc*
  ⟨*proof*⟩

lemma *sndOp-in*: *ac* $\in$ *Collect* (*case-prod* (P *OO* Q)) $\Longrightarrow$ *sndOp* P Q *ac* $\in$ *Collect* (*case-prod* Q)
  ⟨*proof*⟩

lemma *csquare-fstOp-sndOp*:
  csquare (*Collect* (f (P *OO* Q))) *snd fst* (*fstOp* P Q) (*sndOp* P Q)
  ⟨*proof*⟩

lemma *snd-fst-flip*: *snd xy* = (*fst* $\circ$ (%(x, y). (y, x))) *xy*
  ⟨*proof*⟩

lemma *fst-snd-flip*: *fst xy* = (*snd* $\circ$ (%(x, y). (y, x))) *xy*
  ⟨*proof*⟩

lemma *flip-pred*: $A \subseteq$ *Collect* (*case-prod* (R ^−−*1*)) $\Longrightarrow$ (%(x, y). (y, x)) ‘ $A \subseteq$ *Collect* (*case-prod* R)
  ⟨*proof*⟩

lemma *predicate2-eqD*: $A = B \Longrightarrow A\ a\ b \longleftrightarrow B\ a\ b$
  ⟨*proof*⟩

lemma *case-sum-o-inj*: *case-sum f g* $\circ$ *Inl* = *f case-sum f g* $\circ$ *Inr* = *g*
⟨*proof*⟩

lemma *map-sum-o-inj*: *map-sum f g o Inl* = *Inl o f map-sum f g o Inr* = *Inr o g*
⟨*proof*⟩

lemma *card-order-csum-cone-cexp-def*:
card-order r $\Longrightarrow$ (|*A1*| +*c cone*) ˆ*c r* = |*Func UNIV* (*Inl* ‘ *A1* $\cup$ {*Inr* ()})|
⟨*proof*⟩

lemma *If-the-inv-into-in-Func*:
⟦*inj-on g C*; *C* $\subseteq$ *B* $\cup$ {*x*}⟧ $\Longrightarrow$
(λ*i*. *if i* $\in$ *g* ‘ *C then the-inv-into C g i else x*) $\in$ *Func UNIV* (*B* $\cup$ {*x*})
⟨*proof*⟩

lemma *If-the-inv-into-f-f*:
⟦*i* $\in$ *C*; *inj-on g C*⟧ $\Longrightarrow$ ((λ*i*. *if i* $\in$ *g* ‘ *C then the-inv-into C g i else x*) $\circ$ *g*) *i* = *id i*
⟨*proof*⟩

lemma *the-inv-f-o-f-id*: *inj f* $\Longrightarrow$ (*the-inv f* $\circ$ *f*) *z* = *id z*
⟨*proof*⟩

lemma *vimage2pI*: *R* (*f x*) (*g y*) $\Longrightarrow$ *vimage2p f g R x y*
⟨*proof*⟩

lemma *rel-fun-iff-leq-vimage2p*: (*rel-fun R S*) *f g* = (*R* $\leq$ *vimage2p f g S*)
⟨*proof*⟩

lemma *convol-image-vimage2p*: ⟨*f* $\circ$ *fst*, *g* $\circ$ *snd*⟩ ‘ *Collect* (*case-prod* (*vimage2p f g R*)) $\subseteq$ *Collect* (*case-prod R*)
⟨*proof*⟩

lemma *vimage2p-Grp*: *vimage2p f g P* = *Grp UNIV f OO P OO* (*Grp UNIV g*)$^{-1-1}$
⟨*proof*⟩

lemma *subst-Pair*: *P x y* $\Longrightarrow$ *a* = (*x*, *y*) $\Longrightarrow$ *P* (*fst a*) (*snd a*)
⟨*proof*⟩

lemma *comp-apply-eq*: *f* (*g x*) = *h* (*k x*) $\Longrightarrow$ (*f* $\circ$ *g*) *x* = (*h* $\circ$ *k*) *x*
⟨*proof*⟩

lemma *refl-ge-eq*: ($\bigwedge$*x*. *R x x*) $\Longrightarrow$ *op* = $\leq$ *R*
⟨*proof*⟩

lemma *ge-eq-refl*: *op* = $\leq$ *R* $\Longrightarrow$ *R x x*
⟨*proof*⟩

lemma *reflp-eq*: $reflp\ R = (op = \leq R)$
⟨*proof*⟩

lemma *transp-relcompp*: $transp\ r \longleftrightarrow r\ OO\ r \leq r$
⟨*proof*⟩

lemma *symp-conversep*: $symp\ R = (R^{-1-1} \leq R)$
⟨*proof*⟩

lemma *diag-imp-eq-le*: $(\bigwedge x.\ x \in A \Longrightarrow R\ x\ x) \Longrightarrow \forall\, x\ y.\ x \in A \longrightarrow y \in A \longrightarrow x = y \longrightarrow R\ x\ y$
⟨*proof*⟩

definition *eq-onp* :: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where $eq\text{-}onp\ R = (\lambda x\ y.\ R\ x \wedge x = y)$

lemma *eq-onp-Grp*: $eq\text{-}onp\ P = BNF\text{-}Def.Grp\ (Collect\ P)\ id$
⟨*proof*⟩

lemma *eq-onp-to-eq*: $eq\text{-}onp\ P\ x\ y \Longrightarrow x = y$
⟨*proof*⟩

lemma *eq-onp-top-eq-eq*: $eq\text{-}onp\ top = op =$
⟨*proof*⟩

lemma *eq-onp-same-args*: $eq\text{-}onp\ P\ x\ x = P\ x$
⟨*proof*⟩

lemma *eq-onp-eqD*: $eq\text{-}onp\ P = Q \Longrightarrow P\ x = Q\ x\ x$
⟨*proof*⟩

lemma *Ball-Collect*: $Ball\ A\ P = (A \subseteq (Collect\ P))$
⟨*proof*⟩

lemma *eq-onp-mono0*: $\forall\, x{\in}A.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall\, x{\in}A.\ \forall\, y{\in}A.\ eq\text{-}onp\ P\ x\ y \longrightarrow eq\text{-}onp\ Q\ x\ y$
⟨*proof*⟩

lemma *eq-onp-True*: $eq\text{-}onp\ (\lambda\text{-}.\ True) = (op =)$
⟨*proof*⟩

lemma *Ball-image-comp*: $Ball\ (f\ `\ A)\ g = Ball\ A\ (g\ o\ f)$
⟨*proof*⟩

lemma *rel-fun-Collect-case-prodD*:
$rel\text{-}fun\ A\ B\ f\ g \Longrightarrow X \subseteq Collect\ (case\text{-}prod\ A) \Longrightarrow x \in X \Longrightarrow B\ ((f\ o\ fst)\ x)\ ((g\ o\ snd)\ x)$
⟨*proof*⟩

lemma *eq-onp-mono-iff*: *eq-onp* $P \leq$ *eq-onp* $Q \longleftrightarrow P \leq Q$
⟨*proof*⟩

⟨*ML*⟩

end

33 Composition of Bounded Natural Functors

theory *BNF-Composition*
imports *BNF-Def*
keywords
copy-bnf :: *thy-decl* **and**
lift-bnf :: *thy-goal*
begin

lemma *ssubst-mem*: $[\![t = s; s \in X]\!] \Longrightarrow t \in X$
⟨*proof*⟩

lemma *empty-natural*: $(\lambda\text{-}. \{\})$ *o* f = *image* g *o* $(\lambda\text{-}. \{\})$
⟨*proof*⟩

lemma *Union-natural*: *Union o image* (*image* f) = *image* f *o Union*
⟨*proof*⟩

lemma *in-Union-o-assoc*: $x \in$ (*Union o gset o gmap*) $A \Longrightarrow x \in$ (*Union o* (*gset o gmap*)) A
⟨*proof*⟩

lemma *comp-single-set-bd*:
assumes *fbd-Card-order*: *Card-order fbd* **and**
fset-bd: $\bigwedge x.$ |*fset* x| $\leq o$ *fbd* **and**
gset-bd: $\bigwedge x.$ |*gset* x| $\leq o$ *gbd*
shows |$\bigcup$(*fset* ‘ *gset* x)| $\leq o$ *gbd* $*c$ *fbd*
⟨*proof*⟩

lemma *csum-dup*: *cinfinite* $r \Longrightarrow$ *Card-order* $r \Longrightarrow p +c\ p' =o\ r +c\ r \Longrightarrow p +c\ p' =o\ r$
⟨*proof*⟩

lemma *cprod-dup*: *cinfinite* $r \Longrightarrow$ *Card-order* $r \Longrightarrow p *c\ p' =o\ r *c\ r \Longrightarrow p *c\ p' =o\ r$
⟨*proof*⟩

lemma *Union-image-insert*: $\bigcup(f$ ‘ *insert* a $B) = f\ a \cup \bigcup(f$ ‘ $B)$
⟨*proof*⟩

lemma *Union-image-empty*: $A \cup \bigcup(f$ ‘ $\{\}) = A$
⟨*proof*⟩

lemma *image-o-collect*: $collect\ ((\lambda f.\ image\ g\ o\ f)\ `\ F) = image\ g\ o\ collect\ F$
⟨*proof*⟩

lemma *conj-subset-def*: $A \subseteq \{x.\ P\ x \wedge Q\ x\} = (A \subseteq \{x.\ P\ x\} \wedge A \subseteq \{x.\ Q\ x\})$
⟨*proof*⟩

lemma *UN-image-subset*: $\bigcup (f\ `\ g\ x) \subseteq X = (g\ x \subseteq \{x.\ f\ x \subseteq X\})$
⟨*proof*⟩

lemma *comp-set-bd-Union-o-collect*: $|\bigcup\bigcup ((\lambda f.\ f\ x)\ `\ X)| \le o\ hbd \Longrightarrow |(Union \circ collect\ X)\ x| \le o\ hbd$
⟨*proof*⟩

lemma *Collect-inj*: $Collect\ P = Collect\ Q \Longrightarrow P = Q$
⟨*proof*⟩

lemma *Grp-fst-snd*: $(Grp\ (Collect\ (case\text{-}prod\ R))\ fst)\hat{}{-}{-}1\ OO\ Grp\ (Collect\ (case\text{-}prod\ R))\ snd = R$
⟨*proof*⟩

lemma *OO-Grp-cong*: $A = B \Longrightarrow (Grp\ A\ f)\hat{}{-}{-}1\ OO\ Grp\ A\ g = (Grp\ B\ f)\hat{}{-}{-}1\ OO\ Grp\ B\ g$
⟨*proof*⟩

lemma *vimage2p-relcompp-mono*: $R\ OO\ S \le T \Longrightarrow$
$vimage2p\ f\ g\ R\ OO\ vimage2p\ g\ h\ S \le vimage2p\ f\ h\ T$
⟨*proof*⟩

lemma *type-copy-map-cong0*: $M\ (g\ x) = N\ (h\ x) \Longrightarrow (f\ o\ M\ o\ g)\ x = (f\ o\ N\ o\ h)\ x$
⟨*proof*⟩

lemma *type-copy-set-bd*: $(\bigwedge y.\ |S\ y| \le o\ bd) \Longrightarrow |(S\ o\ Rep)\ x| \le o\ bd$
⟨*proof*⟩

lemma *vimage2p-cong*: $R = S \Longrightarrow vimage2p\ f\ g\ R = vimage2p\ f\ g\ S$
⟨*proof*⟩

lemma *Ball-comp-iff*: $(\lambda x.\ Ball\ (A\ x)\ f)\ o\ g = (\lambda x.\ Ball\ ((A\ o\ g)\ x)\ f)$
⟨*proof*⟩

lemma *conj-comp-iff*: $(\lambda x.\ P\ x \wedge Q\ x)\ o\ g = (\lambda x.\ (P\ o\ g)\ x \wedge (Q\ o\ g)\ x)$
⟨*proof*⟩

context
fixes *Rep Abs*
assumes *type-copy*: *type-definition Rep Abs UNIV*
begin

lemma *type-copy-map-id0*: $M = id \Longrightarrow Abs\ o\ M\ o\ Rep = id$
⟨*proof*⟩

lemma *type-copy-map-comp0*: $M = M1\ o\ M2 \Longrightarrow f\ o\ M\ o\ g = (f\ o\ M1\ o\ Rep)\ o\ (Abs\ o\ M2\ o\ g)$
⟨*proof*⟩

lemma *type-copy-set-map0*: $S\ o\ M = image\ f\ o\ S' \Longrightarrow (S\ o\ Rep)\ o\ (Abs\ o\ M\ o\ g) = image\ f\ o\ (S'\ o\ g)$
⟨*proof*⟩

lemma *type-copy-wit*: $x \in (S\ o\ Rep)\ (Abs\ y) \Longrightarrow x \in S\ y$
⟨*proof*⟩

lemma *type-copy-vimage2p-Grp-Rep*: $vimage2p\ f\ Rep\ (Grp\ (Collect\ P)\ h) =$
$Grp\ (Collect\ (\lambda x.\ P\ (f\ x)))\ (Abs\ o\ h\ o\ f)$
⟨*proof*⟩

lemma *type-copy-vimage2p-Grp-Abs*:
$\bigwedge h.\ vimage2p\ g\ Abs\ (Grp\ (Collect\ P)\ h) = Grp\ (Collect\ (\lambda x.\ P\ (g\ x)))\ (Rep\ o\ h\ o\ g)$
⟨*proof*⟩

lemma *type-copy-ex-RepI*: $(\exists\, b.\ F\ b) = (\exists\, b.\ F\ (Rep\ b))$
⟨*proof*⟩

lemma *vimage2p-relcompp-converse*:
$vimage2p\ f\ g\ (R\hat{}{--}1\ OO\ S) = (vimage2p\ Rep\ f\ R)\hat{}{--}1\ OO\ vimage2p\ Rep\ g\ S$
⟨*proof*⟩

end

bnf *DEADID*: $'a$
map: $id :: 'a \Rightarrow 'a$
bd: *natLeq*
rel: $op = :: 'a \Rightarrow 'a \Rightarrow bool$
⟨*proof*⟩

definition *id-bnf* :: $'a \Rightarrow 'a$ **where**
id-bnf $\equiv (\lambda x.\ x)$

lemma *id-bnf-apply*: *id-bnf* $x = x$
⟨*proof*⟩

bnf *ID*: $'a$
map: *id-bnf* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
sets: $\lambda x.\ \{x\}$
bd: *natLeq*

rel: *id-bnf* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
pred: *id-bnf* :: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$
⟨*proof*⟩

lemma *type-definition-id-bnf-UNIV*: *type-definition id-bnf id-bnf UNIV*
⟨*proof*⟩

⟨*ML*⟩

hide-fact
DEADID.inj-map DEADID.inj-map-strong DEADID.map-comp DEADID.map-cong DEADID.map-cong0
DEADID.map-cong-simp DEADID.map-id DEADID.map-id0 DEADID.map-ident DEADID.map-transfer
DEADID.rel-Grp DEADID.rel-compp DEADID.rel-compp-Grp DEADID.rel-conversep DEADID.rel-eq
DEADID.rel-flip DEADID.rel-map DEADID.rel-mono DEADID.rel-transfer
ID.inj-map ID.inj-map-strong ID.map-comp ID.map-cong ID.map-cong0 ID.map-cong-simp ID.map-id
ID.map-id0 ID.map-ident ID.map-transfer ID.rel-Grp ID.rel-compp ID.rel-compp-Grp ID.rel-conversep
ID.rel-eq ID.rel-flip ID.rel-map ID.rel-mono ID.rel-transfer ID.set-map ID.set-transfer

end

34 Registration of Basic Types as Bounded Natural Functors

theory *Basic-BNFs*
imports *BNF-Def*
begin

inductive-set *setl* :: $'a + 'b \Rightarrow 'a$ *set* **for** $s :: 'a + 'b$ **where**
$s = Inl\ x \Longrightarrow x \in setl\ s$
inductive-set *setr* :: $'a + 'b \Rightarrow 'b$ *set* **for** $s :: 'a + 'b$ **where**
$s = Inr\ x \Longrightarrow x \in setr\ s$

lemma *sum-set-defs*[*code*]:
$setl = (\lambda x.\ case\ x\ of\ Inl\ z => \{z\}\ |\ \text{-} => \{\})$
$setr = (\lambda x.\ case\ x\ of\ Inr\ z => \{z\}\ |\ \text{-} => \{\})$
⟨*proof*⟩

lemma *rel-sum-simps*[*code*, *simp*]:
rel-sum R1 R2 (Inl a1) (Inl b1) = R1 a1 b1
rel-sum R1 R2 (Inl a1) (Inr b2) = False
rel-sum R1 R2 (Inr a2) (Inl b1) = False
rel-sum R1 R2 (Inr a2) (Inr b2) = R2 a2 b2
⟨*proof*⟩

inductive
  $pred\text{-}sum :: ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a + 'b \Rightarrow bool$ **for** $P1\ P2$
where
  $P1\ a \Longrightarrow pred\text{-}sum\ P1\ P2\ (Inl\ a)$
| $P2\ b \Longrightarrow pred\text{-}sum\ P1\ P2\ (Inr\ b)$

lemma *pred-sum-inject*[*code*, *simp*]:
  $pred\text{-}sum\ P1\ P2\ (Inl\ a) \longleftrightarrow P1\ a$
  $pred\text{-}sum\ P1\ P2\ (Inr\ b) \longleftrightarrow P2\ b$
  ⟨*proof*⟩

bnf $'a + 'b$
  map: *map-sum*
  sets: *setl setr*
  bd: *natLeq*
  wits: *Inl Inr*
  rel: *rel-sum*
  pred: *pred-sum*
⟨*proof*⟩

inductive-set $fsts :: 'a \times 'b \Rightarrow 'a\ set$ **for** $p :: 'a \times 'b$ **where**
  $fst\ p \in fsts\ p$
inductive-set $snds :: 'a \times 'b \Rightarrow 'b\ set$ **for** $p :: 'a \times 'b$ **where**
  $snd\ p \in snds\ p$

lemma *prod-set-defs*[*code*]: $fsts = (\lambda p.\ \{fst\ p\})\ snds = (\lambda p.\ \{snd\ p\})$
  ⟨*proof*⟩

inductive
  $rel\text{-}prod :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('c \Rightarrow 'd \Rightarrow bool) \Rightarrow 'a \times 'c \Rightarrow 'b \times 'd \Rightarrow bool$
for $R1\ R2$
where
  $[\![R1\ a\ b;\ R2\ c\ d]\!] \Longrightarrow rel\text{-}prod\ R1\ R2\ (a,\ c)\ (b,\ d)$

inductive
  $pred\text{-}prod :: ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \times 'b \Rightarrow bool$ **for** $P1\ P2$
where
  $[\![P1\ a;\ P2\ b]\!] \Longrightarrow pred\text{-}prod\ P1\ P2\ (a,\ b)$

lemma *rel-prod-inject* [*code*, *simp*]:
  $rel\text{-}prod\ R1\ R2\ (a,\ b)\ (c,\ d) \longleftrightarrow R1\ a\ c \wedge R2\ b\ d$
  ⟨*proof*⟩

lemma *pred-prod-inject* [*code*, *simp*]:
  $pred\text{-}prod\ P1\ P2\ (a,\ b) \longleftrightarrow P1\ a \wedge P2\ b$
  ⟨*proof*⟩

lemma *rel-prod-conv*:

rel-prod R1 R2 = $(\lambda(a, b)\ (c, d).\ R1\ a\ c \wedge R2\ b\ d)$
⟨*proof*⟩

definition
pred-fun :: $('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where
pred-fun A B = $(\lambda f.\ \forall x.\ A\ x \longrightarrow B\ (f\ x))$

lemma *pred-funI*: $(\bigwedge x.\ A\ x \Longrightarrow B\ (f\ x)) \Longrightarrow$ *pred-fun A B f*
⟨*proof*⟩

bnf $'a \times 'b$
map: *map-prod*
sets: *fsts snds*
bd: *natLeq*
rel: *rel-prod*
pred: *pred-prod*
⟨*proof*⟩

bnf $'a \Rightarrow 'b$
map: *op* $\circ$
sets: *range*
bd: *natLeq* +*c* |*UNIV* :: *'a set*|
rel: *rel-fun op* =
pred: *pred-fun* (λ-. *True*)
⟨*proof*⟩

end

35 Shared Fixpoint Operations on Bounded Natural Functors

theory *BNF-Fixpoint-Base*
imports *BNF-Composition Basic-BNFs*
begin

lemma *conj-imp-eq-imp-imp*: $(P \wedge Q \Longrightarrow PROP\ R) \equiv (P \Longrightarrow Q \Longrightarrow PROP\ R)$
⟨*proof*⟩

lemma *predicate2D-conj*: $P \leq Q \wedge R \Longrightarrow R \wedge (P\ x\ y \longrightarrow Q\ x\ y)$
⟨*proof*⟩

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

lemma *case-unit-Unity*: $(case\ u\ of\ () \Rightarrow f) = f$
⟨*proof*⟩

lemma *case-prod-Pair-iden*: $(case\ p\ of\ (x,\ y) \Rightarrow (x,\ y)) = p$
⟨*proof*⟩

lemma *unit-all-impI*: $(P\ () \Longrightarrow Q\ ()) \Longrightarrow \forall x.\ P\ x \longrightarrow Q\ x$
⟨*proof*⟩

lemma *pointfree-idE*: $f \circ g = id \Longrightarrow f\ (g\ x) = x$
⟨*proof*⟩

lemma *o-bij*:
assumes *gf*: $g \circ f = id$ **and** *fg*: $f \circ g = id$
shows *bij f*
⟨*proof*⟩

lemma *case-sum-step*:
$case\text{-}sum\ (case\text{-}sum\ f'\ g')\ g\ (Inl\ p) = case\text{-}sum\ f'\ g'\ p$
$case\text{-}sum\ f\ (case\text{-}sum\ f'\ g')\ (Inr\ p) = case\text{-}sum\ f'\ g'\ p$
⟨*proof*⟩

lemma *obj-one-pointE*: $\forall x.\ s = x \longrightarrow P \Longrightarrow P$
⟨*proof*⟩

lemma *type-copy-obj-one-point-absE*:
assumes $type\text{-}definition\ Rep\ Abs\ UNIV\ \forall x.\ s = Abs\ x \longrightarrow P$ **shows** P
⟨*proof*⟩

lemma *obj-sumE-f*:
assumes $\forall x.\ s = f\ (Inl\ x) \longrightarrow P\ \forall x.\ s = f\ (Inr\ x) \longrightarrow P$
shows $\forall x.\ s = f\ x \longrightarrow P$
⟨*proof*⟩

lemma *case-sum-if*:
$case\text{-}sum\ f\ g\ (if\ p\ then\ Inl\ x\ else\ Inr\ y) = (if\ p\ then\ f\ x\ else\ g\ y)$
⟨*proof*⟩

lemma *prod-set-simps*[*simp*]:
$fsts\ (x,\ y) = \{x\}$
$snds\ (x,\ y) = \{y\}$
⟨*proof*⟩

lemma *sum-set-simps*[*simp*]:
$setl\ (Inl\ x) = \{x\}$
$setl\ (Inr\ x) = \{\}$
$setr\ (Inl\ x) = \{\}$
$setr\ (Inr\ x) = \{x\}$
⟨*proof*⟩

lemma *Inl-Inr-False*: $(Inl\ x = Inr\ y) = False$
⟨*proof*⟩

lemma *Inr-Inl-False*: $(Inr\ x = Inl\ y) = False$
⟨*proof*⟩

lemma *spec2*: $\forall x\ y.\ P\ x\ y \Longrightarrow P\ x\ y$
⟨*proof*⟩

lemma *rewriteR-comp-comp*: $[\![g \circ h = r]\!] \Longrightarrow f \circ g \circ h = f \circ r$
⟨*proof*⟩

lemma *rewriteR-comp-comp2*: $[\![g \circ h = r1 \circ r2;\ f \circ r1 = l]\!] \Longrightarrow f \circ g \circ h = l \circ r2$
⟨*proof*⟩

lemma *rewriteL-comp-comp*: $[\![f \circ g = l]\!] \Longrightarrow f \circ (g \circ h) = l \circ h$
⟨*proof*⟩

lemma *rewriteL-comp-comp2*: $[\![f \circ g = l1 \circ l2;\ l2 \circ h = r]\!] \Longrightarrow f \circ (g \circ h) = l1 \circ r$
⟨*proof*⟩

lemma *convol-o*: $\langle f,\ g\rangle \circ h = \langle f \circ h,\ g \circ h\rangle$
⟨*proof*⟩

lemma *map-prod-o-convol*: $map\text{-}prod\ h1\ h2 \circ \langle f,\ g\rangle = \langle h1 \circ f,\ h2 \circ g\rangle$
⟨*proof*⟩

lemma *map-prod-o-convol-id*: $(map\text{-}prod\ f\ id \circ \langle id,\ g\rangle)\ x = \langle id \circ f,\ g\rangle\ x$
⟨*proof*⟩

lemma *o-case-sum*: $h \circ case\text{-}sum\ f\ g = case\text{-}sum\ (h \circ f)\ (h \circ g)$
⟨*proof*⟩

lemma *case-sum-o-map-sum*: $case\text{-}sum\ f\ g \circ map\text{-}sum\ h1\ h2 = case\text{-}sum\ (f \circ h1)\ (g \circ h2)$
⟨*proof*⟩

lemma *case-sum-o-map-sum-id*: $(case\text{-}sum\ id\ g \circ map\text{-}sum\ f\ id)\ x = case\text{-}sum\ (f \circ id)\ g\ x$
⟨*proof*⟩

lemma *rel-fun-def-butlast*:
$rel\text{-}fun\ R\ (rel\text{-}fun\ S\ T)\ f\ g = (\forall x\ y.\ R\ x\ y \longrightarrow (rel\text{-}fun\ S\ T)\ (f\ x)\ (g\ y))$
⟨*proof*⟩

lemma *subst-eq-imp*: $(\forall a\ b.\ a = b \longrightarrow P\ a\ b) \equiv (\forall a.\ P\ a\ a)$
⟨*proof*⟩

lemma *eq-subset*: $op = \le (\lambda a\ b.\ P\ a\ b \vee a = b)$

⟨*proof*⟩

lemma *eq-le-Grp-id-iff*: $(op = \leq Grp\ (Collect\ R)\ id) = (All\ R)$
⟨*proof*⟩

lemma *Grp-id-mono-subst*: $(\bigwedge x\ y.\ Grp\ P\ id\ x\ y \Longrightarrow Grp\ Q\ id\ (f\ x)\ (f\ y)) \equiv (\bigwedge x.\ x \in P \Longrightarrow f\ x \in Q)$
⟨*proof*⟩

lemma *vimage2p-mono*: $vimage2p\ f\ g\ R\ x\ y \Longrightarrow R \leq S \Longrightarrow vimage2p\ f\ g\ S\ x\ y$
⟨*proof*⟩

lemma *vimage2p-refl*: $(\bigwedge x.\ R\ x\ x) \Longrightarrow vimage2p\ f\ f\ R\ x\ x$
⟨*proof*⟩

lemma
assumes *type-definition Rep Abs UNIV*
shows *type-copy-Rep-o-Abs*: $Rep \circ Abs = id$ **and** *type-copy-Abs-o-Rep*: $Abs \circ Rep = id$
⟨*proof*⟩

lemma *type-copy-map-comp0-undo*:
assumes *type-definition Rep Abs UNIV*
type-definition Rep′ Abs′ UNIV
type-definition Rep″ Abs″ UNIV
shows $Abs' \circ M \circ Rep'' = (Abs' \circ M1 \circ Rep) \circ (Abs \circ M2 \circ Rep'') \Longrightarrow M1 \circ M2 = M$
⟨*proof*⟩

lemma *vimage2p-id*: $vimage2p\ id\ id\ R = R$
⟨*proof*⟩

lemma *vimage2p-comp*: $vimage2p\ (f1 \circ f2)\ (g1 \circ g2) = vimage2p\ f2\ g2 \circ vimage2p\ f1\ g1$
⟨*proof*⟩

lemma *vimage2p-rel-fun*: $rel\text{-}fun\ (vimage2p\ f\ g\ R)\ R\ f\ g$
⟨*proof*⟩

lemma *fun-cong-unused-0*: $f = (\lambda x.\ g) \Longrightarrow f\ (\lambda x.\ 0) = g$
⟨*proof*⟩

lemma *inj-on-convol-ident*: $inj\text{-}on\ (\lambda x.\ (x, f\ x))\ X$
⟨*proof*⟩

lemma *map-sum-if-distrib-then*:
$\bigwedge f\ g\ e\ x\ y.\ map\text{-}sum\ f\ g\ (if\ e\ then\ Inl\ x\ else\ y) = (if\ e\ then\ Inl\ (f\ x)\ else\ map\text{-}sum\ f\ g\ y)$
$\bigwedge f\ g\ e\ x\ y.\ map\text{-}sum\ f\ g\ (if\ e\ then\ Inr\ x\ else\ y) = (if\ e\ then\ Inr\ (g\ x)\ else$

map-sum f g y)
⟨*proof*⟩

lemma *map-sum-if-distrib-else*:
$\bigwedge f\ g\ e\ x\ y.$ *map-sum f g* (*if e then x else Inl y*) = (*if e then map-sum f g x else Inl* (*f y*))
$\bigwedge f\ g\ e\ x\ y.$ *map-sum f g* (*if e then x else Inr y*) = (*if e then map-sum f g x else Inr* (*g y*))
⟨*proof*⟩

lemma *case-prod-app*: *case-prod f x y* = *case-prod* ($\lambda l\ r.$ *f l r y*) *x*
⟨*proof*⟩

lemma *case-sum-map-sum*: *case-sum l r* (*map-sum f g x*) = *case-sum* ($l \circ f$) ($r \circ g$) *x*
⟨*proof*⟩

lemma *case-sum-transfer*:
rel-fun (*rel-fun R T*) (*rel-fun* (*rel-fun S T*) (*rel-fun* (*rel-sum R S*) *T*)) *case-sum case-sum*
⟨*proof*⟩

lemma *case-prod-map-prod*: *case-prod h* (*map-prod f g x*) = *case-prod* ($\lambda l\ r.$ *h* (*f l*) (*g r*)) *x*
⟨*proof*⟩

lemma *case-prod-o-map-prod*: *case-prod f* $\circ$ *map-prod g1 g2* = *case-prod* ($\lambda l\ r.$ *f* (*g1 l*) (*g2 r*))
⟨*proof*⟩

lemma *case-prod-transfer*:
(*rel-fun* (*rel-fun A* (*rel-fun B C*)) (*rel-fun* (*rel-prod A B*) *C*)) *case-prod case-prod*
⟨*proof*⟩

lemma *eq-ifI*: ($P \longrightarrow t = u1$) $\Longrightarrow$ ($\neg\ P \longrightarrow t = u2$) $\Longrightarrow$ t = (*if P then u1 else u2*)
⟨*proof*⟩

lemma *comp-transfer*:
rel-fun (*rel-fun B C*) (*rel-fun* (*rel-fun A B*) (*rel-fun A C*)) (*op* $\circ$) (*op* $\circ$)
⟨*proof*⟩

lemma *If-transfer*: *rel-fun* (*op* =) (*rel-fun A* (*rel-fun A A*)) *If If*
⟨*proof*⟩

lemma *Abs-transfer*:
assumes *type-copy1*: *type-definition Rep1 Abs1 UNIV*
assumes *type-copy2*: *type-definition Rep2 Abs2 UNIV*
shows *rel-fun R* (*vimage2p Rep1 Rep2 R*) *Abs1 Abs2*

⟨proof⟩

lemma *Inl-transfer*:
rel-fun S (*rel-sum S T*) *Inl Inl*
⟨proof⟩

lemma *Inr-transfer*:
rel-fun T (*rel-sum S T*) *Inr Inr*
⟨proof⟩

lemma *Pair-transfer*: *rel-fun A* (*rel-fun B* (*rel-prod A B*)) *Pair Pair*
⟨proof⟩

lemma *eq-onp-live-step*: $x = y \Longrightarrow$ *eq-onp* $P\ a\ a \wedge x \longleftrightarrow P\ a \wedge y$
⟨proof⟩

lemma *top-conj*: *top* $x \wedge P \longleftrightarrow P$ $P \wedge$ *top* $x \longleftrightarrow P$
⟨proof⟩

lemma *fst-convol'*: *fst* (⟨*f*, *g*⟩ *x*) = *f x*
⟨proof⟩

lemma *snd-convol'*: *snd* (⟨*f*, *g*⟩ *x*) = *g x*
⟨proof⟩

lemma *convol-expand-snd*: *fst o f* = *g* $\Longrightarrow$ ⟨*g*, *snd o f*⟩ = *f*
⟨proof⟩

lemma *convol-expand-snd'*:
assumes (*fst o f* = *g*)
shows *h* = *snd o f* $\longleftrightarrow$ ⟨*g*, *h*⟩ = *f*
⟨proof⟩

lemma *case-sum-expand-Inr-pointfree*: *f o Inl* = *g* $\Longrightarrow$ *case-sum g* (*f o Inr*) = *f*
⟨proof⟩

lemma *case-sum-expand-Inr'*: *f o Inl* = *g* $\Longrightarrow$ *h* = *f o Inr* $\longleftrightarrow$ *case-sum g h* = *f*
⟨proof⟩

lemma *case-sum-expand-Inr*: *f o Inl* = *g* $\Longrightarrow$ *f x* = *case-sum g* (*f o Inr*) *x*
⟨proof⟩

lemma *id-transfer*: *rel-fun A A id id*
⟨proof⟩

lemma *fst-transfer*: *rel-fun* (*rel-prod A B*) *A fst fst*
⟨proof⟩

lemma *snd-transfer*: *rel-fun* (*rel-prod A B*) *B snd snd*

⟨*proof*⟩

lemma *convol-transfer*:
rel-fun (*rel-fun R S*) (*rel-fun* (*rel-fun R T*) (*rel-fun R* (*rel-prod S T*))) *BNF-Def.convol*
BNF-Def.convol
⟨*proof*⟩

lemma *Let-const*: *Let x* (λ-. *c*) = *c*
⟨*proof*⟩

⟨*ML*⟩

end

36 Least Fixpoint (Datatype) Operation on Bounded Natural Functors

theory *BNF-Least-Fixpoint*
imports *BNF-Fixpoint-Base*
keywords
datatype :: *thy-decl* **and**
datatype-compat :: *thy-decl*
begin

lemma *subset-emptyI*: ($\bigwedge x.\ x \in A \Longrightarrow False$) $\Longrightarrow A \subseteq \{\}$
⟨*proof*⟩

lemma *image-Collect-subsetI*: ($\bigwedge x.\ P\ x \Longrightarrow f\ x \in B$) $\Longrightarrow f$ ‘ $\{x.\ P\ x\} \subseteq B$
⟨*proof*⟩

lemma *Collect-restrict*: $\{x.\ x \in X \wedge P\ x\} \subseteq X$
⟨*proof*⟩

lemma *prop-restrict*: $[\![x \in Z;\ Z \subseteq \{x.\ x \in X \wedge P\ x\}]\!] \Longrightarrow P\ x$
⟨*proof*⟩

lemma *underS-I*: $[\![i \neq j;\ (i, j) \in R]\!] \Longrightarrow i \in$ *underS R j*
⟨*proof*⟩

lemma *underS-E*: $i \in$ *underS R j* $\Longrightarrow i \neq j \wedge (i, j) \in R$
⟨*proof*⟩

lemma *underS-Field*: $i \in$ *underS R j* $\Longrightarrow i \in$ *Field R*
⟨*proof*⟩

lemma *bij-betwE*: *bij-betw f A B* $\Longrightarrow \forall\, a{\in}A.\ f\ a \in B$
⟨*proof*⟩

lemma *f-the-inv-into-f-bij-betw*:
$bij\text{-}betw\ f\ A\ B \Longrightarrow (bij\text{-}betw\ f\ A\ B \Longrightarrow x \in B) \Longrightarrow f\ (the\text{-}inv\text{-}into\ A\ f\ x) = x$
⟨*proof*⟩

lemma *ex-bij-betw*: $|A| \leq o\ (r :: 'b\ rel) \Longrightarrow \exists f\ B :: 'b\ set.\ bij\text{-}betw\ f\ B\ A$
⟨*proof*⟩

lemma *bij-betwI'*:
$[\![\bigwedge x\ y.\ [\![x \in X;\ y \in X]\!] \Longrightarrow (f\ x = f\ y) = (x = y);$
$\bigwedge x.\ x \in X \Longrightarrow f\ x \in Y;$
$\bigwedge y.\ y \in Y \Longrightarrow \exists x \in X.\ y = f\ x]\!] \Longrightarrow bij\text{-}betw\ f\ X\ Y$
⟨*proof*⟩

lemma *surj-fun-eq*:
assumes *surj-on*: $f\ `\ X = UNIV$ **and** *eq-on*: $\forall x \in X.\ (g1\ o\ f)\ x = (g2\ o\ f)\ x$
shows $g1 = g2$
⟨*proof*⟩

lemma *Card-order-wo-rel*: $Card\text{-}order\ r \Longrightarrow wo\text{-}rel\ r$
⟨*proof*⟩

lemma *Cinfinite-limit*: $[\![x \in Field\ r;\ Cinfinite\ r]\!] \Longrightarrow \exists y \in Field\ r.\ x \neq y \wedge (x, y) \in r$
⟨*proof*⟩

lemma *Card-order-trans*:
$[\![Card\text{-}order\ r;\ x \neq y;\ (x, y) \in r;\ y \neq z;\ (y, z) \in r]\!] \Longrightarrow x \neq z \wedge (x, z) \in r$
⟨*proof*⟩

lemma *Cinfinite-limit2*:
assumes *x1*: $x1 \in Field\ r$ **and** *x2*: $x2 \in Field\ r$ **and** *r*: $Cinfinite\ r$
shows $\exists y \in Field\ r.\ (x1 \neq y \wedge (x1, y) \in r) \wedge (x2 \neq y \wedge (x2, y) \in r)$
⟨*proof*⟩

lemma *Cinfinite-limit-finite*:
$[\![finite\ X;\ X \subseteq Field\ r;\ Cinfinite\ r]\!] \Longrightarrow \exists y \in Field\ r.\ \forall x \in X.\ (x \neq y \wedge (x, y) \in r)$
⟨*proof*⟩

lemma *insert-subsetI*: $[\![x \in A;\ X \subseteq A]\!] \Longrightarrow insert\ x\ X \subseteq A$
⟨*proof*⟩

lemmas *well-order-induct-imp* = *wo-rel.well-order-induct*[*of r* $\lambda x.\ x \in Field\ r \longrightarrow P\ x$ **for** *r P*]

lemma *meta-spec2*:
assumes $(\bigwedge x\ y.\ PROP\ P\ x\ y)$
shows $PROP\ P\ x\ y$
⟨*proof*⟩

lemma *nchotomy-relcomppE*:
assumes $\bigwedge y. \exists x. y = f\ x\ (r\ OO\ s)\ a\ c\ \bigwedge b.\ r\ a\ (f\ b) \Longrightarrow s\ (f\ b)\ c \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *predicate2D-vimage2p*: $[\![R \leq vimage2p\ f\ g\ S;\ R\ x\ y]\!] \Longrightarrow S\ (f\ x)\ (g\ y)$
⟨*proof*⟩

lemma *ssubst-Pair-rhs*: $[\![(r,\ s) \in R;\ s' = s]\!] \Longrightarrow (r,\ s') \in R$
⟨*proof*⟩

lemma *all-mem-range1*:
$(\bigwedge y.\ y \in range\ f \Longrightarrow P\ y) \equiv (\bigwedge x.\ P\ (f\ x))$
⟨*proof*⟩

lemma *all-mem-range2*:
$(\bigwedge fa\ y.\ fa \in range\ f \Longrightarrow y \in range\ fa \Longrightarrow P\ y) \equiv (\bigwedge x\ xa.\ P\ (f\ x\ xa))$
⟨*proof*⟩

lemma *all-mem-range3*:
$(\bigwedge fa\ fb\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow y \in range\ fb \Longrightarrow P\ y) \equiv (\bigwedge x\ xa\ xb.\ P\ (f\ x\ xa\ xb))$
⟨*proof*⟩

lemma *all-mem-range4*:
$(\bigwedge fa\ fb\ fc\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow y \in range\ fc \Longrightarrow P\ y) \equiv$
$(\bigwedge x\ xa\ xb\ xc.\ P\ (f\ x\ xa\ xb\ xc))$
⟨*proof*⟩

lemma *all-mem-range5*:
$(\bigwedge fa\ fb\ fc\ fd\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow fd \in range\ fc \Longrightarrow$
$y \in range\ fd \Longrightarrow P\ y) \equiv$
$(\bigwedge x\ xa\ xb\ xc\ xd.\ P\ (f\ x\ xa\ xb\ xc\ xd))$
⟨*proof*⟩

lemma *all-mem-range6*:
$(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow fd \in range\ fc \Longrightarrow$
$fe \in range\ fd \Longrightarrow ff \in range\ fe \Longrightarrow y \in range\ ff \Longrightarrow P\ y) \equiv$
$(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf.\ P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf))$
⟨*proof*⟩

lemma *all-mem-range7*:
$(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow fd \in range\ fc \Longrightarrow$
$fe \in range\ fd \Longrightarrow ff \in range\ fe \Longrightarrow fg \in range\ ff \Longrightarrow y \in range\ fg \Longrightarrow P\ y)$

$\equiv$
  $(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg.\ P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg))$
  ⟨proof⟩

lemma *all-mem-range8*:
  $(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ fh\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow$
$fd \in range\ fc \Longrightarrow$
    $fe \in range\ fd \Longrightarrow ff \in range\ fe \Longrightarrow fg \in range\ ff \Longrightarrow fh \in range\ fg \Longrightarrow y \in$
$range\ fh \Longrightarrow P\ y) \equiv$
  $(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh.\ P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh))$
  ⟨proof⟩

lemmas *all-mem-range* = *all-mem-range1 all-mem-range2 all-mem-range3 all-mem-range4*
all-mem-range5
  all-mem-range6 all-mem-range7 all-mem-range8

lemma *pred-fun-True-id*: $NO\text{-}MATCH\ id\ p \Longrightarrow pred\text{-}fun\ (\lambda x.\ True)\ p\ f = pred\text{-}fun$
$(\lambda x.\ True)\ id\ (p \circ f)$
  ⟨proof⟩

⟨ML⟩

end

theory *Basic-BNF-LFPs*
imports *BNF-Least-Fixpoint*
begin

definition $xtor :: 'a \Rightarrow 'a$ **where**
  $xtor\ x = x$

lemma *xtor-map*: $f\ (xtor\ x) = xtor\ (f\ x)$
  ⟨proof⟩

lemma *xtor-map-unique*: $u \circ xtor = xtor \circ f \Longrightarrow u = f$
  ⟨proof⟩

lemma *xtor-set*: $f\ (xtor\ x) = f\ x$
  ⟨proof⟩

lemma *xtor-rel*: $R\ (xtor\ x)\ (xtor\ y) = R\ x\ y$
  ⟨proof⟩

lemma *xtor-induct*: $(\bigwedge x.\ P\ (xtor\ x)) \Longrightarrow P\ z$
  ⟨proof⟩

lemma *xtor-xtor*: $xtor\ (xtor\ x) = x$
  ⟨proof⟩

lemmas *xtor-inject* = *xtor-rel*[*of op* =]

lemma *xtor-rel-induct*: ($\bigwedge x\ y.\ vimage2p\ id\text{-}bnf\ id\text{-}bnf\ R\ x\ y \Longrightarrow IR\ (xtor\ x)\ (xtor\ y)) \Longrightarrow R \leq IR$
⟨*proof*⟩

lemma *xtor-iff-xtor*: $u = xtor\ w \longleftrightarrow xtor\ u = w$
⟨*proof*⟩

lemma *Inl-def-alt*: $Inl \equiv (\lambda a.\ xtor\ (id\text{-}bnf\ (Inl\ a)))$
⟨*proof*⟩

lemma *Inr-def-alt*: $Inr \equiv (\lambda a.\ xtor\ (id\text{-}bnf\ (Inr\ a)))$
⟨*proof*⟩

lemma *Pair-def-alt*: $Pair \equiv (\lambda a\ b.\ xtor\ (id\text{-}bnf\ (a,\ b)))$
⟨*proof*⟩

definition *ctor-rec* :: $'a \Rightarrow 'a$ **where**
$ctor\text{-}rec\ x = x$

lemma *ctor-rec*: $g = id \Longrightarrow ctor\text{-}rec\ f\ (xtor\ x) = f\ ((id\text{-}bnf \circ g \circ id\text{-}bnf)\ x)$
⟨*proof*⟩

lemma *ctor-rec-unique*: $g = id \Longrightarrow f \circ xtor = s \circ (id\text{-}bnf \circ g \circ id\text{-}bnf) \Longrightarrow f = ctor\text{-}rec\ s$
⟨*proof*⟩

lemma *ctor-rec-def-alt*: $f = ctor\text{-}rec\ (f \circ id\text{-}bnf)$
⟨*proof*⟩

lemma *ctor-rec-o-map*: $ctor\text{-}rec\ f \circ g = ctor\text{-}rec\ (f \circ (id\text{-}bnf \circ g \circ id\text{-}bnf))$
⟨*proof*⟩

lemma *ctor-rec-transfer*: $rel\text{-}fun\ (rel\text{-}fun\ (vimage2p\ id\text{-}bnf\ id\text{-}bnf\ R)\ S)\ (rel\text{-}fun\ R\ S)\ ctor\text{-}rec\ ctor\text{-}rec$
⟨*proof*⟩

lemma *eq-fst-iff*: $a = fst\ p \longleftrightarrow (\exists\, b.\ p = (a,\ b))$
⟨*proof*⟩

lemma *eq-snd-iff*: $b = snd\ p \longleftrightarrow (\exists\, a.\ p = (a,\ b))$
⟨*proof*⟩

lemma *ex-neg-all-pos*: $((\exists\, x.\ P\ x) \Longrightarrow Q) \equiv (\bigwedge x.\ P\ x \Longrightarrow Q)$
⟨*proof*⟩

lemma *hypsubst-in-prems*: $(\bigwedge x.\ y = x \Longrightarrow z = f\ x \Longrightarrow P) \equiv (z = f\ y \Longrightarrow P)$

⟨*proof*⟩

lemma *isl-map-sum*:
isl (*map-sum f g s*) = *isl s*
⟨*proof*⟩

lemma *map-sum-sel*:
isl s $\Longrightarrow$ *projl* (*map-sum f g s*) = *f* (*projl s*)
$\neg$ *isl s* $\Longrightarrow$ *projr* (*map-sum f g s*) = *g* (*projr s*)
⟨*proof*⟩

lemma *set-sum-sel*:
isl s $\Longrightarrow$ *projl s* $\in$ *setl s*
$\neg$ *isl s* $\Longrightarrow$ *projr s* $\in$ *setr s*
⟨*proof*⟩

lemma *rel-sum-sel*: *rel-sum R1 R2 a b* = (*isl a* = *isl b* $\wedge$
(*isl a* $\longrightarrow$ *isl b* $\longrightarrow$ *R1* (*projl a*) (*projl b*)) $\wedge$
($\neg$ *isl a* $\longrightarrow$ $\neg$ *isl b* $\longrightarrow$ *R2* (*projr a*) (*projr b*)))
⟨*proof*⟩

lemma *isl-transfer*: *rel-fun* (*rel-sum A B*) (*op* =) *isl isl*
⟨*proof*⟩

lemma *rel-prod-sel*: *rel-prod R1 R2 p q* = (*R1* (*fst p*) (*fst q*) $\wedge$ *R2* (*snd p*) (*snd q*))
⟨*proof*⟩

⟨*ML*⟩

lemma *size-bool*[*code*]: *size* (*b* :: *bool*) = *0*
⟨*proof*⟩

declare *prod.size*[*no-atp*]

lemmas *size-nat* = *size-nat-def*

hide-const (**open**) *xtor ctor-rec*

hide-fact (**open**)
xtor-def xtor-map xtor-set xtor-rel xtor-induct xtor-xtor xtor-inject ctor-rec-def ctor-rec
ctor-rec-def-alt ctor-rec-o-map xtor-rel-induct Inl-def-alt Inr-def-alt Pair-def-alt

end

37 MESON Proof Method

theory *Meson*

imports *Nat*
begin

37.1 Negation Normal Form

de Morgan laws

lemma *not-conjD*: $\sim(P\&Q) ==> \sim P \mid \sim Q$
and *not-disjD*: $\sim(P|Q) ==> \sim P \;\&\; \sim Q$
and *not-notD*: $\sim\sim P ==> P$
and *not-allD*: $!!P.\ \sim(\forall\, x.\ P(x)) ==> \exists\, x.\ \sim P(x)$
and *not-exD*: $!!P.\ \sim(\exists\, x.\ P(x)) ==> \forall\, x.\ \sim P(x)$
⟨*proof*⟩

Removal of $\longrightarrow$ and $\longleftrightarrow$ (positive and negative occurrences)

lemma *imp-to-disjD*: $P{-}{-}{>}Q ==> \sim P \mid Q$
and *not-impD*: $\sim(P{-}{-}{>}Q) ==> P \;\&\; \sim Q$
and *iff-to-disjD*: $P{=}Q ==> (\sim P \mid Q) \;\&\; (\sim Q \mid P)$
and *not-iffD*: $\sim(P{=}Q) ==> (P \mid Q) \;\&\; (\sim P \mid \sim Q)$
— Much more efficient than $P \wedge \neg\ Q \vee Q \wedge \neg\ P$ for computing CNF
and *not-refl-disj-D*: $x \sim= x \mid P ==> P$
⟨*proof*⟩

37.2 Pulling out the existential quantifiers

Conjunction

lemma *conj-exD1*: $!!P\ Q.\ (\exists\, x.\ P(x)) \;\&\; Q ==> \exists\, x.\ P(x) \;\&\; Q$
and *conj-exD2*: $!!P\ Q.\ P \;\&\; (\exists\, x.\ Q(x)) ==> \exists\, x.\ P \;\&\; Q(x)$
⟨*proof*⟩

Disjunction

lemma *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 *disj-exD1*: $!!P\ Q.\ (\exists\, x.\ P(x)) \mid Q ==> \exists\, x.\ P(x) \mid Q$
and *disj-exD2*: $!!P\ Q.\ P \mid (\exists\, x.\ Q(x)) ==> \exists\, x.\ P \mid Q(x)$
⟨*proof*⟩

lemma *disj-assoc*: $(P|Q)|R ==> P|(Q|R)$
and *disj-comm*: $P|Q ==> Q|P$
and *disj-FalseD1*: $\mathit{False}|P ==> P$
and *disj-FalseD2*: $P|\mathit{False} ==> P$
⟨*proof*⟩

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*⟩

37.3 Lemmas for Forward Proof

There is a similarity to congruence rules. They are also useful in ordinary proofs.

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 *imp-forward*: $[|\ P' \longrightarrow Q';\ \ P ==> P';\ \ Q' ==> Q\ |] ==> P \longrightarrow Q$
⟨*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*⟩

37.4 Clausification helper

lemma *TruepropI*: $P \equiv Q \Longrightarrow Trueprop\ P \equiv Trueprop\ Q$
⟨*proof*⟩

lemma *ext-cong-neq*: $F\ g \neq F\ h \Longrightarrow F\ g \neq F\ h \wedge (\exists\, x.\ g\ x \neq h\ x)$
⟨*proof*⟩

Combinator translation helpers

definition *COMBI* :: $'a \Rightarrow 'a$ **where**
$COMBI\ P = P$

definition *COMBK* :: $'a \Rightarrow 'b \Rightarrow 'a$ **where**
$COMBK\ P\ Q = P$

definition *COMBB* :: $('b => 'c) \Rightarrow ('a => 'b) \Rightarrow 'a \Rightarrow 'c$ **where**
$COMBB\ P\ Q\ R = P\ (Q\ R)$

definition *COMBC* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
$COMBC\ P\ Q\ R = P\ R\ Q$

definition *COMBS* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ **where**
$COMBS\ P\ Q\ R = P\ R\ (Q\ R)$

lemma *abs-S*: $\lambda x.\ (f\ x)\ (g\ x) \equiv COMBS\ f\ g$
⟨*proof*⟩

lemma *abs-I*: $\lambda x.\ x \equiv COMBI$
⟨*proof*⟩

lemma *abs-K*: $\lambda x.\ y \equiv COMBK\ y$
⟨*proof*⟩

lemma *abs-B*: $\lambda x.\ a\ (g\ x) \equiv COMBB\ a\ g$
⟨*proof*⟩

lemma *abs-C*: $\lambda x.\ (f\ x)\ b \equiv COMBC\ f\ b$
⟨*proof*⟩

37.5 Skolemization helpers

definition *skolem* :: $'a \Rightarrow 'a$ **where**
$skolem = (\lambda x.\ x)$

lemma *skolem-COMBK-iff*: $P \longleftrightarrow skolem\ (COMBK\ P\ (i::nat))$
⟨*proof*⟩

lemmas *skolem-COMBK-I* = *iffD1* [*OF skolem-COMBK-iff*]
lemmas *skolem-COMBK-D* = *iffD2* [*OF skolem-COMBK-iff*]

37.6 Meson package

⟨*ML*⟩

hide-const (**open**) *COMBI COMBK COMBB COMBC COMBS skolem*
hide-fact (**open**) *not-conjD not-disjD not-notD not-allD not-exD imp-to-disjD not-impD iff-to-disjD not-iffD not-refl-disj-D conj-exD1 conj-exD2 disj-exD disj-exD1 disj-exD2 disj-assoc disj-comm disj-FalseD1 disj-FalseD2 TruepropI ext-cong-neq COMBI-def COMBK-def COMBB-def COMBC-def COMBS-def abs-I abs-K abs-B abs-C abs-S skolem-def skolem-COMBK-iff skolem-COMBK-I skolem-COMBK-D*

end

38 Automatic Theorem Provers (ATPs)

theory *ATP*
imports *Meson*
begin

38.1 ATP problems and proofs

⟨*ML*⟩

38.2 Higher-order reasoning helpers

definition *fFalse* :: *bool* **where**
$fFalse \longleftrightarrow False$

definition *fTrue* :: *bool* **where**
$fTrue \longleftrightarrow True$

definition *fNot* :: $bool \Rightarrow bool$ **where**
$fNot\ P \longleftrightarrow \neg\ P$

definition *fComp* :: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
$fComp\ P = (\lambda x.\ \neg\ P\ x)$

definition *fconj* :: $bool \Rightarrow bool \Rightarrow bool$ **where**
$fconj\ P\ Q \longleftrightarrow P \wedge Q$

definition *fdisj* :: $bool \Rightarrow bool \Rightarrow bool$ **where**
$fdisj\ P\ Q \longleftrightarrow P \vee Q$

definition *fimplies* :: $bool \Rightarrow bool \Rightarrow bool$ **where**
$fimplies\ P\ Q \longleftrightarrow (P \longrightarrow Q)$

definition *fAll* :: $('a \Rightarrow bool) \Rightarrow bool$ **where**
$fAll\ P \longleftrightarrow All\ P$

definition fEx :: $('a \Rightarrow bool) \Rightarrow bool$ **where**
$fEx\ P \longleftrightarrow Ex\ P$

definition $fequal$ :: $'a \Rightarrow 'a \Rightarrow bool$ **where**
$fequal\ x\ y \longleftrightarrow (x = y)$

lemma $fTrue\text{-}ne\text{-}fFalse$: $fFalse \neq fTrue$
$\langle proof \rangle$

lemma $fNot\text{-}table$:
$fNot\ fFalse = fTrue$
$fNot\ fTrue = fFalse$
$\langle proof \rangle$

lemma $fconj\text{-}table$:
$fconj\ fFalse\ P = fFalse$
$fconj\ P\ fFalse = fFalse$
$fconj\ fTrue\ fTrue = fTrue$
$\langle proof \rangle$

lemma $fdisj\text{-}table$:
$fdisj\ fTrue\ P = fTrue$
$fdisj\ P\ fTrue = fTrue$
$fdisj\ fFalse\ fFalse = fFalse$
$\langle proof \rangle$

lemma $fimplies\text{-}table$:
$fimplies\ P\ fTrue = fTrue$
$fimplies\ fFalse\ P = fTrue$
$fimplies\ fTrue\ fFalse = fFalse$
$\langle proof \rangle$

lemma $fAll\text{-}table$:
$Ex\ (fComp\ P) \lor fAll\ P = fTrue$
$All\ P \lor fAll\ P = fFalse$
$\langle proof \rangle$

lemma $fEx\text{-}table$:
$All\ (fComp\ P) \lor fEx\ P = fTrue$
$Ex\ P \lor fEx\ P = fFalse$
$\langle proof \rangle$

lemma $fequal\text{-}table$:
$fequal\ x\ x = fTrue$
$x = y \lor fequal\ x\ y = fFalse$
$\langle proof \rangle$

lemma $fNot\text{-}law$:
$fNot\ P \neq P$

⟨*proof*⟩

lemma *fComp-law*:
$fComp\ P\ x \longleftrightarrow \neg\ P\ x$
⟨*proof*⟩

lemma *fconj-laws*:
$fconj\ P\ P \longleftrightarrow P$
$fconj\ P\ Q \longleftrightarrow fconj\ Q\ P$
$fNot\ (fconj\ P\ Q) \longleftrightarrow fdisj\ (fNot\ P)\ (fNot\ Q)$
⟨*proof*⟩

lemma *fdisj-laws*:
$fdisj\ P\ P \longleftrightarrow P$
$fdisj\ P\ Q \longleftrightarrow fdisj\ Q\ P$
$fNot\ (fdisj\ P\ Q) \longleftrightarrow fconj\ (fNot\ P)\ (fNot\ Q)$
⟨*proof*⟩

lemma *fimplies-laws*:
$fimplies\ P\ Q \longleftrightarrow fdisj\ (\neg\ P)\ Q$
$fNot\ (fimplies\ P\ Q) \longleftrightarrow fconj\ P\ (fNot\ Q)$
⟨*proof*⟩

lemma *fAll-law*:
$fNot\ (fAll\ R) \longleftrightarrow fEx\ (fComp\ R)$
⟨*proof*⟩

lemma *fEx-law*:
$fNot\ (fEx\ R) \longleftrightarrow fAll\ (fComp\ R)$
⟨*proof*⟩

lemma *fequal-laws*:
$fequal\ x\ y = fequal\ y\ x$
$fequal\ x\ y = fFalse \lor fequal\ y\ z = fFalse \lor fequal\ x\ z = fTrue$
$fequal\ x\ y = fFalse \lor fequal\ (f\ x)\ (f\ y) = fTrue$
⟨*proof*⟩

38.3 Waldmeister helpers

lemma *boolean-equality*: $(P \longleftrightarrow P) = True$
⟨*proof*⟩

lemma *boolean-comm*: $(P \longleftrightarrow Q) = (Q \longleftrightarrow P)$
⟨*proof*⟩

lemmas *waldmeister-fol* = *boolean-equality boolean-comm*
simp-thms(*1*−*5*,*7*−*8*,*11*−*25*,*27*−*33*) *disj-comms disj-assoc conj-comms conj-assoc*

38.4 Basic connection between ATPs and HOL

⟨*ML*⟩

hide-fact (**open**) *waldmeister-fol boolean-equality boolean-comm*

end

39 Metis Proof Method

theory *Metis*
imports *ATP*
begin

⟨*ML*⟩

39.1 Literal selection and lambda-lifting helpers

definition $select :: 'a \Rightarrow 'a$ **where**
$select = (\lambda x.\ x)$

lemma *not-atomize*: $(\neg\ A \Longrightarrow False) \equiv Trueprop\ A$
⟨*proof*⟩

lemma *atomize-not-select*: $(A \Longrightarrow select\ False) \equiv Trueprop\ (\neg\ A)$
⟨*proof*⟩

lemma *not-atomize-select*: $(\neg\ A \Longrightarrow select\ False) \equiv Trueprop\ A$
⟨*proof*⟩

lemma *select-FalseI*: $False \Longrightarrow select\ False$ ⟨*proof*⟩

definition $lambda :: 'a \Rightarrow 'a$ **where**
$lambda = (\lambda x.\ x)$

lemma *eq-lambdaI*: $x \equiv y \Longrightarrow x \equiv lambda\ y$
⟨*proof*⟩

39.2 Metis package

⟨*ML*⟩

hide-const (**open**) *select fFalse fTrue fNot fComp fconj fdisj fimplies fAll fEx fequal lambda*
hide-fact (**open**) *select-def not-atomize atomize-not-select not-atomize-select select-FalseI fFalse-def fTrue-def fNot-def fconj-def fdisj-def fimplies-def fAll-def fEx-def fequal-def fTrue-ne-fFalse fNot-table fconj-table fdisj-table fimplies-table fAll-table fEx-table fequal-table fAll-table fEx-table fNot-law fComp-law fconj-laws fdisj-laws fimplies-laws fequal-laws fAll-law fEx-law lambda-def eq-lambdaI*

end

40 Generic theorem transfer using relations

theory *Transfer*
imports *Basic-BNF-LFPs Hilbert-Choice Metis*
begin

40.1 Relator for function space

bundle *lifting-syntax*
begin
  notation *rel-fun* (**infixr** ===> *55*)
  notation *map-fun* (**infixr** −−−> *55*)
end

context includes *lifting-syntax*
begin

lemma *rel-funD2*:
  assumes *rel-fun A B f g* **and** *A x x*
  shows $B\ (f\ x)\ (g\ x)$
  ⟨*proof*⟩

lemma *rel-funE*:
  assumes *rel-fun A B f g* **and** *A x y*
  obtains $B\ (f\ x)\ (g\ y)$
  ⟨*proof*⟩

lemmas *rel-fun-eq* = *fun.rel-eq*

lemma *rel-fun-eq-rel*:
shows *rel-fun* (*op* =) $R = (\lambda f\ g.\ \forall x.\ R\ (f\ x)\ (g\ x))$
  ⟨*proof*⟩

40.2 Transfer method

Explicit tag for relation membership allows for backward proof methods.

definition *Rel* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
  where $Rel\ r \equiv r$

Handling of equality relations

definition *is-equality* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  where *is-equality* $R \longleftrightarrow R = (op\ =)$

lemma *is-equality-eq*: *is-equality* (*op* =)
  ⟨*proof*⟩

Reverse implication for monotonicity rules

definition *rev-implies* **where**
$\textit{rev-implies}\ x\ y \longleftrightarrow (y \longrightarrow x)$

Handling of meta-logic connectives

definition *transfer-forall* **where**
$\textit{transfer-forall} \equiv \textit{All}$

definition *transfer-implies* **where**
$\textit{transfer-implies} \equiv \textit{op} \longrightarrow$

definition *transfer-bforall* :: $('a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow \textit{bool}) \Rightarrow \textit{bool}$
where $\textit{transfer-bforall} \equiv (\lambda P\ Q.\ \forall x.\ P\ x \longrightarrow Q\ x)$

lemma *transfer-forall-eq*: $(\bigwedge x.\ P\ x) \equiv \textit{Trueprop}\ (\textit{transfer-forall}\ (\lambda x.\ P\ x))$
⟨*proof*⟩

lemma *transfer-implies-eq*: $(A \Longrightarrow B) \equiv \textit{Trueprop}\ (\textit{transfer-implies}\ A\ B)$
⟨*proof*⟩

lemma *transfer-bforall-unfold*:
$\textit{Trueprop}\ (\textit{transfer-bforall}\ P\ (\lambda x.\ Q\ x)) \equiv (\bigwedge x.\ P\ x \Longrightarrow Q\ x)$
⟨*proof*⟩

lemma *transfer-start*: $[\![P;\ \textit{Rel}\ (\textit{op} =)\ P\ Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *transfer-start'*: $[\![P;\ \textit{Rel}\ (\textit{op} \longrightarrow)\ P\ Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *transfer-prover-start*: $[\![x = x';\ \textit{Rel}\ R\ x'\ y]\!] \Longrightarrow \textit{Rel}\ R\ x\ y$
⟨*proof*⟩

lemma *untransfer-start*: $[\![Q;\ \textit{Rel}\ (\textit{op} =)\ P\ Q]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *Rel-eq-refl*: $\textit{Rel}\ (\textit{op} =)\ x\ x$
⟨*proof*⟩

lemma *Rel-app*:
assumes $\textit{Rel}\ (A\ {=}{=}{=}{>}\ B)\ f\ g$ **and** $\textit{Rel}\ A\ x\ y$
shows $\textit{Rel}\ B\ (f\ x)\ (g\ y)$
⟨*proof*⟩

lemma *Rel-abs*:
assumes $\bigwedge x\ y.\ \textit{Rel}\ A\ x\ y \Longrightarrow \textit{Rel}\ B\ (f\ x)\ (g\ y)$
shows $\textit{Rel}\ (A\ {=}{=}{=}{>}\ B)\ (\lambda x.\ f\ x)\ (\lambda y.\ g\ y)$
⟨*proof*⟩

40.3 Predicates on relations, i.e. "class constraints"

definition *left-total* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where *left-total* $R \longleftrightarrow (\forall x.\ \exists y.\ R\ x\ y)$

definition *left-unique* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where *left-unique* $R \longleftrightarrow (\forall x\ y\ z.\ R\ x\ z \longrightarrow R\ y\ z \longrightarrow x = y)$

definition *right-total* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where *right-total* $R \longleftrightarrow (\forall y.\ \exists x.\ R\ x\ y)$

definition *right-unique* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where *right-unique* $R \longleftrightarrow (\forall x\ y\ z.\ R\ x\ y \longrightarrow R\ x\ z \longrightarrow y = z)$

definition *bi-total* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where *bi-total* $R \longleftrightarrow (\forall x.\ \exists y.\ R\ x\ y) \wedge (\forall y.\ \exists x.\ R\ x\ y)$

definition *bi-unique* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where *bi-unique* $R \longleftrightarrow$
$(\forall x\ y\ z.\ R\ x\ y \longrightarrow R\ x\ z \longrightarrow y = z) \wedge$
$(\forall x\ y\ z.\ R\ x\ z \longrightarrow R\ y\ z \longrightarrow x = y)$

lemma *left-uniqueI*: $(\bigwedge x\ y\ z.\ [\![A\ x\ z;\ A\ y\ z]\!] \Longrightarrow x = y) \Longrightarrow$ *left-unique* A
⟨*proof*⟩

lemma *left-uniqueD*: $[\![$ *left-unique* $A;\ A\ x\ z;\ A\ y\ z]\!] \Longrightarrow x = y$
⟨*proof*⟩

lemma *left-totalI*:
$(\bigwedge x.\ \exists y.\ R\ x\ y) \Longrightarrow$ *left-total* R
⟨*proof*⟩

lemma *left-totalE*:
assumes *left-total* R
obtains $(\bigwedge x.\ \exists y.\ R\ x\ y)$
⟨*proof*⟩

lemma *bi-uniqueDr*: $[\![$ *bi-unique* $A;\ A\ x\ y;\ A\ x\ z]\!] \Longrightarrow y = z$
⟨*proof*⟩

lemma *bi-uniqueDl*: $[\![$ *bi-unique* $A;\ A\ x\ y;\ A\ z\ y]\!] \Longrightarrow x = z$
⟨*proof*⟩

lemma *right-uniqueI*: $(\bigwedge x\ y\ z.\ [\![A\ x\ y;\ A\ x\ z]\!] \Longrightarrow y = z) \Longrightarrow$ *right-unique* A
⟨*proof*⟩

lemma *right-uniqueD*: $[\![$ *right-unique* $A;\ A\ x\ y;\ A\ x\ z]\!] \Longrightarrow y = z$
⟨*proof*⟩

lemma *right-totalI*: $(\bigwedge y.\ \exists x.\ A\ x\ y) \Longrightarrow$ *right-total* A

⟨*proof*⟩

lemma *right-totalE*:
  assumes *right-total A*
  obtains *x* **where** *A x y*
⟨*proof*⟩

lemma *right-total-alt-def2*:
  $right\text{-}total\ R \longleftrightarrow ((R\ {=}{=}{=}{>}\ op \longrightarrow)\ {=}{=}{=}{>}\ op \longrightarrow)\ All\ All$
  ⟨*proof*⟩

lemma *right-unique-alt-def2*:
  $right\text{-}unique\ R \longleftrightarrow (R\ {=}{=}{=}{>}\ R\ {=}{=}{=}{>}\ op \longrightarrow)\ (op =)\ (op =)$
  ⟨*proof*⟩

lemma *bi-total-alt-def2*:
  $bi\text{-}total\ R \longleftrightarrow ((R\ {=}{=}{=}{>}\ op =)\ {=}{=}{=}{>}\ op =)\ All\ All$
  ⟨*proof*⟩

lemma *bi-unique-alt-def2*:
  $bi\text{-}unique\ R \longleftrightarrow (R\ {=}{=}{=}{>}\ R\ {=}{=}{=}{>}\ op =)\ (op =)\ (op =)$
  ⟨*proof*⟩

lemma [*simp*]:
  shows *left-unique-conversep*: $left\text{-}unique\ A^{-1-1} \longleftrightarrow right\text{-}unique\ A$
  and *right-unique-conversep*: $right\text{-}unique\ A^{-1-1} \longleftrightarrow left\text{-}unique\ A$
⟨*proof*⟩

lemma [*simp*]:
  shows *left-total-conversep*: $left\text{-}total\ A^{-1-1} \longleftrightarrow right\text{-}total\ A$
  and *right-total-conversep*: $right\text{-}total\ A^{-1-1} \longleftrightarrow left\text{-}total\ A$
⟨*proof*⟩

lemma *bi-unique-conversep* [*simp*]: $bi\text{-}unique\ R^{-1-1} = bi\text{-}unique\ R$
⟨*proof*⟩

lemma *bi-total-conversep* [*simp*]: $bi\text{-}total\ R^{-1-1} = bi\text{-}total\ R$
⟨*proof*⟩

lemma *right-unique-alt-def*: $right\text{-}unique\ R = (conversep\ R\ OO\ R \le op{=})$ ⟨*proof*⟩
lemma *left-unique-alt-def*: $left\text{-}unique\ R = (R\ OO\ (conversep\ R) \le op{=})$ ⟨*proof*⟩

lemma *right-total-alt-def*: $right\text{-}total\ R = (conversep\ R\ OO\ R \ge op{=})$ ⟨*proof*⟩
lemma *left-total-alt-def*: $left\text{-}total\ R = (R\ OO\ conversep\ R \ge op{=})$ ⟨*proof*⟩

lemma *bi-total-alt-def*: $bi\text{-}total\ A = (left\text{-}total\ A \land right\text{-}total\ A)$
⟨*proof*⟩

lemma *bi-unique-alt-def*: $bi\text{-}unique\ A = (left\text{-}unique\ A \land right\text{-}unique\ A)$

⟨*proof*⟩

lemma *bi-totalI*: *left-total R* $\Longrightarrow$ *right-total R* $\Longrightarrow$ *bi-total R*
⟨*proof*⟩

lemma *bi-uniqueI*: *left-unique R* $\Longrightarrow$ *right-unique R* $\Longrightarrow$ *bi-unique R*
⟨*proof*⟩

end

⟨*ML*⟩
declare *refl* [*transfer-rule*]

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

context includes *lifting-syntax*
begin

Handling of domains

lemma *Domainp-iff*: *Domainp T x* $\longleftrightarrow$ ($\exists$ *y*. *T x y*)
⟨*proof*⟩

lemma *Domainp-refl*[*transfer-domain-rule*]:
Domainp T = *Domainp T* ⟨*proof*⟩

lemma *Domain-eq-top*[*transfer-domain-rule*]: *Domainp op=* = *top* ⟨*proof*⟩

lemma *Domainp-pred-fun-eq*[*relator-domain*]:
assumes *left-unique T*
shows *Domainp* (*T* ===> *S*) = *pred-fun* (*Domainp T*) (*Domainp S*)
⟨*proof*⟩

Properties are preserved by relation composition.

lemma *OO-def*: *R OO S* = (λ*x z*. $\exists$ *y*. *R x y* $\wedge$ *S y z*)
⟨*proof*⟩

lemma *bi-total-OO*: ⟦*bi-total A*; *bi-total B*⟧ $\Longrightarrow$ *bi-total* (*A OO B*)
⟨*proof*⟩

lemma *bi-unique-OO*: ⟦*bi-unique A*; *bi-unique B*⟧ $\Longrightarrow$ *bi-unique* (*A OO B*)
⟨*proof*⟩

lemma *right-total-OO*:
⟦*right-total A*; *right-total B*⟧ $\Longrightarrow$ *right-total* (*A OO B*)
⟨*proof*⟩

lemma *right-unique-OO*:

$[\![$*right-unique* A; *right-unique* $B]\!] \Longrightarrow$ *right-unique* $(A\ OO\ B)$
⟨*proof*⟩

lemma *left-total-OO*: *left-total* $R \Longrightarrow$ *left-total* $S \Longrightarrow$ *left-total* $(R\ OO\ S)$
⟨*proof*⟩

lemma *left-unique-OO*: *left-unique* $R \Longrightarrow$ *left-unique* $S \Longrightarrow$ *left-unique* $(R\ OO\ S)$
⟨*proof*⟩

40.4 Properties of relators

lemma *left-total-eq*[*transfer-rule*]: *left-total op=*
⟨*proof*⟩

lemma *left-unique-eq*[*transfer-rule*]: *left-unique op=*
⟨*proof*⟩

lemma *right-total-eq* [*transfer-rule*]: *right-total op=*
⟨*proof*⟩

lemma *right-unique-eq* [*transfer-rule*]: *right-unique op=*
⟨*proof*⟩

lemma *bi-total-eq*[*transfer-rule*]: *bi-total* (*op* =)
⟨*proof*⟩

lemma *bi-unique-eq*[*transfer-rule*]: *bi-unique* (*op* =)
⟨*proof*⟩

lemma *left-total-fun*[*transfer-rule*]:
$[\![$*left-unique* A; *left-total* $B]\!] \Longrightarrow$ *left-total* $(A$ ===> $B)$
⟨*proof*⟩

lemma *left-unique-fun*[*transfer-rule*]:
$[\![$*left-total* A; *left-unique* $B]\!] \Longrightarrow$ *left-unique* $(A$ ===> $B)$
⟨*proof*⟩

lemma *right-total-fun* [*transfer-rule*]:
$[\![$*right-unique* A; *right-total* $B]\!] \Longrightarrow$ *right-total* $(A$ ===> $B)$
⟨*proof*⟩

lemma *right-unique-fun* [*transfer-rule*]:
$[\![$*right-total* A; *right-unique* $B]\!] \Longrightarrow$ *right-unique* $(A$ ===> $B)$
⟨*proof*⟩

lemma *bi-total-fun*[*transfer-rule*]:
$[\![$*bi-unique* A; *bi-total* $B]\!] \Longrightarrow$ *bi-total* $(A$ ===> $B)$
⟨*proof*⟩

lemma *bi-unique-fun*[*transfer-rule*]:
  $[\![$*bi-total A*; *bi-unique B*$]\!] \Longrightarrow$ *bi-unique* (*A* ===> *B*)
  ⟨*proof*⟩

end

lemma *if-conn*:
  (*if P* $\wedge$ *Q then t else e*) = (*if P then if Q then t else e else e*)
  (*if P* $\vee$ *Q then t else e*) = (*if P then t else if Q then t else e*)
  (*if P* $\longrightarrow$ *Q then t else e*) = (*if P then if Q then t else e else t*)
  (*if* $\neg$ *P then t else e*) = (*if P then e else t*)
⟨*proof*⟩

⟨*ML*⟩

declare *pred-fun-def* [*simp*]
declare *rel-fun-eq* [*relator-eq*]

declare *fun.Domainp-rel*[*relator-domain del*]

40.5 Transfer rules

context includes *lifting-syntax*
begin

lemma *Domainp-forall-transfer* [*transfer-rule*]:
  assumes *right-total A*
  shows ((*A* ===> *op* =) ===> *op* =)
    (*transfer-bforall* (*Domainp A*)) *transfer-forall*
  ⟨*proof*⟩

Transfer rules using implication instead of equality on booleans.

lemma *transfer-forall-transfer* [*transfer-rule*]:
  bi-total A $\Longrightarrow$ ((*A* ===> *op* =) ===> *op* =) *transfer-forall transfer-forall*
  right-total A $\Longrightarrow$ ((*A* ===> *op* =) ===> *implies*) *transfer-forall transfer-forall*
  right-total A $\Longrightarrow$ ((*A* ===> *implies*) ===> *implies*) *transfer-forall transfer-forall*
  bi-total A $\Longrightarrow$ ((*A* ===> *op* =) ===> *rev-implies*) *transfer-forall transfer-forall*
  bi-total A $\Longrightarrow$ ((*A* ===> *rev-implies*) ===> *rev-implies*) *transfer-forall transfer-forall*
  ⟨*proof*⟩

lemma *transfer-implies-transfer* [*transfer-rule*]:
  (*op* = ===> *op* = ===> *op* =) *transfer-implies transfer-implies*
  (*rev-implies* ===> *implies* ===> *implies*) *transfer-implies transfer-implies*
  (*rev-implies* ===> *op* = ===> *implies*) *transfer-implies transfer-implies*
  (*op* = ===> *implies* ===> *implies*) *transfer-implies transfer-implies*
  (*op* = ===> *op* = ===> *implies*) *transfer-implies transfer-implies*
  (*implies* ===> *rev-implies* ===> *rev-implies*) *transfer-implies transfer-implies*
  (*implies* ===> *op* = ===> *rev-implies*) *transfer-implies transfer-implies*

(*op =* ===> *rev-implies* ===> *rev-implies*) *transfer-implies transfer-implies*
(*op =* ===> *op =* ===> *rev-implies*) *transfer-implies transfer-implies*
⟨*proof*⟩

lemma *eq-imp-transfer* [*transfer-rule*]:
right-unique A ⟹ (*A* ===> *A* ===> *op* ⟶) (*op* =) (*op* =)
⟨*proof*⟩

Transfer rules using equality.

lemma *left-unique-transfer* [*transfer-rule*]:
assumes *right-total A*
assumes *right-total B*
assumes *bi-unique A*
shows ((*A* ===> *B* ===> *op*=) ===> *implies*) *left-unique left-unique*
⟨*proof*⟩

lemma *eq-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *A* ===> *op* =) (*op* =) (*op* =)
⟨*proof*⟩

lemma *right-total-Ex-transfer*[*transfer-rule*]:
assumes *right-total A*
shows ((*A* ===> *op*=) ===> *op*=) (*Bex* (*Collect* (*Domainp A*))) *Ex*
⟨*proof*⟩

lemma *right-total-All-transfer*[*transfer-rule*]:
assumes *right-total A*
shows ((*A* ===> *op* =) ===> *op* =) (*Ball* (*Collect* (*Domainp A*))) *All*
⟨*proof*⟩

lemma *All-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows ((*A* ===> *op* =) ===> *op* =) *All All*
⟨*proof*⟩

lemma *Ex-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows ((*A* ===> *op* =) ===> *op* =) *Ex Ex*
⟨*proof*⟩

lemma *Ex1-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A bi-total A*
shows ((*A* ===> *op* =) ===> *op* =) *Ex1 Ex1*
⟨*proof*⟩

declare *If-transfer* [*transfer-rule*]

lemma *Let-transfer* [*transfer-rule*]: (*A* ===> (*A* ===> *B*) ===> *B*) *Let Let*

⟨*proof*⟩

declare *id-transfer* [*transfer-rule*]

declare *comp-transfer* [*transfer-rule*]

lemma *curry-transfer* [*transfer-rule*]:
((*rel-prod A B* ===> *C*) ===> *A* ===> *B* ===> *C*) *curry curry*
⟨*proof*⟩

lemma *fun-upd-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ((*A* ===> *B*) ===> *A* ===> *B* ===> *A* ===> *B*) *fun-upd fun-upd*
⟨*proof*⟩

lemma *case-nat-transfer* [*transfer-rule*]:
(*A* ===> (*op* = ===> *A*) ===> *op* = ===> *A*) *case-nat case-nat*
⟨*proof*⟩

lemma *rec-nat-transfer* [*transfer-rule*]:
(*A* ===> (*op* = ===> *A* ===> *A*) ===> *op* = ===> *A*) *rec-nat rec-nat*
⟨*proof*⟩

lemma *funpow-transfer* [*transfer-rule*]:
(*op* = ===> (*A* ===> *A*) ===> (*A* ===> *A*)) *compow compow*
⟨*proof*⟩

lemma *mono-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
assumes [*transfer-rule*]: (*A* ===> *A* ===> *op*=) *op*≤ *op*≤
assumes [*transfer-rule*]: (*B* ===> *B* ===> *op*=) *op*≤ *op*≤
shows ((*A* ===> *B*) ===> *op*=) *mono mono*
⟨*proof*⟩

lemma *right-total-relcompp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *right-total B*
shows ((*A* ===> *B* ===> *op*=) ===> (*B* ===> *C* ===> *op*=) ===> *A* ===> *C* ===> *op*=)
(λ*R S x z*. ∃ *y*∈*Collect* (*Domainp B*). *R x y* ∧ *S y z*) *op OO*
⟨*proof*⟩

lemma *relcompp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total B*
shows ((*A* ===> *B* ===> *op*=) ===> (*B* ===> *C* ===> *op*=) ===> *A* ===> *C* ===> *op*=) *op OO op OO*
⟨*proof*⟩

lemma *right-total-Domainp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *right-total B*

shows $((A ===> B ===> op=) ===> A ===> op=)$ $(\lambda T\, x.\ \exists\, y{\in}Collect(Domainp\ B).\ T\ x\ y)$ *Domainp*
⟨*proof*⟩

lemma *Domainp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total B*
shows $((A ===> B ===> op=) ===> A ===> op=)$ *Domainp Domainp*
⟨*proof*⟩

lemma *reflp-transfer*[*transfer-rule*]:
bi-total A $\Longrightarrow$ $((A ===> A ===> op=) ===> op=)$ *reflp reflp*
right-total A $\Longrightarrow$ $((A ===> A ===> implies) ===> implies)$ *reflp reflp*
right-total A $\Longrightarrow$ $((A ===> A ===> op=) ===> implies)$ *reflp reflp*
bi-total A $\Longrightarrow$ $((A ===> A ===> rev\text{-}implies) ===> rev\text{-}implies)$ *reflp reflp*
bi-total A $\Longrightarrow$ $((A ===> A ===> op=) ===> rev\text{-}implies)$ *reflp reflp*
⟨*proof*⟩

lemma *right-unique-transfer* [*transfer-rule*]:
$[\![$ *right-total A*; *right-total B*; *bi-unique B* $]\!]$
$\Longrightarrow$ $((A ===> B ===> op=) ===> implies)$ *right-unique right-unique*
⟨*proof*⟩

lemma *left-total-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-total B*
shows $((A ===> B ===> op\ =) ===> op\ =)$ *left-total left-total*
⟨*proof*⟩

lemma *right-total-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-total B*
shows $((A ===> B ===> op\ =) ===> op\ =)$ *right-total right-total*
⟨*proof*⟩

lemma *left-unique-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A bi-total A bi-total B*
shows $((A ===> B ===> op\ =) ===> op\ =)$ *left-unique left-unique*
⟨*proof*⟩

lemma *prod-pred-parametric* [*transfer-rule*]:
$((A ===> op\ =) ===> (B ===> op\ =) ===> rel\text{-}prod\ A\ B ===> op\ =)$ *pred-prod pred-prod*
⟨*proof*⟩

lemma *apfst-parametric* [*transfer-rule*]:
$((A ===> B) ===> rel\text{-}prod\ A\ C ===> rel\text{-}prod\ B\ C)$ *apfst apfst*
⟨*proof*⟩

lemma *rel-fun-eq-eq-onp*: $(op{=} ===> eq\text{-}onp\ P) = eq\text{-}onp\ (\lambda f.\ \forall\, x.\ P(f\ x))$
⟨*proof*⟩

lemma *rel-fun-eq-onp-rel*:
shows $((eq\text{-}onp\ R) ===> S) = (\lambda f\ g.\ \forall x.\ R\ x \longrightarrow S\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *eq-onp-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ((*A* ===> *op*=) ===> *A* ===> *A* ===> *op*=) *eq-onp eq-onp*
⟨*proof*⟩

lemma *rtranclp-parametric* [*transfer-rule*]:
assumes *bi-unique A bi-total A*
shows ((*A* ===> *A* ===> *op* =) ===> *A* ===> *A* ===> *op* =) *rtranclp rtranclp*
⟨*proof*⟩

lemma *right-unique-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique B bi-total B*
shows ((*A* ===> *B* ===> *op* =) ===> *op* =) *right-unique right-unique*
⟨*proof*⟩

lemma *map-fun-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> (*C* ===> *D*) ===> (*B* ===> *C*) ===> *A* ===> *D*) *map-fun map-fun*
⟨*proof*⟩

end

40.6 *of-nat*

lemma *transfer-rule-of-nat*:
fixes *R* :: *'a*::*semiring-1* ⇒ *'b*::*semiring-1* ⇒ *bool*
assumes [*transfer-rule*]: *R 0 0 R 1 1*
rel-fun R (*rel-fun R R*) *plus plus*
shows *rel-fun HOL.eq R of-nat of-nat*
⟨*proof*⟩

end

41 Binary Numerals

theory *Num*
imports *BNF-Least-Fixpoint Transfer*
begin

41.1 The *num* type

datatype *num* = *One* | *Bit0 num* | *Bit1 num*

Increment function for type *num*

primrec *inc* :: *num* ⇒ *num*
  where
    inc One = *Bit0 One*
  | *inc* (*Bit0 x*) = *Bit1 x*
  | *inc* (*Bit1 x*) = *Bit0* (*inc x*)

Converting between type *num* and type *nat*

primrec *nat-of-num* :: *num* ⇒ *nat*
  where
    nat-of-num One = *Suc 0*
  | *nat-of-num* (*Bit0 x*) = *nat-of-num x* + *nat-of-num x*
  | *nat-of-num* (*Bit1 x*) = *Suc* (*nat-of-num x* + *nat-of-num x*)

primrec *num-of-nat* :: *nat* ⇒ *num*
  where
    num-of-nat 0 = *One*
  | *num-of-nat* (*Suc n*) = (*if 0* < *n then inc* (*num-of-nat n*) *else One*)

lemma *nat-of-num-pos*: *0* < *nat-of-num x*
  ⟨*proof*⟩

lemma *nat-of-num-neq-0*: *nat-of-num x* ≠ *0*
  ⟨*proof*⟩

lemma *nat-of-num-inc*: *nat-of-num* (*inc x*) = *Suc* (*nat-of-num x*)
  ⟨*proof*⟩

lemma *num-of-nat-double*: *0* < *n* ⟹ *num-of-nat* (*n* + *n*) = *Bit0* (*num-of-nat n*)
  ⟨*proof*⟩

Type *num* is isomorphic to the strictly positive natural numbers.

lemma *nat-of-num-inverse*: *num-of-nat* (*nat-of-num x*) = *x*
  ⟨*proof*⟩

lemma *num-of-nat-inverse*: *0* < *n* ⟹ *nat-of-num* (*num-of-nat n*) = *n*
  ⟨*proof*⟩

lemma *num-eq-iff*: *x* = *y* ⟷ *nat-of-num x* = *nat-of-num y*
  ⟨*proof*⟩

lemma *num-induct* [*case-names One inc*]:
  fixes *P* :: *num* ⇒ *bool*
  assumes *One*: *P One*
    and *inc*: ⋀*x*. *P x* ⟹ *P* (*inc x*)
  shows *P x*
⟨*proof*⟩

From now on, there are two possible models for *num*: as positive naturals (rule *num-induct*) and as digit representation (rules *num.induct*, *num.cases*).

41.2 Numeral operations

instantiation *num* :: {*plus*,*times*,*linorder*}
begin

definition [*code del*]: $m + n = \textit{num-of-nat}\ (\textit{nat-of-num}\ m + \textit{nat-of-num}\ n)$

definition [*code del*]: $m * n = \textit{num-of-nat}\ (\textit{nat-of-num}\ m * \textit{nat-of-num}\ n)$

definition [*code del*]: $m \leq n \longleftrightarrow \textit{nat-of-num}\ m \leq \textit{nat-of-num}\ n$

definition [*code del*]: $m < n \longleftrightarrow \textit{nat-of-num}\ m < \textit{nat-of-num}\ n$

instance
⟨*proof*⟩

end

lemma *nat-of-num-add*: $\textit{nat-of-num}\ (x + y) = \textit{nat-of-num}\ x + \textit{nat-of-num}\ y$
⟨*proof*⟩

lemma *nat-of-num-mult*: $\textit{nat-of-num}\ (x * y) = \textit{nat-of-num}\ x * \textit{nat-of-num}\ y$
⟨*proof*⟩

lemma *add-num-simps* [*simp*, *code*]:
$\textit{One} + \textit{One} = \textit{Bit0 One}$
$\textit{One} + \textit{Bit0}\ n = \textit{Bit1}\ n$
$\textit{One} + \textit{Bit1}\ n = \textit{Bit0}\ (n + \textit{One})$
$\textit{Bit0}\ m + \textit{One} = \textit{Bit1}\ m$
$\textit{Bit0}\ m + \textit{Bit0}\ n = \textit{Bit0}\ (m + n)$
$\textit{Bit0}\ m + \textit{Bit1}\ n = \textit{Bit1}\ (m + n)$
$\textit{Bit1}\ m + \textit{One} = \textit{Bit0}\ (m + \textit{One})$
$\textit{Bit1}\ m + \textit{Bit0}\ n = \textit{Bit1}\ (m + n)$
$\textit{Bit1}\ m + \textit{Bit1}\ n = \textit{Bit0}\ (m + n + \textit{One})$
⟨*proof*⟩

lemma *mult-num-simps* [*simp*, *code*]:
$m * \textit{One} = m$
$\textit{One} * n = n$
$\textit{Bit0}\ m * \textit{Bit0}\ n = \textit{Bit0}\ (\textit{Bit0}\ (m * n))$
$\textit{Bit0}\ m * \textit{Bit1}\ n = \textit{Bit0}\ (m * \textit{Bit1}\ n)$
$\textit{Bit1}\ m * \textit{Bit0}\ n = \textit{Bit0}\ (\textit{Bit1}\ m * n)$
$\textit{Bit1}\ m * \textit{Bit1}\ n = \textit{Bit1}\ (m + n + \textit{Bit0}\ (m * n))$
⟨*proof*⟩

lemma *eq-num-simps*:
$\textit{One} = \textit{One} \longleftrightarrow \textit{True}$

$One = Bit0\ n \longleftrightarrow False$
$One = Bit1\ n \longleftrightarrow False$
$Bit0\ m = One \longleftrightarrow False$
$Bit1\ m = One \longleftrightarrow False$
$Bit0\ m = Bit0\ n \longleftrightarrow m = n$
$Bit0\ m = Bit1\ n \longleftrightarrow False$
$Bit1\ m = Bit0\ n \longleftrightarrow False$
$Bit1\ m = Bit1\ n \longleftrightarrow m = n$
⟨*proof*⟩

lemma *le-num-simps* [*simp*, *code*]:
$One \leq n \longleftrightarrow True$
$Bit0\ m \leq One \longleftrightarrow False$
$Bit1\ m \leq One \longleftrightarrow False$
$Bit0\ m \leq Bit0\ n \longleftrightarrow m \leq n$
$Bit0\ m \leq Bit1\ n \longleftrightarrow m \leq n$
$Bit1\ m \leq Bit1\ n \longleftrightarrow m \leq n$
$Bit1\ m \leq Bit0\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *less-num-simps* [*simp*, *code*]:
$m < One \longleftrightarrow False$
$One < Bit0\ n \longleftrightarrow True$
$One < Bit1\ n \longleftrightarrow True$
$Bit0\ m < Bit0\ n \longleftrightarrow m < n$
$Bit0\ m < Bit1\ n \longleftrightarrow m \leq n$
$Bit1\ m < Bit1\ n \longleftrightarrow m < n$
$Bit1\ m < Bit0\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *le-num-One-iff*: $x \leq num.One \longleftrightarrow x = num.One$
⟨*proof*⟩

Rules using *One* and *inc* as constructors.

lemma *add-One*: $x + One = inc\ x$
⟨*proof*⟩

lemma *add-One-commute*: $One + n = n + One$
⟨*proof*⟩

lemma *add-inc*: $x + inc\ y = inc\ (x + y)$
⟨*proof*⟩

lemma *mult-inc*: $x * inc\ y = x * y + x$
⟨*proof*⟩

The *num-of-nat* conversion.

lemma *num-of-nat-One*: $n \leq 1 \Longrightarrow num\text{-}of\text{-}nat\ n = One$
⟨*proof*⟩

lemma *num-of-nat-plus-distrib*:
$0 < m \Longrightarrow 0 < n \Longrightarrow$ *num-of-nat* $(m + n) =$ *num-of-nat* $m +$ *num-of-nat* n
⟨*proof*⟩

A double-and-decrement function.

primrec *BitM* :: *num* ⇒ *num*
where
BitM One = *One*
| *BitM* (*Bit0 n*) = *Bit1* (*BitM n*)
| *BitM* (*Bit1 n*) = *Bit1* (*Bit0 n*)

lemma *BitM-plus-one*: *BitM n* + *One* = *Bit0 n*
⟨*proof*⟩

lemma *one-plus-BitM*: *One* + *BitM n* = *Bit0 n*
⟨*proof*⟩

Squaring and exponentiation.

primrec *sqr* :: *num* ⇒ *num*
where
sqr One = *One*
| *sqr* (*Bit0 n*) = *Bit0* (*Bit0* (*sqr n*))
| *sqr* (*Bit1 n*) = *Bit1* (*Bit0* (*sqr n* + *n*))

primrec *pow* :: *num* ⇒ *num* ⇒ *num*
where
pow x One = *x*
| *pow x* (*Bit0 y*) = *sqr* (*pow x y*)
| *pow x* (*Bit1 y*) = *sqr* (*pow x y*) * *x*

lemma *nat-of-num-sqr*: *nat-of-num* (*sqr x*) = *nat-of-num x* * *nat-of-num x*
⟨*proof*⟩

lemma *sqr-conv-mult*: *sqr x* = *x* * *x*
⟨*proof*⟩

41.3 Binary numerals

We embed binary representations into a generic algebraic structure using *numeral*.

class *numeral* = *one* + *semigroup-add*
begin

primrec *numeral* :: *num* ⇒ *'a*
where
numeral-One: *numeral One* = *1*
| *numeral-Bit0*: *numeral* (*Bit0 n*) = *numeral n* + *numeral n*

| *numeral-Bit1*: *numeral* (*Bit1 n*) = *numeral n* + *numeral n* + *1*

lemma *numeral-code* [*code*]:
numeral One = *1*
numeral (*Bit0 n*) = (*let m* = *numeral n in m* + *m*)
numeral (*Bit1 n*) = (*let m* = *numeral n in m* + *m* + *1*)
⟨*proof*⟩

lemma *one-plus-numeral-commute*: *1* + *numeral x* = *numeral x* + *1*
⟨*proof*⟩

lemma *numeral-inc*: *numeral* (*inc x*) = *numeral x* + *1*
⟨*proof*⟩

declare *numeral.simps* [*simp del*]

abbreviation *Numeral1* ≡ *numeral One*

declare *numeral-One* [*code-post*]

end

Numeral syntax.

syntax
-*Numeral* :: *num-const* ⇒ *'a* (-)

⟨*ML*⟩

41.4 Class-specific numeral rules

numeral is a morphism.

41.4.1 Structures with addition: class *numeral*

context *numeral*
begin

lemma *numeral-add*: *numeral* (*m* + *n*) = *numeral m* + *numeral n*
⟨*proof*⟩

lemma *numeral-plus-numeral*: *numeral m* + *numeral n* = *numeral* (*m* + *n*)
⟨*proof*⟩

lemma *numeral-plus-one*: *numeral n* + *1* = *numeral* (*n* + *One*)
⟨*proof*⟩

lemma *one-plus-numeral*: *1* + *numeral n* = *numeral* (*One* + *n*)
⟨*proof*⟩

lemma *one-add-one*: $1 + 1 = 2$
  ⟨*proof*⟩

lemmas *add-numeral-special* =
  numeral-plus-one one-plus-numeral one-add-one

end

41.4.2 Structures with negation: class *neg-numeral*

class *neg-numeral* = *numeral* + *group-add*
begin

lemma *uminus-numeral-One*: $-$ *Numeral1* $= -$ *1*
  ⟨*proof*⟩

Numerals form an abelian subgroup.

inductive *is-num* :: $'a \Rightarrow$ *bool*
  where
    is-num 1
  | *is-num* $x \Longrightarrow$ *is-num* $(- x)$
  | *is-num* $x \Longrightarrow$ *is-num* $y \Longrightarrow$ *is-num* $(x + y)$

lemma *is-num-numeral*: *is-num* (*numeral* k)
  ⟨*proof*⟩

lemma *is-num-add-commute*: *is-num* $x \Longrightarrow$ *is-num* $y \Longrightarrow x + y = y + x$
  ⟨*proof*⟩

lemma *is-num-add-left-commute*: *is-num* $x \Longrightarrow$ *is-num* $y \Longrightarrow x + (y + z) = y + (x + z)$
  ⟨*proof*⟩

lemmas *is-num-normalize* =
  add.assoc is-num-add-commute is-num-add-left-commute
  is-num.intros is-num-numeral
  minus-add

definition *dbl* :: $'a \Rightarrow 'a$
  where *dbl* $x = x + x$

definition *dbl-inc* :: $'a \Rightarrow 'a$
  where *dbl-inc* $x = x + x + 1$

definition *dbl-dec* :: $'a \Rightarrow 'a$
  where *dbl-dec* $x = x + x - 1$

definition *sub* :: *num* $\Rightarrow$ *num* $\Rightarrow 'a$
  where *sub* $k\ l$ = *numeral* k $-$ *numeral* l

lemma *numeral-BitM*: *numeral* (*BitM n*) = *numeral* (*Bit0 n*) − *1*
⟨*proof*⟩

lemma *dbl-simps* [*simp*]:
dbl (− *numeral k*) = − *dbl* (*numeral k*)
dbl 0 = *0*
dbl 1 = *2*
dbl (− *1*) = − *2*
dbl (*numeral k*) = *numeral* (*Bit0 k*)
⟨*proof*⟩

lemma *dbl-inc-simps* [*simp*]:
dbl-inc (− *numeral k*) = − *dbl-dec* (*numeral k*)
dbl-inc 0 = *1*
dbl-inc 1 = *3*
dbl-inc (− *1*) = − *1*
dbl-inc (*numeral k*) = *numeral* (*Bit1 k*)
⟨*proof*⟩

lemma *dbl-dec-simps* [*simp*]:
dbl-dec (− *numeral k*) = − *dbl-inc* (*numeral k*)
dbl-dec 0 = − *1*
dbl-dec 1 = *1*
dbl-dec (− *1*) = − *3*
dbl-dec (*numeral k*) = *numeral* (*BitM k*)
⟨*proof*⟩

lemma *sub-num-simps* [*simp*]:
sub One One = *0*
sub One (*Bit0 l*) = − *numeral* (*BitM l*)
sub One (*Bit1 l*) = − *numeral* (*Bit0 l*)
sub (*Bit0 k*) *One* = *numeral* (*BitM k*)
sub (*Bit1 k*) *One* = *numeral* (*Bit0 k*)
sub (*Bit0 k*) (*Bit0 l*) = *dbl* (*sub k l*)
sub (*Bit0 k*) (*Bit1 l*) = *dbl-dec* (*sub k l*)
sub (*Bit1 k*) (*Bit0 l*) = *dbl-inc* (*sub k l*)
sub (*Bit1 k*) (*Bit1 l*) = *dbl* (*sub k l*)
⟨*proof*⟩

lemma *add-neg-numeral-simps*:
numeral m + − *numeral n* = *sub m n*
− *numeral m* + *numeral n* = *sub n m*
− *numeral m* + − *numeral n* = − (*numeral m* + *numeral n*)
⟨*proof*⟩

lemma *add-neg-numeral-special*:
1 + − *numeral m* = *sub One m*
− *numeral m* + *1* = *sub One m*

numeral m + − *1* = *sub m One*
− *1* + *numeral n* = *sub n One*
− *1* + − *numeral n* = − *numeral* (*inc n*)
− *numeral m* + − *1* = − *numeral* (*inc m*)
1 + − *1* = *0*
− *1* + *1* = *0*
− *1* + − *1* = − *2*
⟨*proof*⟩

lemma *diff-numeral-simps*:
numeral m − *numeral n* = *sub m n*
numeral m − − *numeral n* = *numeral* (*m* + *n*)
− *numeral m* − *numeral n* = − *numeral* (*m* + *n*)
− *numeral m* − − *numeral n* = *sub n m*
⟨*proof*⟩

lemma *diff-numeral-special*:
1 − *numeral n* = *sub One n*
numeral m − *1* = *sub m One*
1 − − *numeral n* = *numeral* (*One* + *n*)
− *numeral m* − *1* = − *numeral* (*m* + *One*)
− *1* − *numeral n* = − *numeral* (*inc n*)
numeral m − − *1* = *numeral* (*inc m*)
− *1* − − *numeral n* = *sub n One*
− *numeral m* − − *1* = *sub One m*
1 − *1* = *0*
− *1* − *1* = − *2*
1 − − *1* = *2*
− *1* − − *1* = *0*
⟨*proof*⟩

end

41.4.3 Structures with multiplication: class *semiring-numeral*

class *semiring-numeral* = *semiring* + *monoid-mult*
begin

subclass *numeral* ⟨*proof*⟩

lemma *numeral-mult*: *numeral* (*m* ∗ *n*) = *numeral m* ∗ *numeral n*
⟨*proof*⟩

lemma *numeral-times-numeral*: *numeral m* ∗ *numeral n* = *numeral* (*m* ∗ *n*)
⟨*proof*⟩

lemma *mult-2*: *2* ∗ *z* = *z* + *z*
⟨*proof*⟩

lemma *mult-2-right*: $z * 2 = z + z$
⟨*proof*⟩

end

41.4.4 Structures with a zero: class *semiring-1*

context *semiring-1*
begin

subclass *semiring-numeral* ⟨*proof*⟩

lemma *of-nat-numeral* [*simp*]: *of-nat* (*numeral n*) = *numeral n*
⟨*proof*⟩

lemma *numeral-unfold-funpow*:
numeral k = (*op* + *1* ^^ *numeral k*) *0*
⟨*proof*⟩

end

lemma *transfer-rule-numeral*:
fixes *R* :: *'a::semiring-1* ⇒ *'b::semiring-1* ⇒ *bool*
assumes [*transfer-rule*]: *R 0 0 R 1 1*
rel-fun R (*rel-fun R R*) *plus plus*
shows *rel-fun HOL.eq R numeral numeral*
⟨*proof*⟩

lemma *nat-of-num-numeral* [*code-abbrev*]: *nat-of-num* = *numeral*
⟨*proof*⟩

lemma *nat-of-num-code* [*code*]:
nat-of-num One = *1*
nat-of-num (*Bit0 n*) = (*let m* = *nat-of-num n in m* + *m*)
nat-of-num (*Bit1 n*) = (*let m* = *nat-of-num n in Suc* (*m* + *m*))
⟨*proof*⟩

41.4.5 Equality: class *semiring-char-0*

context *semiring-char-0*
begin

lemma *numeral-eq-iff*: *numeral m* = *numeral n* ⟷ *m* = *n*
⟨*proof*⟩

lemma *numeral-eq-one-iff*: *numeral n* = *1* ⟷ *n* = *One*
⟨*proof*⟩

lemma *one-eq-numeral-iff*: *1* = *numeral n* ⟷ *One* = *n*
⟨*proof*⟩

lemma *numeral-neq-zero*: $numeral\ n \neq 0$
⟨*proof*⟩

lemma *zero-neq-numeral*: $0 \neq numeral\ n$
⟨*proof*⟩

lemmas *eq-numeral-simps* [*simp*] =
numeral-eq-iff
numeral-eq-one-iff
one-eq-numeral-iff
numeral-neq-zero
zero-neq-numeral

end

41.4.6 Comparisons: class *linordered-semidom*

Could be perhaps more general than here.

context *linordered-semidom*
begin

lemma *numeral-le-iff*: $numeral\ m \leq numeral\ n \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *one-le-numeral*: $1 \leq numeral\ n$
⟨*proof*⟩

lemma *numeral-le-one-iff*: $numeral\ n \leq 1 \longleftrightarrow n \leq One$
⟨*proof*⟩

lemma *numeral-less-iff*: $numeral\ m < numeral\ n \longleftrightarrow m < n$
⟨*proof*⟩

lemma *not-numeral-less-one*: $\neg\ numeral\ n < 1$
⟨*proof*⟩

lemma *one-less-numeral-iff*: $1 < numeral\ n \longleftrightarrow One < n$
⟨*proof*⟩

lemma *zero-le-numeral*: $0 \leq numeral\ n$
⟨*proof*⟩

lemma *zero-less-numeral*: $0 < numeral\ n$
⟨*proof*⟩

lemma *not-numeral-le-zero*: $\neg\ numeral\ n \leq 0$
⟨*proof*⟩

lemma *not-numeral-less-zero*: $\neg$ *numeral n* $<$ *0*
⟨*proof*⟩

lemmas *le-numeral-extra* =
zero-le-one not-one-le-zero
order-refl [*of 0*] *order-refl* [*of 1*]

lemmas *less-numeral-extra* =
zero-less-one not-one-less-zero
less-irrefl [*of 0*] *less-irrefl* [*of 1*]

lemmas *le-numeral-simps* [*simp*] =
numeral-le-iff
one-le-numeral
numeral-le-one-iff
zero-le-numeral
not-numeral-le-zero

lemmas *less-numeral-simps* [*simp*] =
numeral-less-iff
one-less-numeral-iff
not-numeral-less-one
zero-less-numeral
not-numeral-less-zero

lemma *min-0-1* [*simp*]:
fixes *min′* :: *′a* ⇒ *′a* ⇒ *′a*
defines *min′* ≡ *min*
shows
min′ 0 1 = *0*
min′ 1 0 = *0*
min′ 0 (*numeral x*) = *0*
min′ (*numeral x*) *0* = *0*
min′ 1 (*numeral x*) = *1*
min′ (*numeral x*) *1* = *1*
⟨*proof*⟩

lemma *max-0-1* [*simp*]:
fixes *max′* :: *′a* ⇒ *′a* ⇒ *′a*
defines *max′* ≡ *max*
shows
max′ 0 1 = *1*
max′ 1 0 = *1*
max′ 0 (*numeral x*) = *numeral x*
max′ (*numeral x*) *0* = *numeral x*
max′ 1 (*numeral x*) = *numeral x*
max′ (*numeral x*) *1* = *numeral x*
⟨*proof*⟩

end

41.4.7 Multiplication and negation: class *ring-1*

context *ring-1*
begin

subclass *neg-numeral* ⟨*proof*⟩

lemma *mult-neg-numeral-simps*:
$-$ *numeral m* $*$ $-$ *numeral n* $=$ *numeral* $(m * n)$
$-$ *numeral m* $*$ *numeral n* $= -$ *numeral* $(m * n)$
numeral m $* -$ *numeral n* $= -$ *numeral* $(m * n)$
⟨*proof*⟩

lemma *mult-minus1* [*simp*]: $-$ *1* $*$ $z = -$ z
⟨*proof*⟩

lemma *mult-minus1-right* [*simp*]: $z * -$ *1* $= -$ z
⟨*proof*⟩

end

41.4.8 Equality using *iszero* for rings with non-zero characteristic

context *ring-1*
begin

definition *iszero* :: $'a \Rightarrow$ *bool*
where *iszero z* $\longleftrightarrow$ $z = 0$

lemma *iszero-0* [*simp*]: *iszero 0*
⟨*proof*⟩

lemma *not-iszero-1* [*simp*]: $\neg$ *iszero 1*
⟨*proof*⟩

lemma *not-iszero-Numeral1*: $\neg$ *iszero Numeral1*
⟨*proof*⟩

lemma *not-iszero-neg-1* [*simp*]: $\neg$ *iszero* ($-$ *1*)
⟨*proof*⟩

lemma *not-iszero-neg-Numeral1*: $\neg$ *iszero* ($-$ *Numeral1*)
⟨*proof*⟩

lemma *iszero-neg-numeral* [*simp*]: *iszero* ($-$ *numeral w*) $\longleftrightarrow$ *iszero* (*numeral w*)
⟨*proof*⟩

lemma *eq-iff-iszero-diff*: $x = y \longleftrightarrow$ *iszero* $(x - y)$

⟨*proof*⟩

The *eq-numeral-iff-iszero* lemmas are not declared [*simp*] by default, because for rings of characteristic zero, better simp rules are possible. For a type like integers mod n, type-instantiated versions of these rules should be added to the simplifier, along with a type-specific rule for deciding propositions of the form *iszero* (*numeral w*).

bh: Maybe it would not be so bad to just declare these as simp rules anyway? I should test whether these rules take precedence over the *ring-char-0* rules in the simplifier.

lemma *eq-numeral-iff-iszero*:
$numeral\ x = numeral\ y \longleftrightarrow iszero\ (sub\ x\ y)$
$numeral\ x = -\ numeral\ y \longleftrightarrow iszero\ (numeral\ (x + y))$
$-\ numeral\ x = numeral\ y \longleftrightarrow iszero\ (numeral\ (x + y))$
$-\ numeral\ x = -\ numeral\ y \longleftrightarrow iszero\ (sub\ y\ x)$
$numeral\ x = 1 \longleftrightarrow iszero\ (sub\ x\ One)$
$1 = numeral\ y \longleftrightarrow iszero\ (sub\ One\ y)$
$-\ numeral\ x = 1 \longleftrightarrow iszero\ (numeral\ (x + One))$
$1 = -\ numeral\ y \longleftrightarrow iszero\ (numeral\ (One + y))$
$numeral\ x = 0 \longleftrightarrow iszero\ (numeral\ x)$
$0 = numeral\ y \longleftrightarrow iszero\ (numeral\ y)$
$-\ numeral\ x = 0 \longleftrightarrow iszero\ (numeral\ x)$
$0 = -\ numeral\ y \longleftrightarrow iszero\ (numeral\ y)$
⟨*proof*⟩

end

41.4.9 Equality and negation: class *ring-char-0*

context *ring-char-0*
begin

lemma *not-iszero-numeral* [*simp*]: $\neg\ iszero\ (numeral\ w)$
⟨*proof*⟩

lemma *neg-numeral-eq-iff*: $-\ numeral\ m = -\ numeral\ n \longleftrightarrow m = n$
⟨*proof*⟩

lemma *numeral-neq-neg-numeral*: $numeral\ m \neq -\ numeral\ n$
⟨*proof*⟩

lemma *neg-numeral-neq-numeral*: $-\ numeral\ m \neq numeral\ n$
⟨*proof*⟩

lemma *zero-neq-neg-numeral*: $0 \neq -\ numeral\ n$
⟨*proof*⟩

lemma *neg-numeral-neq-zero*: $-\ numeral\ n \neq 0$

⟨*proof*⟩

lemma *one-neq-neg-numeral*: $1 \neq -$ *numeral n*
⟨*proof*⟩

lemma *neg-numeral-neq-one*: $-$ *numeral n* $\neq 1$
⟨*proof*⟩

lemma *neg-one-neq-numeral*: $- 1 \neq$ *numeral n*
⟨*proof*⟩

lemma *numeral-neq-neg-one*: *numeral n* $\neq - 1$
⟨*proof*⟩

lemma *neg-one-eq-numeral-iff*: $- 1 = -$ *numeral n* $\longleftrightarrow n = One$
⟨*proof*⟩

lemma *numeral-eq-neg-one-iff*: $-$ *numeral n* $= - 1 \longleftrightarrow n = One$
⟨*proof*⟩

lemma *neg-one-neq-zero*: $- 1 \neq 0$
⟨*proof*⟩

lemma *zero-neq-neg-one*: $0 \neq - 1$
⟨*proof*⟩

lemma *neg-one-neq-one*: $- 1 \neq 1$
⟨*proof*⟩

lemma *one-neq-neg-one*: $1 \neq - 1$
⟨*proof*⟩

lemmas *eq-neg-numeral-simps* [*simp*] =
neg-numeral-eq-iff
numeral-neq-neg-numeral neg-numeral-neq-numeral
one-neq-neg-numeral neg-numeral-neq-one
zero-neq-neg-numeral neg-numeral-neq-zero
neg-one-neq-numeral numeral-neq-neg-one
neg-one-eq-numeral-iff numeral-eq-neg-one-iff
neg-one-neq-zero zero-neq-neg-one
neg-one-neq-one one-neq-neg-one

end

41.4.10 Structures with negation and order: class *linordered-idom*

context *linordered-idom*
begin

subclass *ring-char-0* ⟨*proof*⟩

lemma *neg-numeral-le-iff*: $-$ *numeral* $m \leq -$ *numeral* $n \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *neg-numeral-less-iff*: $-$ *numeral* $m < -$ *numeral* $n \longleftrightarrow n < m$
⟨*proof*⟩

lemma *neg-numeral-less-zero*: $-$ *numeral* $n < 0$
⟨*proof*⟩

lemma *neg-numeral-le-zero*: $-$ *numeral* $n \leq 0$
⟨*proof*⟩

lemma *not-zero-less-neg-numeral*: $\neg\ 0 < -$ *numeral* n
⟨*proof*⟩

lemma *not-zero-le-neg-numeral*: $\neg\ 0 \leq -$ *numeral* n
⟨*proof*⟩

lemma *neg-numeral-less-numeral*: $-$ *numeral* $m <$ *numeral* n
⟨*proof*⟩

lemma *neg-numeral-le-numeral*: $-$ *numeral* $m \leq$ *numeral* n
⟨*proof*⟩

lemma *not-numeral-less-neg-numeral*: $\neg$ *numeral* $m < -$ *numeral* n
⟨*proof*⟩

lemma *not-numeral-le-neg-numeral*: $\neg$ *numeral* $m \leq -$ *numeral* n
⟨*proof*⟩

lemma *neg-numeral-less-one*: $-$ *numeral* $m < 1$
⟨*proof*⟩

lemma *neg-numeral-le-one*: $-$ *numeral* $m \leq 1$
⟨*proof*⟩

lemma *not-one-less-neg-numeral*: $\neg\ 1 < -$ *numeral* m
⟨*proof*⟩

lemma *not-one-le-neg-numeral*: $\neg\ 1 \leq -$ *numeral* m
⟨*proof*⟩

lemma *not-numeral-less-neg-one*: $\neg$ *numeral* $m < -\ 1$
⟨*proof*⟩

lemma *not-numeral-le-neg-one*: $\neg$ *numeral* $m \leq -\ 1$
⟨*proof*⟩

lemma *neg-one-less-numeral*: $- 1 < numeral\ m$
⟨*proof*⟩

lemma *neg-one-le-numeral*: $- 1 \leq numeral\ m$
⟨*proof*⟩

lemma *neg-numeral-less-neg-one-iff*: $- numeral\ m < - 1 \longleftrightarrow m \neq One$
⟨*proof*⟩

lemma *neg-numeral-le-neg-one*: $- numeral\ m \leq - 1$
⟨*proof*⟩

lemma *not-neg-one-less-neg-numeral*: $\neg - 1 < - numeral\ m$
⟨*proof*⟩

lemma *not-neg-one-le-neg-numeral-iff*: $\neg - 1 \leq - numeral\ m \longleftrightarrow m \neq One$
⟨*proof*⟩

lemma *sub-non-negative*: $sub\ n\ m \geq 0 \longleftrightarrow n \geq m$
⟨*proof*⟩

lemma *sub-positive*: $sub\ n\ m > 0 \longleftrightarrow n > m$
⟨*proof*⟩

lemma *sub-non-positive*: $sub\ n\ m \leq 0 \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *sub-negative*: $sub\ n\ m < 0 \longleftrightarrow n < m$
⟨*proof*⟩

lemmas *le-neg-numeral-simps* [*simp*] =
neg-numeral-le-iff
neg-numeral-le-numeral not-numeral-le-neg-numeral
neg-numeral-le-zero not-zero-le-neg-numeral
neg-numeral-le-one not-one-le-neg-numeral
neg-one-le-numeral not-numeral-le-neg-one
neg-numeral-le-neg-one not-neg-one-le-neg-numeral-iff

lemma *le-minus-one-simps* [*simp*]:
$- 1 \leq 0$
$- 1 \leq 1$
$\neg 0 \leq - 1$
$\neg 1 \leq - 1$
⟨*proof*⟩

lemmas *less-neg-numeral-simps* [*simp*] =
neg-numeral-less-iff
neg-numeral-less-numeral not-numeral-less-neg-numeral

neg-numeral-less-zero not-zero-less-neg-numeral
neg-numeral-less-one not-one-less-neg-numeral
neg-one-less-numeral not-numeral-less-neg-one
neg-numeral-less-neg-one-iff not-neg-one-less-neg-numeral

lemma *less-minus-one-simps* [*simp*]:
$- 1 < 0$
$- 1 < 1$
$\neg\ 0 < - 1$
$\neg\ 1 < - 1$
⟨*proof*⟩

lemma *abs-numeral* [*simp*]: $|numeral\ n| = numeral\ n$
⟨*proof*⟩

lemma *abs-neg-numeral* [*simp*]: $|- numeral\ n| = numeral\ n$
⟨*proof*⟩

lemma *abs-neg-one* [*simp*]: $|- 1| = 1$
⟨*proof*⟩

end

41.4.11 Natural numbers

lemma *Suc-1* [*simp*]: $Suc\ 1 = 2$
⟨*proof*⟩

lemma *Suc-numeral* [*simp*]: $Suc\ (numeral\ n) = numeral\ (n + One)$
⟨*proof*⟩

definition *pred-numeral* :: $num \Rightarrow nat$
where [*code del*]: $pred\text{-}numeral\ k = numeral\ k - 1$

lemma *numeral-eq-Suc*: $numeral\ k = Suc\ (pred\text{-}numeral\ k)$
⟨*proof*⟩

lemma *eval-nat-numeral*:
$numeral\ One = Suc\ 0$
$numeral\ (Bit0\ n) = Suc\ (numeral\ (BitM\ n))$
$numeral\ (Bit1\ n) = Suc\ (numeral\ (Bit0\ n))$
⟨*proof*⟩

lemma *pred-numeral-simps* [*simp*]:
$pred\text{-}numeral\ One = 0$
$pred\text{-}numeral\ (Bit0\ k) = numeral\ (BitM\ k)$
$pred\text{-}numeral\ (Bit1\ k) = numeral\ (Bit0\ k)$
⟨*proof*⟩

lemma *numeral-2-eq-2*: $2 = Suc\ (Suc\ 0)$
⟨*proof*⟩

lemma *numeral-3-eq-3*: $3 = Suc\ (Suc\ (Suc\ 0))$
⟨*proof*⟩

lemma *numeral-1-eq-Suc-0*: $Numeral1 = Suc\ 0$
⟨*proof*⟩

lemma *Suc-nat-number-of-add*: $Suc\ (numeral\ v + n) = numeral\ (v + One) + n$
⟨*proof*⟩

lemma *numerals*: $Numeral1 = (1::nat)$ $2 = Suc\ (Suc\ 0)$
⟨*proof*⟩

lemmas *numeral-nat* = *eval-nat-numeral BitM.simps One-nat-def*

Comparisons involving *Suc*.

lemma *eq-numeral-Suc* [*simp*]: $numeral\ k = Suc\ n \longleftrightarrow pred\text{-}numeral\ k = n$
⟨*proof*⟩

lemma *Suc-eq-numeral* [*simp*]: $Suc\ n = numeral\ k \longleftrightarrow n = pred\text{-}numeral\ k$
⟨*proof*⟩

lemma *less-numeral-Suc* [*simp*]: $numeral\ k < Suc\ n \longleftrightarrow pred\text{-}numeral\ k < n$
⟨*proof*⟩

lemma *less-Suc-numeral* [*simp*]: $Suc\ n < numeral\ k \longleftrightarrow n < pred\text{-}numeral\ k$
⟨*proof*⟩

lemma *le-numeral-Suc* [*simp*]: $numeral\ k \le Suc\ n \longleftrightarrow pred\text{-}numeral\ k \le n$
⟨*proof*⟩

lemma *le-Suc-numeral* [*simp*]: $Suc\ n \le numeral\ k \longleftrightarrow n \le pred\text{-}numeral\ k$
⟨*proof*⟩

lemma *diff-Suc-numeral* [*simp*]: $Suc\ n - numeral\ k = n - pred\text{-}numeral\ k$
⟨*proof*⟩

lemma *diff-numeral-Suc* [*simp*]: $numeral\ k - Suc\ n = pred\text{-}numeral\ k - n$
⟨*proof*⟩

lemma *max-Suc-numeral* [*simp*]: $max\ (Suc\ n)\ (numeral\ k) = Suc\ (max\ n\ (pred\text{-}numeral\ k))$
⟨*proof*⟩

lemma *max-numeral-Suc* [*simp*]: $max\ (numeral\ k)\ (Suc\ n) = Suc\ (max\ (pred\text{-}numeral\ k)\ n)$
⟨*proof*⟩

lemma *min-Suc-numeral* [*simp*]: *min* (*Suc n*) (*numeral k*) = *Suc* (*min n* (*pred-numeral k*))
⟨*proof*⟩

lemma *min-numeral-Suc* [*simp*]: *min* (*numeral k*) (*Suc n*) = *Suc* (*min* (*pred-numeral k*) *n*)
⟨*proof*⟩

For *case-nat* and *rec-nat*.

lemma *case-nat-numeral* [*simp*]: *case-nat a f* (*numeral v*) = (*let pv* = *pred-numeral v in f pv*)
⟨*proof*⟩

lemma *case-nat-add-eq-if* [*simp*]:
case-nat a f ((*numeral v*) + *n*) = (*let pv* = *pred-numeral v in f* (*pv* + *n*))
⟨*proof*⟩

lemma *rec-nat-numeral* [*simp*]:
rec-nat a f (*numeral v*) = (*let pv* = *pred-numeral v in f pv* (*rec-nat a f pv*))
⟨*proof*⟩

lemma *rec-nat-add-eq-if* [*simp*]:
rec-nat a f (*numeral v* + *n*) = (*let pv* = *pred-numeral v in f* (*pv* + *n*) (*rec-nat a f* (*pv* + *n*)))
⟨*proof*⟩

Case analysis on $n < (2::'a)$.

lemma *less-2-cases*: $n < 2 \Longrightarrow n = 0 \vee n = Suc\ 0$
⟨*proof*⟩

Removal of Small Numerals: 0, 1 and (in additive positions) 2.

bh: Are these rules really a good idea?

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*⟩

lemmas *nat-1-add-1* = *one-add-one* [**where** *'a*=*nat*]

41.5 Particular lemmas concerning $2::'a$

context *linordered-field*

begin

subclass *field-char-0* ⟨*proof*⟩

lemma *half-gt-zero-iff*: $0 < a / 2 \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *half-gt-zero* [*simp*]: $0 < a \Longrightarrow 0 < a / 2$
⟨*proof*⟩

end

41.6 Numeral equations as default simplification rules

declare (**in** *numeral*) *numeral-One* [*simp*]
declare (**in** *numeral*) *numeral-plus-numeral* [*simp*]
declare (**in** *numeral*) *add-numeral-special* [*simp*]
declare (**in** *neg-numeral*) *add-neg-numeral-simps* [*simp*]
declare (**in** *neg-numeral*) *add-neg-numeral-special* [*simp*]
declare (**in** *neg-numeral*) *diff-numeral-simps* [*simp*]
declare (**in** *neg-numeral*) *diff-numeral-special* [*simp*]
declare (**in** *semiring-numeral*) *numeral-times-numeral* [*simp*]
declare (**in** *ring-1*) *mult-neg-numeral-simps* [*simp*]

41.7 Setting up simprocs

lemma *mult-numeral-1*: *Numeral1* $*$ $a = a$
for a :: $'a$::*semiring-numeral*
⟨*proof*⟩

lemma *mult-numeral-1-right*: a $*$ *Numeral1* $= a$
for a :: $'a$::*semiring-numeral*
⟨*proof*⟩

lemma *divide-numeral-1*: a / *Numeral1* $= a$
for a :: $'a$::*field*
⟨*proof*⟩

lemma *inverse-numeral-1*: *inverse Numeral1* = (*Numeral1*::$'a$::*division-ring*)
⟨*proof*⟩

Theorem lists for the cancellation simprocs. The use of a binary numeral for 1 reduces the number of special cases.

lemma *mult-1s*:
Numeral1 $*$ $a = a$
a $*$ *Numeral1* $= a$
$-$ *Numeral1* $*$ $b = -\ b$
b $*$ $-$ *Numeral1* $= -\ b$
for a :: $'a$::*semiring-numeral* **and** b :: $'b$::*ring-1*

⟨*proof*⟩

⟨*ML*⟩

41.7.1 Simplification of arithmetic operations on integer constants

lemmas *arith-special* =
add-numeral-special add-neg-numeral-special
diff-numeral-special

lemmas *arith-extra-simps* =
numeral-plus-numeral add-neg-numeral-simps add-0-left add-0-right
minus-zero
diff-numeral-simps diff-0 diff-0-right
numeral-times-numeral mult-neg-numeral-simps
mult-zero-left mult-zero-right
abs-numeral abs-neg-numeral

For making a minimal simpset, one must include these default simprules. Also include *simp-thms*.

lemmas *arith-simps* =
add-num-simps mult-num-simps sub-num-simps
BitM.simps dbl-simps dbl-inc-simps dbl-dec-simps
abs-zero abs-one arith-extra-simps

lemmas *more-arith-simps* =
neg-le-iff-le
minus-zero left-minus right-minus
mult-1-left mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult.assoc

lemmas *of-nat-simps* =
of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult

Simplification of relational operations.

lemmas *eq-numeral-extra* =
zero-neq-one one-neq-zero

lemmas *rel-simps* =
le-num-simps less-num-simps eq-num-simps
le-numeral-simps le-neg-numeral-simps le-minus-one-simps le-numeral-extra
less-numeral-simps less-neg-numeral-simps less-minus-one-simps less-numeral-extra
eq-numeral-simps eq-neg-numeral-simps eq-numeral-extra

lemma *Let-numeral* [*simp*]: *Let* (*numeral v*) *f* = *f* (*numeral v*)
— Unfold all *let*s involving constants
⟨*proof*⟩

lemma *Let-neg-numeral* [*simp*]: *Let* (− *numeral v*) *f* = *f* (− *numeral v*)
— Unfold all *let*s involving constants
⟨*proof*⟩

⟨*ML*⟩

41.7.2 Simplification of arithmetic when nested to the right

lemma *add-numeral-left* [*simp*]: *numeral v* + (*numeral w* + *z*) = (*numeral*(*v* + *w*) + *z*)
⟨*proof*⟩

lemma *add-neg-numeral-left* [*simp*]:
numeral v + (− *numeral w* + *y*) = (*sub v w* + *y*)
− *numeral v* + (*numeral w* + *y*) = (*sub w v* + *y*)
− *numeral v* + (− *numeral w* + *y*) = (− *numeral*(*v* + *w*) + *y*)
⟨*proof*⟩

lemma *mult-numeral-left* [*simp*]:
numeral v ∗ (*numeral w* ∗ *z*) = (*numeral*(*v* ∗ *w*) ∗ *z* :: ′*a*::*semiring-numeral*)
− *numeral v* ∗ (*numeral w* ∗ *y*) = (− *numeral*(*v* ∗ *w*) ∗ *y* :: ′*b*::*ring-1*)
numeral v ∗ (− *numeral w* ∗ *y*) = (− *numeral*(*v* ∗ *w*) ∗ *y* :: ′*b*::*ring-1*)
− *numeral v* ∗ (− *numeral w* ∗ *y*) = (*numeral*(*v* ∗ *w*) ∗ *y* :: ′*b*::*ring-1*)
⟨*proof*⟩

hide-const (**open**) *One Bit0 Bit1 BitM inc pow sqr sub dbl dbl-inc dbl-dec*

41.8 Code module namespace

code-identifier
code-module *Num* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

41.9 Printing of evaluated natural numbers as numerals

lemma [*code-post*]:
Suc 0 = *1*
Suc 1 = *2*
Suc (*numeral n*) = *numeral* (*Num.inc n*)
⟨*proof*⟩

lemmas [*code-post*] = *Num.inc.simps*

end

42 Exponentiation

theory *Power*
imports *Num*

begin

42.1 Powers for Arbitrary Monoids

class *power* = *one* + *times*
begin

primrec *power* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixr** ˆ *80*)
  where
    power-0: a ˆ $0 = 1$
  | *power-Suc*: a ˆ $Suc\ n = a * a$ ˆ n

notation (*latex* **output**)
  power ((-ˆ) [*1000*] *1000*)

Special syntax for squares.

abbreviation *power2* :: $'a \Rightarrow 'a$ ((-^{2}) [*1000*] *999*)
  where $x^2 \equiv x$ ˆ 2

end

context *monoid-mult*
begin

subclass *power* ⟨*proof*⟩

lemma *power-one* [*simp*]: 1 ˆ $n = 1$
  ⟨*proof*⟩

lemma *power-one-right* [*simp*]: a ˆ $1 = a$
  ⟨*proof*⟩

lemma *power-Suc0-right* [*simp*]: a ˆ $Suc\ 0 = a$
  ⟨*proof*⟩

lemma *power-commutes*: a ˆ $n * a = a * a$ ˆ n
  ⟨*proof*⟩

lemma *power-Suc2*: a ˆ $Suc\ n = a$ ˆ $n * a$
  ⟨*proof*⟩

lemma *power-add*: a ˆ $(m + n) = a$ ˆ $m * a$ ˆ n
  ⟨*proof*⟩

lemma *power-mult*: a ˆ $(m * n) = (a$ ˆ $m)$ ˆ n
  ⟨*proof*⟩

lemma *power2-eq-square*: $a^2 = a * a$
  ⟨*proof*⟩

lemma *power3-eq-cube*: $a \hat{} 3 = a * a * a$
⟨*proof*⟩

lemma *power-even-eq*: $a \hat{} (2 * n) = (a \hat{} n)^2$
⟨*proof*⟩

lemma *power-odd-eq*: $a \hat{} Suc\ (2*n) = a * (a \hat{} n)^2$
⟨*proof*⟩

lemma *power-numeral-even*: $z \hat{}$ *numeral* (*Num.Bit0 w*) = (*let* $w = z \hat{}$ (*numeral* w) *in* $w * w$)
⟨*proof*⟩

lemma *power-numeral-odd*: $z \hat{}$ *numeral* (*Num.Bit1 w*) = (*let* $w = z \hat{}$ (*numeral* w) *in* $z * w * w$)
⟨*proof*⟩

lemma *funpow-times-power*: (*times* x ^^ $f\ x$) = *times* ($x \hat{} f\ x$)
⟨*proof*⟩

lemma *power-commuting-commutes*:
assumes $x * y = y * x$
shows $x \hat{} n * y = y * x \hat{} n$
⟨*proof*⟩

lemma *power-minus-mult*: $0 < n \Longrightarrow a \hat{} (n - 1) * a = a \hat{} n$
⟨*proof*⟩

end

context *comm-monoid-mult*
begin

lemma *power-mult-distrib* [*field-simps*]: $(a * b) \hat{} n = (a \hat{} n) * (b \hat{} n)$
⟨*proof*⟩

end

Extract constant factors from powers.

declare *power-mult-distrib* [**where** a = *numeral* w **for** w, *simp*]
declare *power-mult-distrib* [**where** b = *numeral* w **for** w, *simp*]

lemma *power-add-numeral* [*simp*]: a^*numeral* m * a^*numeral* n = a^*numeral* (m + n)
for a :: '*a*::*monoid-mult*
⟨*proof*⟩

lemma *power-add-numeral2* [*simp*]: a^*numeral* m * (a^*numeral* n * b) = a^*numeral*

$(m + n) * b$
for a :: $'a$::*monoid-mult*
⟨*proof*⟩

lemma *power-mult-numeral* [*simp*]: $(a\,\hat{}\,numeral\ m)\,\hat{}\,numeral\ n = a\,\hat{}\,numeral\ (m * n)$
for a :: $'a$::*monoid-mult*
⟨*proof*⟩

context *semiring-numeral*
begin

lemma *numeral-sqr*: $numeral\ (Num.sqr\ k) = numeral\ k * numeral\ k$
⟨*proof*⟩

lemma *numeral-pow*: $numeral\ (Num.pow\ k\ l) = numeral\ k\ \hat{}\ numeral\ l$
⟨*proof*⟩

lemma *power-numeral* [*simp*]: $numeral\ k\ \hat{}\ numeral\ l = numeral\ (Num.pow\ k\ l)$
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *of-nat-power* [*simp*]: $of\text{-}nat\ (m\ \hat{}\ n) = of\text{-}nat\ m\ \hat{}\ n$
⟨*proof*⟩

lemma *zero-power*: $0 < n \Longrightarrow 0\ \hat{}\ n = 0$
⟨*proof*⟩

lemma *power-zero-numeral* [*simp*]: $0\ \hat{}\ numeral\ k = 0$
⟨*proof*⟩

lemma *zero-power2*: $0^2 = 0$
⟨*proof*⟩

lemma *one-power2*: $1^2 = 1$
⟨*proof*⟩

lemma *power-0-Suc* [*simp*]: $0\ \hat{}\ Suc\ n = 0$
⟨*proof*⟩

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: $0\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ 0)$
⟨*proof*⟩

end

context *comm-semiring-1*
begin

The divides relation.

lemma *le-imp-power-dvd*:
assumes $m \leq n$
shows $a \ \hat{}\ m \ dvd \ a \ \hat{}\ n$
⟨*proof*⟩

lemma *power-le-dvd*: $a \ \hat{}\ n \ dvd \ b \Longrightarrow m \leq n \Longrightarrow a \ \hat{}\ m \ dvd \ b$
⟨*proof*⟩

lemma *dvd-power-same*: $x \ dvd \ y \Longrightarrow x \ \hat{}\ n \ dvd \ y \ \hat{}\ n$
⟨*proof*⟩

lemma *dvd-power-le*: $x \ dvd \ y \Longrightarrow m \geq n \Longrightarrow x \ \hat{}\ n \ dvd \ y \ \hat{}\ m$
⟨*proof*⟩

lemma *dvd-power* [*simp*]:
fixes $n :: nat$
assumes $n > 0 \vee x = 1$
shows $x \ dvd \ (x \ \hat{}\ n)$
⟨*proof*⟩

end

context *semiring-1-no-zero-divisors*
begin

subclass *power* ⟨*proof*⟩

lemma *power-eq-0-iff* [*simp*]: $a \ \hat{}\ n = 0 \longleftrightarrow a = 0 \wedge n > 0$
⟨*proof*⟩

lemma *power-not-zero*: $a \neq 0 \Longrightarrow a \ \hat{}\ n \neq 0$
⟨*proof*⟩

lemma *zero-eq-power2* [*simp*]: $a^2 = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

end

context *ring-1*
begin

lemma *power-minus*: $(- \ a) \ \hat{}\ n = (- \ 1) \ \hat{}\ n * a \ \hat{}\ n$
⟨*proof*⟩

lemma *power-minus'*: $NO\text{-}MATCH\ 1\ x \Longrightarrow (-x)\ \hat{}\ n = (-1)\hat{}n * x\ \hat{}\ n$
⟨*proof*⟩

lemma *power-minus-Bit0*: $(-\ x)\ \hat{}\ numeral\ (Num.Bit0\ k) = x\ \hat{}\ numeral\ (Num.Bit0\ k)$
⟨*proof*⟩

lemma *power-minus-Bit1*: $(-\ x)\ \hat{}\ numeral\ (Num.Bit1\ k) = -\ (x\ \hat{}\ numeral\ (Num.Bit1\ k))$
⟨*proof*⟩

lemma *power2-minus* [*simp*]: $(-\ a)^2 = a^2$
⟨*proof*⟩

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 *power2-eq-1-iff*: $a^2 = 1 \longleftrightarrow a = 1 \lor a = -\ 1$
⟨*proof*⟩

end

context *idom*
begin

lemma *power2-eq-iff*: $x^2 = y^2 \longleftrightarrow x = y \lor x = -\ y$
⟨*proof*⟩

end

context *algebraic-semidom*
begin

lemma *div-power*: $b\ dvd\ a \Longrightarrow (a\ div\ b)\ \hat{}\ n = a\ \hat{}\ n\ div\ b\ \hat{}\ n$
⟨*proof*⟩

lemma *is-unit-power-iff*: $is\text{-}unit\ (a\ \hat{}\ n) \longleftrightarrow is\text{-}unit\ a \lor n = 0$
⟨*proof*⟩

lemma *dvd-power-iff*:
assumes $x \neq 0$
shows $x \mathbin{\hat{}} m$ *dvd* $x \mathbin{\hat{}} n \longleftrightarrow$ *is-unit* $x \vee m \leq n$
⟨*proof*⟩

end

context *normalization-semidom*
begin

lemma *normalize-power*: *normalize* $(a \mathbin{\hat{}} n) =$ *normalize* $a \mathbin{\hat{}} n$
⟨*proof*⟩

lemma *unit-factor-power*: *unit-factor* $(a \mathbin{\hat{}} n) =$ *unit-factor* $a \mathbin{\hat{}} n$
⟨*proof*⟩

end

context *division-ring*
begin

Perhaps these should be simprules.

lemma *power-inverse* [*field-simps*, *divide-simps*]: *inverse* $a \mathbin{\hat{}} n =$ *inverse* $(a \mathbin{\hat{}} n)$
⟨*proof*⟩

lemma *power-one-over* [*field-simps*, *divide-simps*]: $(1 \;/\; a) \mathbin{\hat{}} n = 1 \;/\; a \mathbin{\hat{}} n$
⟨*proof*⟩

end

context *field*
begin

lemma *power-diff*:
assumes $a \neq 0$
shows $n \leq m \Longrightarrow a \mathbin{\hat{}} (m - n) = a \mathbin{\hat{}} m \;/\; a \mathbin{\hat{}} n$
⟨*proof*⟩

lemma *power-divide* [*field-simps*, *divide-simps*]: $(a \;/\; b) \mathbin{\hat{}} n = a \mathbin{\hat{}} n \;/\; b \mathbin{\hat{}} n$
⟨*proof*⟩

end

42.2 Exponentiation on ordered types

context *linordered-semidom*
begin

lemma *zero-less-power* [*simp*]: $0 < a \Longrightarrow 0 < a \hat{} n$
⟨*proof*⟩

lemma *zero-le-power* [*simp*]: $0 \leq a \Longrightarrow 0 \leq a \hat{} n$
⟨*proof*⟩

lemma *power-mono*: $a \leq b \Longrightarrow 0 \leq a \Longrightarrow a \hat{} n \leq b \hat{} n$
⟨*proof*⟩

lemma *one-le-power* [*simp*]: $1 \leq a \Longrightarrow 1 \leq a \hat{} n$
⟨*proof*⟩

lemma *power-le-one*: $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a \hat{} n \leq 1$
⟨*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*⟩

lemma *of-nat-zero-less-power-iff* [*simp*]: $of\text{-}nat\ x \hat{} n > 0 \longleftrightarrow x > 0 \vee n = 0$
⟨*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-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*]: $n < N \Longrightarrow 0 < a \Longrightarrow a < 1 \longrightarrow a$ ^ $N < a$ ^ n
⟨*proof*⟩

Proof resembles that of *power-strict-decreasing*.

lemma *power-decreasing*: $n \leq N \Longrightarrow 0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a$ ^ $N \leq a$ ^ n
⟨*proof*⟩

lemma *power-Suc-less-one*: $0 < a \Longrightarrow a < 1 \Longrightarrow a$ ^ $Suc\ n < 1$
⟨*proof*⟩

Proof again resembles that of *power-strict-decreasing*.

lemma *power-increasing*: $n \leq N \Longrightarrow 1 \leq a \Longrightarrow a$ ^ $n \leq a$ ^ N
⟨*proof*⟩

Lemma for *power-strict-increasing*.

lemma *power-less-power-Suc*: $1 < a \Longrightarrow a$ ^ $n < a * a$ ^ n
⟨*proof*⟩

lemma *power-strict-increasing*: $n < N \Longrightarrow 1 < a \Longrightarrow a$ ^ $n < a$ ^ N
⟨*proof*⟩

lemma *power-increasing-iff* [*simp*]: $1 < b \Longrightarrow b$ ^ $x \leq b$ ^ $y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *power-strict-increasing-iff* [*simp*]: $1 < b \Longrightarrow b$ ^ $x < b$ ^ $y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *power-le-imp-le-base*:
assumes *le*: a ^ $Suc\ n \leq b$ ^ $Suc\ n$
and $0 \leq b$
shows $a \leq b$
⟨*proof*⟩

lemma *power-less-imp-less-base*:
assumes *less*: a ^ $n < b$ ^ n
assumes *nonneg*: $0 \leq b$
shows $a < b$
⟨*proof*⟩

lemma *power-inject-base*: a ^ $Suc\ n = b$ ^ $Suc\ n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a = b$
⟨*proof*⟩

lemma *power-eq-imp-eq-base*: a ^ $n = b$ ^ $n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < n \Longrightarrow a = b$
⟨*proof*⟩

lemma *power-eq-iff-eq-base*: $0 < n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a \hat{~} n = b \hat{~} n \longleftrightarrow a = b$
⟨*proof*⟩

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*⟩

lemma *power-Suc-le-self*: $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a \hat{~} Suc~n \leq a$
⟨*proof*⟩

lemma *power2-eq-iff-nonneg* [*simp*]:
assumes $0 \leq x$ $0 \leq y$
shows $(x \hat{~} 2 = y \hat{~} 2) \longleftrightarrow x = y$
⟨*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-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 *power-abs*: $|a\ \hat{}\ n| = |a|\ \hat{}\ n$ — FIXME simp?
⟨*proof*⟩

lemma *power-sgn* [*simp*]: $sgn\ (a\ \hat{}\ n) = sgn\ a\ \hat{}\ n$
⟨*proof*⟩

lemma *abs-power-minus* [*simp*]: $|(-\ a)\ \hat{}\ n| = |a\ \hat{}\ n|$
⟨*proof*⟩

lemma *zero-less-power-abs-iff* [*simp*]: $0 < |a|\ \hat{}\ n \longleftrightarrow a \neq 0 \vee n = 0$
⟨*proof*⟩

lemma *zero-le-power-abs* [*simp*]: $0 \leq |a|\ \hat{}\ n$
⟨*proof*⟩

lemma *power2-less-eq-zero-iff* [*simp*]: $a^2 \leq 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *abs-power2* [*simp*]: $|a^2| = a^2$
⟨*proof*⟩

lemma *power2-abs* [*simp*]: $|a|^2 = a^2$
⟨*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*⟩

lemma *abs-le-square-iff*: $|x| \leq |y| \longleftrightarrow x^2 \leq y^2$

(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *abs-square-le-1*:$x^2 \leq 1 \longleftrightarrow |x| \leq 1$
⟨*proof*⟩

lemma *abs-square-eq-1*: $x^2 = 1 \longleftrightarrow |x| = 1$
⟨*proof*⟩

lemma *abs-square-less-1*: $x^2 < 1 \longleftrightarrow |x| < 1$
⟨*proof*⟩

end

42.3 Miscellaneous rules

lemma (**in** *linordered-semidom*) *self-le-power*: $1 \leq a \Longrightarrow 0 < n \Longrightarrow a \leq a \mathbin{\hat{}} n$
⟨*proof*⟩

lemma (**in** *power*) *power-eq-if*: $p \mathbin{\hat{}} m = (\mathit{if}\ m{=}0\ \mathit{then}\ 1\ \mathit{else}\ p * (p \mathbin{\hat{}} (m - 1)))$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *power2-sum*: $(x + y)^2 = x^2 + y^2 + 2 * x * y$
⟨*proof*⟩

context *comm-ring-1*
begin

lemma *power2-diff*: $(x - y)^2 = x^2 + y^2 - 2 * x * y$
⟨*proof*⟩

lemma *power2-commute*: $(x - y)^2 = (y - x)^2$
⟨*proof*⟩

lemma *minus-power-mult-self*: $(- a) \mathbin{\hat{}} n * (- a) \mathbin{\hat{}} n = a \mathbin{\hat{}} (2 * n)$
⟨*proof*⟩

lemma *minus-one-mult-self* [*simp*]: $(- 1) \mathbin{\hat{}} n * (- 1) \mathbin{\hat{}} n = 1$
⟨*proof*⟩

lemma *left-minus-one-mult-self* [*simp*]: $(- 1) \mathbin{\hat{}} n * ((- 1) \mathbin{\hat{}} n * a) = a$
⟨*proof*⟩

end

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

42.4 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*]: $x\ \hat{}\ n > 0 \longleftrightarrow x > 0 \lor n = 0$
for x :: *nat*
⟨*proof*⟩

lemma *nat-power-eq-Suc-0-iff* [*simp*]: $x\ \hat{}\ m = Suc\ 0 \longleftrightarrow m = 0 \lor x = Suc\ 0$
⟨*proof*⟩

lemma *power-Suc-0* [*simp*]: $Suc\ 0\ \hat{}\ n = Suc\ 0$
⟨*proof*⟩

Valid for the naturals, but what if $0 < i < 1$? Premises cannot be weakened: consider the case where $i = 0$, $m = 1$ and $n = 0$.

lemma *nat-power-less-imp-less*:
fixes i :: *nat*
assumes *nonneg*: $0 < i$
assumes *less*: $i\ \hat{}\ m < i\ \hat{}\ n$
shows $m < n$
⟨*proof*⟩

lemma *power-dvd-imp-le*: $i\ \hat{}\ m\ dvd\ i\ \hat{}\ n \Longrightarrow 1 < i \Longrightarrow m \leq n$
for $i\ m\ n$ :: *nat*
⟨*proof*⟩

lemma *power2-nat-le-eq-le*: $m^2 \leq n^2 \longleftrightarrow m \leq n$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *power2-nat-le-imp-le*:
fixes $m\ n$ :: *nat*
assumes $m^2 \leq n$
shows $m \leq n$
⟨*proof*⟩

lemma *ex-power-ivl1*: **fixes** $b\ k$ :: *nat* **assumes** $b \geq 2$
shows $k \geq 1 \Longrightarrow \exists n.\ b\hat{}n \leq k \land k < b\hat{}(n+1)$ (**is** - $\Longrightarrow \exists n.\ ?P\ k\ n$)
⟨*proof*⟩

lemma *ex-power-ivl2*: **fixes** $b\ k$ :: *nat* **assumes** $b \geq 2\ k \geq 2$
shows $\exists n.\ b\hat{}n < k \land k \leq b\hat{}(n+1)$

⟨*proof*⟩

42.4.1 Cardinality of the Powerset

lemma *card-UNIV-bool* [*simp*]: $card\ (UNIV :: bool\ set) = 2$
⟨*proof*⟩

lemma *card-Pow*: $finite\ A \Longrightarrow card\ (Pow\ A) = 2\ \hat{}\ card\ A$
⟨*proof*⟩

42.5 Code generator tweak

code-identifier
code-module *Power* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

43 Big sum and product over finite (non-empty) sets

theory *Groups-Big*
imports *Power*
begin

43.1 Generic monoid operation over a set

locale *comm-monoid-set* = *comm-monoid*
begin

interpretation *comp-fun-commute f*
⟨*proof*⟩

interpretation *comp?*: *comp-fun-commute* $f \circ g$
⟨*proof*⟩

definition $F :: ('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'a$
where *eq-fold*: $F\ g\ A = Finite\text{-}Set.fold\ (f \circ g)\ \mathbf{1}\ A$

lemma *infinite* [*simp*]: $\neg\ finite\ A \Longrightarrow F\ g\ A = \mathbf{1}$
⟨*proof*⟩

lemma *empty* [*simp*]: $F\ g\ \{\} = \mathbf{1}$
⟨*proof*⟩

lemma *insert* [*simp*]: $finite\ A \Longrightarrow x \notin A \Longrightarrow F\ g\ (insert\ x\ A) = g\ x * F\ g\ A$
⟨*proof*⟩

lemma *remove*:
assumes *finite* A **and** $x \in A$

shows $F\ g\ A = g\ x * F\ g\ (A - \{x\})$
⟨*proof*⟩

lemma *insert-remove*: *finite* $A \Longrightarrow F\ g\ (\mathit{insert}\ x\ A) = g\ x * F\ g\ (A - \{x\})$
⟨*proof*⟩

lemma *insert-if*: *finite* $A \Longrightarrow F\ g\ (\mathit{insert}\ x\ A) = (\mathit{if}\ x \in A\ \mathit{then}\ F\ g\ A\ \mathit{else}\ g\ x * F\ g\ A)$
⟨*proof*⟩

lemma *neutral*: $\forall\, x{\in}A.\ g\ x = \mathbf{1} \Longrightarrow F\ g\ A = \mathbf{1}$
⟨*proof*⟩

lemma *neutral-const* [*simp*]: $F\ (\lambda\text{-}.\ \mathbf{1})\ A = \mathbf{1}$
⟨*proof*⟩

lemma *union-inter*:
assumes *finite* A **and** *finite* B
shows $F\ g\ (A \cup B) * F\ g\ (A \cap B) = F\ g\ A * F\ g\ B$
— The reversed orientation looks more natural, but LOOPS as a simprule!
⟨*proof*⟩

corollary *union-inter-neutral*:
assumes *finite* A **and** *finite* B
and $\forall\, x \in A \cap B.\ g\ x = \mathbf{1}$
shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
⟨*proof*⟩

corollary *union-disjoint*:
assumes *finite* A **and** *finite* B
assumes $A \cap B = \{\}$
shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
⟨*proof*⟩

lemma *union-diff2*:
assumes *finite* A **and** *finite* B
shows $F\ g\ (A \cup B) = F\ g\ (A - B) * F\ g\ (B - A) * F\ g\ (A \cap B)$
⟨*proof*⟩

lemma *subset-diff*:
assumes $B \subseteq A$ **and** *finite* A
shows $F\ g\ A = F\ g\ (A - B) * F\ g\ B$
⟨*proof*⟩

lemma *setdiff-irrelevant*:
assumes *finite* A
shows $F\ g\ (A - \{x.\ g\ x = z\}) = F\ g\ A$
⟨*proof*⟩

lemma *not-neutral-contains-not-neutral*:
  assumes $F\ g\ A \neq \mathbf{1}$
  obtains a **where** $a \in A$ **and** $g\ a \neq \mathbf{1}$
⟨*proof*⟩

lemma *reindex*:
  assumes *inj-on* $h\ A$
  shows $F\ g\ (h\ `\ A) = F\ (g \circ h)\ A$
⟨*proof*⟩

lemma *cong* [*fundef-cong*]:
  assumes $A = B$
  assumes *g-h*: $\bigwedge x.\ x \in B \Longrightarrow g\ x = h\ x$
  shows $F\ g\ A = F\ h\ B$
  ⟨*proof*⟩

lemma *strong-cong* [*cong*]:
  assumes $A = B\ \bigwedge x.\ x \in B =simp=> g\ x = h\ x$
  shows $F\ (\lambda x.\ g\ x)\ A = F\ (\lambda x.\ h\ x)\ B$
  ⟨*proof*⟩

lemma *reindex-cong*:
  assumes *inj-on* $l\ B$
  assumes $A = l\ `\ B$
  assumes $\bigwedge x.\ x \in B \Longrightarrow g\ (l\ x) = h\ x$
  shows $F\ g\ A = F\ h\ B$
  ⟨*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 $F\ g\ (UNION\ I\ A) = F\ (\lambda x.\ F\ g\ (A\ x))\ I$
  ⟨*proof*⟩

lemma *Union-disjoint*:
  assumes $\forall A{\in}C.$ *finite* $A\ \forall A{\in}C.\ \forall B{\in}C.\ A \neq B \longrightarrow A \cap B = \{\}$
  shows $F\ g\ (\bigcup C) = (F \circ F)\ g\ C$
⟨*proof*⟩

lemma *distrib*: $F\ (\lambda x.\ g\ x * h\ x)\ A = F\ g\ A * F\ h\ A$
  ⟨*proof*⟩

lemma *Sigma*:
  finite $A \Longrightarrow \forall x{\in}A.$ *finite* $(B\ x) \Longrightarrow F\ (\lambda x.\ F\ (g\ x)\ (B\ x))\ A = F$ (*case-prod* g) $(SIGMA\ x{:}A.\ B\ x)$
  ⟨*proof*⟩

lemma *related*:
  assumes *Re*: $R\ \mathbf{1}\ \mathbf{1}$

and *Rop*: $\forall x1\ y1\ x2\ y2.\ R\ x1\ x2 \wedge R\ y1\ y2 \longrightarrow R\ (x1 * y1)\ (x2 * y2)$
and *fin*: *finite* S
and *R-h-g*: $\forall x{\in}S.\ R\ (h\ x)\ (g\ x)$
shows $R\ (F\ h\ S)\ (F\ g\ S)$
⟨*proof*⟩

lemma *mono-neutral-cong-left*:
assumes *finite* T
and $S \subseteq T$
and $\forall i \in T - S.\ h\ i = \mathbf{1}$
and $\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x$
shows $F\ g\ S = F\ h\ T$
⟨*proof*⟩

lemma *mono-neutral-cong-right*:
finite $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x)$
$\Longrightarrow$
$F\ g\ T = F\ h\ S$
⟨*proof*⟩

lemma *mono-neutral-left*: *finite* $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow F\ g$
$S = F\ g\ T$
⟨*proof*⟩

lemma *mono-neutral-right*: *finite* $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow F$
$g\ T = F\ g\ S$
⟨*proof*⟩

lemma *mono-neutral-cong*:
assumes [*simp*]: *finite* T *finite* S
and $*$: $\bigwedge i.\ i \in T - S \Longrightarrow h\ i = \mathbf{1}$ $\bigwedge i.\ i \in S - T \Longrightarrow g\ i = \mathbf{1}$
and *gh*: $\bigwedge x.\ x \in S \cap T \Longrightarrow g\ x = h\ x$
shows $F\ g\ S = F\ h\ T$
⟨*proof*⟩

lemma *reindex-bij-betw*: *bij-betw* $h\ S\ T \Longrightarrow F\ (\lambda x.\ g\ (h\ x))\ S = F\ g\ T$
⟨*proof*⟩

lemma *reindex-bij-witness*:
assumes *witness*:
$\bigwedge a.\ a \in S \Longrightarrow i\ (j\ a) = a$
$\bigwedge a.\ a \in S \Longrightarrow j\ a \in T$
$\bigwedge b.\ b \in T \Longrightarrow j\ (i\ b) = b$
$\bigwedge b.\ b \in T \Longrightarrow i\ b \in S$
assumes *eq*:
$\bigwedge a.\ a \in S \Longrightarrow h\ (j\ a) = g\ a$
shows $F\ g\ S = F\ h\ T$
⟨*proof*⟩

lemma *reindex-bij-betw-not-neutral*:
assumes *fin*: *finite* S' *finite* T'
assumes *bij*: *bij-betw* h $(S - S')$ $(T - T')$
assumes *nn*:
$\bigwedge a.\ a \in S' \Longrightarrow g\ (h\ a) = z$
$\bigwedge b.\ b \in T' \Longrightarrow g\ b = z$
shows $F\ (\lambda x.\ g\ (h\ x))\ S = F\ g\ T$
⟨*proof*⟩

lemma *reindex-nontrivial*:
assumes *finite* A
and *nz*: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow h\ x = h\ y \Longrightarrow g\ (h\ x) = \mathbf{1}$
shows $F\ g\ (h\ `\ A) = F\ (g \circ h)\ A$
⟨*proof*⟩

lemma *reindex-bij-witness-not-neutral*:
assumes *fin*: *finite* S' *finite* T'
assumes *witness*:
$\bigwedge a.\ a \in S - S' \Longrightarrow i\ (j\ a) = a$
$\bigwedge a.\ a \in S - S' \Longrightarrow j\ a \in T - T'$
$\bigwedge b.\ b \in T - T' \Longrightarrow j\ (i\ b) = b$
$\bigwedge b.\ b \in T - T' \Longrightarrow i\ b \in S - S'$
assumes *nn*:
$\bigwedge a.\ a \in S' \Longrightarrow g\ a = z$
$\bigwedge b.\ b \in T' \Longrightarrow h\ b = z$
assumes *eq*:
$\bigwedge a.\ a \in S \Longrightarrow h\ (j\ a) = g\ a$
shows $F\ g\ S = F\ h\ T$
⟨*proof*⟩

lemma *delta* [*simp*]:
assumes *fS*: *finite* S
shows $F\ (\lambda k.\ \mathit{if}\ k = a\ \mathit{then}\ b\ k\ \mathit{else}\ \mathbf{1})\ S = (\mathit{if}\ a \in S\ \mathit{then}\ b\ a\ \mathit{else}\ \mathbf{1})$
⟨*proof*⟩

lemma *delta'* [*simp*]:
assumes *fin*: *finite* S
shows $F\ (\lambda k.\ \mathit{if}\ a = k\ \mathit{then}\ b\ k\ \mathit{else}\ \mathbf{1})\ S = (\mathit{if}\ a \in S\ \mathit{then}\ b\ a\ \mathit{else}\ \mathbf{1})$
⟨*proof*⟩

lemma *If-cases*:
fixes $P :: 'b \Rightarrow \mathit{bool}$ **and** $g\ h :: 'b \Rightarrow 'a$
assumes *fin*: *finite* A
shows $F\ (\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ h\ x\ \mathit{else}\ g\ x)\ A = F\ h\ (A \cap \{x.\ P\ x\}) * F\ g\ (A \cap - \{x.\ P\ x\})$
⟨*proof*⟩

lemma *cartesian-product*: $F\ (\lambda x.\ F\ (g\ x)\ B)\ A = F\ (\mathit{case\text{-}prod}\ g)\ (A \times B)$
⟨*proof*⟩

lemma *inter-restrict*:
  assumes *finite* A
  shows $F\ g\ (A \cap B) = F\ (\lambda x.\ \textit{if}\ x \in B\ \textit{then}\ g\ x\ \textit{else}\ \mathbf{1})\ A$
⟨*proof*⟩

lemma *inter-filter*:
  $\textit{finite}\ A \Longrightarrow F\ g\ \{x \in A.\ P\ x\} = F\ (\lambda x.\ \textit{if}\ P\ x\ \textit{then}\ g\ x\ \textit{else}\ \mathbf{1})\ A$
  ⟨*proof*⟩

lemma *Union-comp*:
  assumes $\forall A \in B.\ \textit{finite}\ A$
    and $\bigwedge A1\ A2\ x.\ A1 \in B \Longrightarrow A2 \in B \Longrightarrow A1 \neq A2 \Longrightarrow x \in A1 \Longrightarrow x \in A2 \Longrightarrow g\ x = \mathbf{1}$
  shows $F\ g\ (\bigcup B) = (F \circ F)\ g\ B$
  ⟨*proof*⟩

lemma *commute*: $F\ (\lambda i.\ F\ (g\ i)\ B)\ A = F\ (\lambda j.\ F\ (\lambda i.\ g\ i\ j)\ A)\ B$
  ⟨*proof*⟩

lemma *commute-restrict*:
  $\textit{finite}\ A \Longrightarrow \textit{finite}\ B \Longrightarrow$
    $F\ (\lambda x.\ F\ (g\ x)\ \{y.\ y \in B \wedge R\ x\ y\})\ A = F\ (\lambda y.\ F\ (\lambda x.\ g\ x\ y)\ \{x.\ x \in A \wedge R\ x\ y\})\ B$
  ⟨*proof*⟩

lemma *Plus*:
  fixes $A :: 'b\ \textit{set}$ **and** $B :: 'c\ \textit{set}$
  assumes *fin*: *finite* A *finite* B
  shows $F\ g\ (A <+> B) = F\ (g \circ \textit{Inl})\ A * F\ (g \circ \textit{Inr})\ B$
⟨*proof*⟩

lemma *same-carrier*:
  assumes *finite* C
  assumes *subset*: $A \subseteq C$ $B \subseteq C$
  assumes *trivial*: $\bigwedge a.\ a \in C - A \Longrightarrow g\ a = \mathbf{1}$ $\bigwedge b.\ b \in C - B \Longrightarrow h\ b = \mathbf{1}$
  shows $F\ g\ A = F\ h\ B \longleftrightarrow F\ g\ C = F\ h\ C$
⟨*proof*⟩

lemma *same-carrierI*:
  assumes *finite* C
  assumes *subset*: $A \subseteq C$ $B \subseteq C$
  assumes *trivial*: $\bigwedge a.\ a \in C - A \Longrightarrow g\ a = \mathbf{1}$ $\bigwedge b.\ b \in C - B \Longrightarrow h\ b = \mathbf{1}$
  assumes $F\ g\ C = F\ h\ C$
  shows $F\ g\ A = F\ h\ B$
  ⟨*proof*⟩

end

43.2 Generalized summation over a set

context *comm-monoid-add*
begin

sublocale *sum*: *comm-monoid-set plus 0*
  defines *sum* = *sum.F* ⟨*proof*⟩

abbreviation *Sum* ($\sum$- [*1000*] *999*)
  where $\sum A \equiv sum\ (\lambda x.\ x)\ A$

end

Now: lot's of fancy syntax. First, *sum* $(\lambda x.\ e)$ A is written $\sum x{\in}A.\ e$.

syntax (*ASCII*)
  -sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* ((*3SUM* -:-./ -) [*0*, *51*, *10*] *10*)
syntax
  -sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* ((*2*$\sum$-∈-./ -) [*0*, *51*, *10*] *10*)
translations — Beware of argument permutation!
  $\sum i{\in}A.\ b \rightleftharpoons$ *CONST sum* $(\lambda i.\ b)$ A

Instead of $\sum x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\sum x|P.\ e$.

syntax (*ASCII*)
  -qsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3SUM* - |/ -./ -) [*0*, *0*, *10*] *10*)
syntax
  -qsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*2*$\sum$- | (-)./ -) [*0*, *0*, *10*] *10*)
translations
  $\sum x|P.\ t$ => *CONST sum* $(\lambda x.\ t)$ $\{x.\ P\}$

⟨*ML*⟩

lemma (**in** *comm-monoid-add*) *sum-image-gen*:
  assumes *fin*: *finite S*
  shows *sum g S* = *sum* $(\lambda y.\ sum\ g\ \{x.\ x \in S \wedge f\ x = y\})$ $(f\ \text{`}\ S)$
⟨*proof*⟩

43.2.1 Properties in more restricted classes of structures

lemma *sum-Un*:
  finite A ⟹ *finite B* ⟹ *sum f* $(A \cup B)$ = *sum f A* + *sum f B* − *sum f* $(A \cap B)$
  for *f* :: *'b* ⇒ *'a*::*ab-group-add*
  ⟨*proof*⟩

lemma *sum-Un2*:
  assumes *finite* $(A \cup B)$

shows $sum\ f\ (A \cup B) = sum\ f\ (A - B) + sum\ f\ (B - A) + sum\ f\ (A \cap B)$
⟨*proof*⟩

lemma *sum-diff1*:
fixes $f :: 'b \Rightarrow 'a::ab\text{-}group\text{-}add$
assumes *finite A*
shows $sum\ f\ (A - \{a\}) = (if\ a \in A\ then\ sum\ f\ A - f\ a\ else\ sum\ f\ A)$
⟨*proof*⟩

lemma *sum-diff*:
fixes $f :: 'b \Rightarrow 'a::ab\text{-}group\text{-}add$
assumes *finite A* $B \subseteq A$
shows $sum\ f\ (A - B) = sum\ f\ A - sum\ f\ B$
⟨*proof*⟩

lemma (**in** *ordered-comm-monoid-add*) *sum-mono*:
$(\bigwedge i.\ i \in K \Longrightarrow f\ i \le g\ i) \Longrightarrow (\sum i \in K.\ f\ i) \le (\sum i \in K.\ g\ i)$
⟨*proof*⟩

lemma (**in** *strict-ordered-comm-monoid-add*) *sum-strict-mono*:
assumes *finite A* $A \neq \{\}$
and $\bigwedge x.\ x \in A \Longrightarrow f\ x < g\ x$
shows $sum\ f\ A < sum\ g\ A$
⟨*proof*⟩

lemma *sum-strict-mono-ex1*:
fixes $f\ g :: 'i \Rightarrow 'a::ordered\text{-}cancel\text{-}comm\text{-}monoid\text{-}add$
assumes *finite A*
and $\forall x \in A.\ f\ x \le g\ x$
and $\exists a \in A.\ f\ a < g\ a$
shows $sum\ f\ A < sum\ g\ A$
⟨*proof*⟩

lemma *sum-mono-inv*:
fixes $f\ g :: 'i \Rightarrow 'a :: ordered\text{-}cancel\text{-}comm\text{-}monoid\text{-}add$
assumes *eq*: $sum\ f\ I = sum\ g\ I$
assumes *le*: $\bigwedge i.\ i \in I \Longrightarrow f\ i \le g\ i$
assumes *i*: $i \in I$
assumes *I*: *finite I*
shows $f\ i = g\ i$
⟨*proof*⟩

lemma *member-le-sum*:
fixes $f :: \text{-} \Rightarrow 'b::\{semiring\text{-}1,\ ordered\text{-}comm\text{-}monoid\text{-}add\}$
assumes $i \in A$
and *le*: $\bigwedge x.\ x \in A - \{i\} \Longrightarrow 0 \le f\ x$
and *finite A*
shows $f\ i \le sum\ f\ A$
⟨*proof*⟩

lemma *sum-negf*: $(\sum x{\in}A. - f\ x) = - (\sum x{\in}A. f\ x)$
for $f :: 'b \Rightarrow 'a::ab\text{-}group\text{-}add$
⟨*proof*⟩

lemma *sum-subtractf*: $(\sum x{\in}A. f\ x - g\ x) = (\sum x{\in}A. f\ x) - (\sum x{\in}A. g\ x)$
for $f\ g :: 'b \Rightarrow 'a::ab\text{-}group\text{-}add$
⟨*proof*⟩

lemma *sum-subtractf-nat*:
$(\bigwedge x. x \in A \Longrightarrow g\ x \leq f\ x) \Longrightarrow (\sum x{\in}A. f\ x - g\ x) = (\sum x{\in}A. f\ x) - (\sum x{\in}A. g\ x)$
for $f\ g :: 'a \Rightarrow nat$
⟨*proof*⟩

context *ordered-comm-monoid-add*
begin

lemma *sum-nonneg*: $(\bigwedge x. x \in A \Longrightarrow 0 \leq f\ x) \Longrightarrow 0 \leq sum\ f\ A$
⟨*proof*⟩

lemma *sum-nonpos*: $(\bigwedge x. x \in A \Longrightarrow f\ x \leq 0) \Longrightarrow sum\ f\ A \leq 0$
⟨*proof*⟩

lemma *sum-nonneg-eq-0-iff*:
$finite\ A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow 0 \leq f\ x) \Longrightarrow sum\ f\ A = 0 \longleftrightarrow (\forall x{\in}A. f\ x = 0)$
⟨*proof*⟩

lemma *sum-nonneg-0*:
$finite\ s \Longrightarrow (\bigwedge i. i \in s \Longrightarrow f\ i \geq 0) \Longrightarrow (\sum\ i \in s. f\ i) = 0 \Longrightarrow i \in s \Longrightarrow f\ i = 0$
⟨*proof*⟩

lemma *sum-nonneg-leq-bound*:
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 *sum-mono2*:
assumes *fin*: *finite B*
and *sub*: $A \subseteq B$
and *nn*: $\bigwedge b. b \in B{-}A \Longrightarrow 0 \leq f\ b$
shows $sum\ f\ A \leq sum\ f\ B$
⟨*proof*⟩

lemma *sum-le-included*:
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 $sum\ f\ s \leq sum\ g\ t$

⟨*proof*⟩

end

lemma (**in** *canonically-ordered-monoid-add*) *sum-eq-0-iff* [*simp*]:
$finite\ F \Longrightarrow (sum\ f\ F = 0) = (\forall\, a{\in}F.\ f\ a = 0)$
⟨*proof*⟩

lemma *sum-distrib-left*: $r * sum\ f\ A = sum\ (\lambda n.\ r * f\ n)\ A$
for f :: $'a \Rightarrow 'b{::}semiring\text{-}0$
⟨*proof*⟩

lemma *sum-distrib-right*: $sum\ f\ A * r = (\sum n{\in}A.\ f\ n * r)$
for r :: $'a{::}semiring\text{-}0$
⟨*proof*⟩

lemma *sum-divide-distrib*: $sum\ f\ A\ /\ r = (\sum n{\in}A.\ f\ n\ /\ r)$
for r :: $'a{::}field$
⟨*proof*⟩

lemma *sum-abs*[*iff*]: $|sum\ f\ A| \le sum\ (\lambda i.\ |f\ i|)\ A$
for f :: $'a \Rightarrow 'b{::}ordered\text{-}ab\text{-}group\text{-}add\text{-}abs$
⟨*proof*⟩

lemma *sum-abs-ge-zero*[*iff*]: $0 \le sum\ (\lambda i.\ |f\ i|)\ A$
for f :: $'a \Rightarrow 'b{::}ordered\text{-}ab\text{-}group\text{-}add\text{-}abs$
⟨*proof*⟩

lemma *abs-sum-abs*[*simp*]: $|\sum a{\in}A.\ |f\ a|| = (\sum a{\in}A.\ |f\ a|)$
for f :: $'a \Rightarrow 'b{::}ordered\text{-}ab\text{-}group\text{-}add\text{-}abs$
⟨*proof*⟩

lemma *sum-diff1-ring*:
fixes f :: $'b \Rightarrow 'a{::}ring$
assumes *finite* A $a \in A$
shows $sum\ f\ (A - \{a\}) = sum\ f\ A - (f\ a)$
⟨*proof*⟩

lemma *sum-product*:
fixes f :: $'a \Rightarrow 'b{::}semiring\text{-}0$
shows $sum\ f\ A * sum\ g\ B = (\sum i{\in}A.\ \sum j{\in}B.\ f\ i * g\ j)$
⟨*proof*⟩

lemma *sum-mult-sum-if-inj*:
fixes f :: $'a \Rightarrow 'b{::}semiring\text{-}0$
shows $inj\text{-}on\ (\lambda(a,\ b).\ f\ a * g\ b)\ (A \times B) \Longrightarrow$
$sum\ f\ A * sum\ g\ B = sum\ id\ \{f\ a * g\ b\ |a\ b.\ a \in A \wedge b \in B\}$
⟨*proof*⟩

lemma *sum-SucD*: $sum\ f\ A = Suc\ n \Longrightarrow \exists\, a{\in}A.\ 0 < f\ a$
⟨*proof*⟩

lemma *sum-eq-Suc0-iff*:
$finite\ A \Longrightarrow sum\ f\ A = Suc\ 0 \longleftrightarrow (\exists\, a{\in}A.\ f\ a = Suc\ 0 \wedge (\forall\, b{\in}A.\ a \neq b \longrightarrow f\ b = 0))$
⟨*proof*⟩

lemmas *sum-eq-1-iff* = *sum-eq-Suc0-iff*[*simplified One-nat-def*[*symmetric*]]

lemma *sum-Un-nat*:
$finite\ A \Longrightarrow finite\ B \Longrightarrow sum\ f\ (A \cup B) = sum\ f\ A + sum\ f\ B - sum\ f\ (A \cap B)$
for $f :: 'a \Rightarrow nat$
— For the natural numbers, we have subtraction.
⟨*proof*⟩

lemma *sum-diff1-nat*: $sum\ f\ (A - \{a\}) = (if\ a \in A\ then\ sum\ f\ A - f\ a\ else\ sum\ f\ A)$
for $f :: 'a \Rightarrow nat$
⟨*proof*⟩

lemma *sum-diff-nat*:
fixes $f :: 'a \Rightarrow nat$
assumes $finite\ B$ **and** $B \subseteq A$
shows $sum\ f\ (A - B) = sum\ f\ A - sum\ f\ B$
⟨*proof*⟩

lemma *sum-comp-morphism*:
$h\ 0 = 0 \Longrightarrow (\bigwedge x\ y.\ h\ (x + y) = h\ x + h\ y) \Longrightarrow sum\ (h \circ g)\ A = h\ (sum\ g\ A)$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *dvd-sum*: $(\bigwedge a.\ a \in A \Longrightarrow d\ dvd\ f\ a) \Longrightarrow d\ dvd\ sum\ f\ A$
⟨*proof*⟩

lemma (**in** *ordered-comm-monoid-add*) *sum-pos*:
$finite\ I \Longrightarrow I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow 0 < f\ i) \Longrightarrow 0 < sum\ f\ I$
⟨*proof*⟩

lemma (**in** *ordered-comm-monoid-add*) *sum-pos2*:
assumes I: $finite\ I$ $i \in I$ $0 < f\ i$ $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$
shows $0 < sum\ f\ I$
⟨*proof*⟩

lemma *sum-cong-Suc*:
assumes $0 \notin A$ $\bigwedge x.\ Suc\ x \in A \Longrightarrow f\ (Suc\ x) = g\ (Suc\ x)$
shows $sum\ f\ A = sum\ g\ A$
⟨*proof*⟩

43.2.2 Cardinality as special case of *sum*

lemma *card-eq-sum*: $card\ A = sum\ (\lambda x.\ 1)\ A$
⟨*proof*⟩

lemma *sum-constant* [*simp*]: $(\sum x \in A.\ y) = of\text{-}nat\ (card\ A) * y$
⟨*proof*⟩

lemma *sum-Suc*: $sum\ (\lambda x.\ Suc(f\ x))\ A = sum\ f\ A + card\ A$
⟨*proof*⟩

lemma *sum-bounded-above*:
fixes K :: $'a$::{*semiring-1*,*ordered-comm-monoid-add*}
assumes *le*: $\bigwedge i.\ i \in A \Longrightarrow f\ i \leq K$
shows $sum\ f\ A \leq of\text{-}nat\ (card\ A) * K$
⟨*proof*⟩

lemma *sum-bounded-above-strict*:
fixes K :: $'a$::{*ordered-cancel-comm-monoid-add*,*semiring-1*}
assumes $\bigwedge i.\ i \in A \Longrightarrow f\ i < K\ card\ A > 0$
shows $sum\ f\ A < of\text{-}nat\ (card\ A) * K$
⟨*proof*⟩

lemma *sum-bounded-below*:
fixes K :: $'a$::{*semiring-1*,*ordered-comm-monoid-add*}
assumes *le*: $\bigwedge i.\ i \in A \Longrightarrow K \leq f\ i$
shows $of\text{-}nat\ (card\ A) * K \leq sum\ f\ A$
⟨*proof*⟩

lemma *card-UN-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 $card\ (UNION\ I\ A) = (\sum i \in I.\ card(A\ i))$
⟨*proof*⟩

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

lemma *sum-multicount-gen*:
assumes *finite* s *finite* t $\forall j \in t.\ (card\ \{i \in s.\ R\ i\ j\} = k\ j)$
shows $sum\ (\lambda i.\ (card\ \{j \in t.\ R\ i\ j\}))\ s = sum\ k\ t$
(**is** $?l = ?r$)
⟨*proof*⟩

lemma *sum-multicount*:
assumes *finite* S *finite* T $\forall j \in T.\ (card\ \{i \in S.\ R\ i\ j\} = k)$
shows $sum\ (\lambda i.\ card\ \{j \in T.\ R\ i\ j\})\ S = k * card\ T$ (**is** $?l = ?r$)
⟨*proof*⟩

43.2.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
finite A $\Longrightarrow \forall\, a{\in}A.$ *finite* $(B\ a) \Longrightarrow$ *card* (*SIGMA* x: A. $B\ x$) = ($\sum a{\in}A.$ *card* $(B\ a)$)
⟨*proof*⟩

lemma *card-cartesian-product*: *card* $(A \times B)$ = *card* $A *$ *card* B
⟨*proof*⟩

lemma *card-cartesian-product-singleton*: *card* $(\{x\} \times A)$ = *card* A
⟨*proof*⟩

43.3 Generalized product over a set

context *comm-monoid-mult*
begin

sublocale *prod*: *comm-monoid-set times 1*
defines *prod* = *prod.F* ⟨*proof*⟩

abbreviation *Prod* ($\prod$ - [*1000*] *999*)
where $\prod A \equiv$ *prod* $(\lambda x.\ x)\ A$

end

syntax (*ASCII*)
-prod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*4PROD* -:-./ -) [*0*, *51*, *10*] *10*)
syntax
-prod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*2*$\prod$ -∈-./ -) [*0*, *51*, *10*] *10*)
translations — Beware of argument permutation!
$\prod i{\in}A.\ b$ == *CONST prod* $(\lambda i.\ b)\ A$

Instead of $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax (*ASCII*)
-qprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*4PROD* - |/ -./ -) [*0*, *0*, *10*] *10*)
syntax
-qprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*2*$\prod$ - | (-)./ -) [*0*, *0*, *10*] *10*)
translations
$\prod x|P.\ t$ => *CONST prod* $(\lambda x.\ t)$ $\{x.\ P\}$

context *comm-monoid-mult*
begin

lemma *prod-dvd-prod*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a$ *dvd* $g\ a) \Longrightarrow$ *prod f A dvd prod g A*
⟨*proof*⟩

lemma *prod-dvd-prod-subset*: *finite* $B \Longrightarrow A \subseteq B \Longrightarrow$ *prod f A dvd prod f B*
⟨*proof*⟩

end

43.3.1 Properties in more restricted classes of structures

context *linordered-nonzero-semiring*
begin

lemma *prod-ge-1*: $(\bigwedge x.\ x \in A \Longrightarrow 1 \leq f\ x) \Longrightarrow 1 \leq$ *prod f A*
⟨*proof*⟩

lemma *prod-le-1*:
fixes $f :: 'b \Rightarrow 'a$
assumes $\bigwedge x.\ x \in A \Longrightarrow 0 \leq f\ x \wedge f\ x \leq 1$
shows *prod f A* ≤ 1
⟨*proof*⟩

end

context *comm-semiring-1*
begin

lemma *dvd-prod-eqI* [*intro*]:
assumes *finite A* **and** $a \in A$ **and** $b = f\ a$
shows *b dvd prod f A*
⟨*proof*⟩

lemma *dvd-prodI* [*intro*]: *finite* $A \Longrightarrow a \in A \Longrightarrow$ *f a dvd prod f A*
⟨*proof*⟩

lemma *prod-zero*:
assumes *finite A* **and** $\exists\, a \in A.\ f\ a = 0$
shows *prod f A* $= 0$
⟨*proof*⟩

lemma *prod-dvd-prod-subset2*:
assumes *finite B* **and** $A \subseteq B$ **and** $\bigwedge a.\ a \in A \Longrightarrow$ *f a dvd g a*
shows *prod f A dvd prod g B*
⟨*proof*⟩

end

lemma (**in** *semidom*) *prod-zero-iff* [*simp*]:
fixes $f :: 'b \Rightarrow 'a$
assumes *finite A*
shows *prod f A* $= 0 \longleftrightarrow (\exists\, a \in A.\ f\ a = 0)$

⟨*proof*⟩

lemma (**in** *semidom-divide*) *prod-diff1*:
assumes *finite A* **and** $f\ a \neq 0$
shows $prod\ f\ (A - \{a\}) = (if\ a \in A\ then\ prod\ f\ A\ div\ f\ a\ else\ prod\ f\ A)$
⟨*proof*⟩

lemma *sum-zero-power* [*simp*]: $(\sum i{\in}A.\ c\ i * 0\hat{~}i) = (if\ finite\ A \wedge 0 \in A\ then\ c\ 0\ else\ 0)$
for $c :: nat \Rightarrow 'a::division\text{-}ring$
⟨*proof*⟩

lemma *sum-zero-power′* [*simp*]:
$(\sum i{\in}A.\ c\ i * 0\hat{~}i\ /\ d\ i) = (if\ finite\ A \wedge 0 \in A\ then\ c\ 0\ /\ d\ 0\ else\ 0)$
for $c :: nat \Rightarrow 'a::field$
⟨*proof*⟩

lemma (**in** *field*) *prod-inversef*: $prod\ (inverse \circ f)\ A = inverse\ (prod\ f\ A)$
⟨*proof*⟩

lemma (**in** *field*) *prod-dividef*: $(\prod x{\in}A.\ f\ x\ /\ g\ x) = prod\ f\ A\ /\ prod\ g\ A$
⟨*proof*⟩

lemma *prod-Un*:
fixes $f :: 'b \Rightarrow 'a :: field$
assumes *finite A* **and** *finite B*
and $\forall x{\in}A \cap B.\ f\ x \neq 0$
shows $prod\ f\ (A \cup B) = prod\ f\ A * prod\ f\ B\ /\ prod\ f\ (A \cap B)$
⟨*proof*⟩

lemma (**in** *linordered-semidom*) *prod-nonneg*: $(\forall a{\in}A.\ 0 \leq f\ a) \Longrightarrow 0 \leq prod\ f\ A$
⟨*proof*⟩

lemma (**in** *linordered-semidom*) *prod-pos*: $(\forall a{\in}A.\ 0 < f\ a) \Longrightarrow 0 < prod\ f\ A$
⟨*proof*⟩

lemma (**in** *linordered-semidom*) *prod-mono*:
$\forall i{\in}A.\ 0 \leq f\ i \wedge f\ i \leq g\ i \Longrightarrow prod\ f\ A \leq prod\ g\ A$
⟨*proof*⟩

lemma (**in** *linordered-semidom*) *prod-mono-strict*:
assumes $finite\ A\ \forall i{\in}A.\ 0 \leq f\ i \wedge f\ i < g\ i\ A \neq \{\}$
shows $prod\ f\ A < prod\ g\ A$
⟨*proof*⟩

lemma (**in** *linordered-field*) *abs-prod*: $|prod\ f\ A| = (\prod x{\in}A.\ |f\ x|)$
⟨*proof*⟩

lemma *prod-eq-1-iff* [*simp*]: $finite\ A \Longrightarrow prod\ f\ A = 1 \longleftrightarrow (\forall a{\in}A.\ f\ a = 1)$

for f :: $'a \Rightarrow nat$
⟨*proof*⟩

lemma *prod-pos-nat-iff* [*simp*]: $finite\ A \Longrightarrow prod\ f\ A > 0 \longleftrightarrow (\forall\, a{\in}A.\ f\ a > 0)$
for f :: $'a \Rightarrow nat$
⟨*proof*⟩

lemma *prod-constant*: $(\prod x{\in}\ A.\ y) = y\ \hat{}\ card\ A$
for y :: $'a{::}comm\text{-}monoid\text{-}mult$
⟨*proof*⟩

lemma *prod-power-distrib*: $prod\ f\ A\ \hat{}\ n = prod\ (\lambda x.\ (f\ x)\ \hat{}\ n)\ A$
for f :: $'a \Rightarrow 'b{::}comm\text{-}semiring\text{-}1$
⟨*proof*⟩

lemma *power-sum*: $c\ \hat{}\ (\sum a{\in}A.\ f\ a) = (\prod a{\in}A.\ c\ \hat{}\ f\ a)$
⟨*proof*⟩

lemma *prod-gen-delta*:
fixes b :: $'b \Rightarrow 'a{::}comm\text{-}monoid\text{-}mult$
assumes *fin*: *finite S*
shows $prod\ (\lambda k.\ if\ k = a\ then\ b\ k\ else\ c)\ S =$
$(if\ a \in S\ then\ b\ a * c\ \hat{}\ (card\ S - 1)\ else\ c\ \hat{}\ card\ S)$
⟨*proof*⟩

lemma *sum-image-le*:
fixes g :: $'a \Rightarrow 'b{::}ordered\text{-}ab\text{-}group\text{-}add$
assumes $finite\ I\ \bigwedge i.\ i \in I \Longrightarrow 0 \le g(f\ i)$
shows $sum\ g\ (f\ `\ I) \le sum\ (g \circ f)\ I$
⟨*proof*⟩

end

44 Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *Groups-Big*
begin

44.1 Equivalence relations – set version

definition $equiv$ :: $'a\ set \Rightarrow ('a \times 'a)\ set \Rightarrow bool$
where $equiv\ A\ r \longleftrightarrow refl\text{-}on\ A\ r \wedge sym\ r \wedge trans\ r$

lemma *equivI*: $refl\text{-}on\ A\ r \Longrightarrow sym\ r \Longrightarrow trans\ r \Longrightarrow equiv\ A\ r$
⟨*proof*⟩

lemma *equivE*:
assumes *equiv A r*
obtains *refl-on A r* **and** *sym r* **and** *trans r*
⟨*proof*⟩

Suppes, Theorem 70: r is an equiv relation iff $r^{-1}\ O\ r = r$.
First half: $equiv\ A\ r \Longrightarrow r^{-1}\ O\ r = r$.

lemma *sym-trans-comp-subset*: $sym\ r \Longrightarrow trans\ r \Longrightarrow r^{-1}\ O\ r \subseteq r$
⟨*proof*⟩

lemma *refl-on-comp-subset*: $refl\text{-}on\ A\ r \Longrightarrow r \subseteq r^{-1}\ O\ r$
⟨*proof*⟩

lemma *equiv-comp-eq*: $equiv\ A\ r \Longrightarrow r^{-1}\ O\ r = r$
⟨*proof*⟩

Second half.

lemma *comp-equivI*: $r^{-1}\ O\ r = r \Longrightarrow Domain\ r = A \Longrightarrow equiv\ A\ r$
⟨*proof*⟩

44.2 Equivalence classes

lemma *equiv-class-subset*: $equiv\ A\ r \Longrightarrow (a,\ b) \in r \Longrightarrow r``\{a\} \subseteq r``\{b\}$
— lemma for the next result
⟨*proof*⟩

theorem *equiv-class-eq*: $equiv\ A\ r \Longrightarrow (a,\ b) \in r \Longrightarrow r``\{a\} = r``\{b\}$
⟨*proof*⟩

lemma *equiv-class-self*: $equiv\ A\ r \Longrightarrow a \in A \Longrightarrow a \in r``\{a\}$
⟨*proof*⟩

lemma *subset-equiv-class*: $equiv\ A\ r \Longrightarrow r``\{b\} \subseteq r``\{a\} \Longrightarrow b \in A \Longrightarrow (a,\ b) \in r$
— lemma for the next result
⟨*proof*⟩

lemma *eq-equiv-class*: $r``\{a\} = r``\{b\} \Longrightarrow equiv\ A\ r \Longrightarrow b \in A \Longrightarrow (a,\ b) \in r$
⟨*proof*⟩

lemma *equiv-class-nondisjoint*: $equiv\ A\ r \Longrightarrow x \in (r``\{a\} \cap r``\{b\}) \Longrightarrow (a,\ b) \in r$
⟨*proof*⟩

lemma *equiv-type*: $equiv\ A\ r \Longrightarrow r \subseteq A \times A$
⟨*proof*⟩

lemma *equiv-class-eq-iff*: $equiv\ A\ r \Longrightarrow (x,\ y) \in r \longleftrightarrow r``\{x\} = r``\{y\} \wedge x \in A \wedge y \in A$

⟨*proof*⟩

lemma *eq-equiv-class-iff*: $equiv\ A\ r \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow r``\{x\} = r``\{y\} \longleftrightarrow (x, y) \in r$
⟨*proof*⟩

44.3 Quotients

definition *quotient* :: $'a\ set \Rightarrow ('a \times 'a)\ set \Rightarrow 'a\ set\ set$ (**infixl** *'//' 90*)
where $A//r = (\bigcup x \in A.\ \{r``\{x\}\})$ — set of equiv classes

lemma *quotientI*: $x \in A ==> r``\{x\} \in A//r$
⟨*proof*⟩

lemma *quotientE*: $X \in A//r \Longrightarrow (\bigwedge x.\ X = r``\{x\} \Longrightarrow x \in A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Union-quotient*: $equiv\ A\ r \Longrightarrow \bigcup(A//r) = A$
⟨*proof*⟩

lemma *quotient-disj*: $equiv\ A\ r \Longrightarrow X \in A//r \Longrightarrow Y \in A//r \Longrightarrow X = Y \lor X \cap Y = \{\}$
⟨*proof*⟩

lemma *quotient-eqI*:
$equiv\ A\ r \Longrightarrow X \in A//r \Longrightarrow Y \in A//r \Longrightarrow x \in X \Longrightarrow y \in Y \Longrightarrow (x, y) \in r \Longrightarrow X = Y$
⟨*proof*⟩

lemma *quotient-eq-iff*:
$equiv\ A\ r \Longrightarrow X \in A//r \Longrightarrow Y \in A//r \Longrightarrow x \in X \Longrightarrow y \in Y \Longrightarrow X = Y \longleftrightarrow (x, y) \in r$
⟨*proof*⟩

lemma *eq-equiv-class-iff2*: $equiv\ A\ r \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow \{x\}//r = \{y\}//r \longleftrightarrow (x, y) \in r$
⟨*proof*⟩

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
⟨*proof*⟩

lemma *quotient-is-empty* [*iff*]: $A//r = \{\} \longleftrightarrow A = \{\}$
⟨*proof*⟩

lemma *quotient-is-empty2* [*iff*]: $\{\} = A//r \longleftrightarrow A = \{\}$
⟨*proof*⟩

lemma *singleton-quotient*: $\{x\}//r = \{r\ ``\ \{x\}\}$
⟨*proof*⟩

lemma *quotient-diff1*: $inj\text{-}on\ (\lambda a.\ \{a\}//r)\ A \Longrightarrow a \in A \Longrightarrow (A - \{a\})//r = A//r - \{a\}//r$
⟨*proof*⟩

44.4 Refinement of one equivalence relation WRT another

lemma *refines-equiv-class-eq*: $R \subseteq S \Longrightarrow equiv\ A\ R \Longrightarrow equiv\ A\ S \Longrightarrow R``(S``\{a\}) = S``\{a\}$
⟨*proof*⟩

lemma *refines-equiv-class-eq2*: $R \subseteq S \Longrightarrow equiv\ A\ R \Longrightarrow equiv\ A\ S \Longrightarrow S``(R``\{a\}) = S``\{a\}$
⟨*proof*⟩

lemma *refines-equiv-image-eq*: $R \subseteq S \Longrightarrow equiv\ A\ R \Longrightarrow equiv\ A\ S \Longrightarrow (\lambda X.\ S``X)\ `\ (A//R) = A//S$
⟨*proof*⟩

lemma *finite-refines-finite*:
$finite\ (A//R) \Longrightarrow R \subseteq S \Longrightarrow equiv\ A\ R \Longrightarrow equiv\ A\ S \Longrightarrow finite\ (A//S)$
⟨*proof*⟩

lemma *finite-refines-card-le*:
$finite\ (A//R) \Longrightarrow R \subseteq S \Longrightarrow equiv\ A\ R \Longrightarrow equiv\ A\ S \Longrightarrow card\ (A//S) \le card\ (A//R)$
⟨*proof*⟩

44.5 Defining unary operations upon equivalence classes

A congruence-preserving function.

definition *congruent* :: $('a \times 'a)\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where $congruent\ r\ f \longleftrightarrow (\forall (y, z) \in r.\ f\ y = f\ z)$

lemma *congruentI*: $(\bigwedge y\ z.\ (y, z) \in r \Longrightarrow f\ y = f\ z) \Longrightarrow congruent\ r\ f$
⟨*proof*⟩

lemma *congruentD*: $congruent\ r\ f \Longrightarrow (y, z) \in r \Longrightarrow f\ y = f\ z$
⟨*proof*⟩

abbreviation *RESPECTS* :: $('a \Rightarrow 'b) \Rightarrow ('a \times 'a)\ set \Rightarrow bool$ (**infixr** *respects* *80*)
where $f\ respects\ r \equiv congruent\ r\ f$

lemma *UN-constant-eq*: $a \in A \Longrightarrow \forall\, y \in A.\ f\ y = c \Longrightarrow (\bigcup y \in A.\ f\ y) = c$
— lemma required to prove *UN-equiv-class*
⟨*proof*⟩

lemma *UN-equiv-class*: $equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow a \in A \Longrightarrow (\bigcup x \in r``\{a\}.\ f\ x) = f\ a$
— Conversion rule
⟨*proof*⟩

lemma *UN-equiv-class-type*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow X \in A//r \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\bigcup x \in X.\ f\ x) \in B$
⟨*proof*⟩

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; *bcong* could be $\bigwedge y.\ y \in A \Longrightarrow f\ y \in B$.

lemma *UN-equiv-class-inject*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow$
$(\bigcup x \in X.\ f\ x) = (\bigcup y \in Y.\ f\ y) \Longrightarrow X \in A//r ==> Y \in A//r$
$\Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow (x,\ y) \in r)$
$\Longrightarrow X = Y$
⟨*proof*⟩

44.6 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments.

definition *congruent2* :: $('a \times 'a)\ set \Rightarrow ('b \times 'b)\ set \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow bool$
where $congruent2\ r1\ r2\ f \longleftrightarrow (\forall (y1,\ z1) \in r1.\ \forall (y2,\ z2) \in r2.\ f\ y1\ y2 = f\ z1\ z2)$

lemma *congruent2I'*:
assumes $\bigwedge y1\ z1\ y2\ z2.\ (y1,\ z1) \in r1 \Longrightarrow (y2,\ z2) \in r2 \Longrightarrow f\ y1\ y2 = f\ z1\ z2$
shows $congruent2\ r1\ r2\ f$
⟨*proof*⟩

lemma *congruent2D*: $congruent2\ r1\ r2\ f \Longrightarrow (y1,\ z1) \in r1 \Longrightarrow (y2,\ z2) \in r2 \Longrightarrow f\ y1\ y2 = f\ z1\ z2$
⟨*proof*⟩

Abbreviation for the common case where the relations are identical.

abbreviation *RESPECTS2*:: $('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('a \times 'a)\ set \Rightarrow bool$ (**infixr** *respects2 80*)
where $f\ respects2\ r \equiv congruent2\ r\ r\ f$

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*:
$equiv\ A1\ r1 \Longrightarrow equiv\ A2\ r2 \Longrightarrow congruent2\ r1\ r2\ f$
$\Longrightarrow X1 \in A1//r1 \Longrightarrow X2 \in A2//r2$
$\Longrightarrow (\bigwedge x1\ x2.\ x1 \in A1 \Longrightarrow x2 \in A2 \Longrightarrow f\ x1\ x2 \in B)$
$\Longrightarrow (\bigcup x1 \in X1.\ \bigcup x2 \in X2.\ f\ x1\ x2) \in B$
⟨*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*:
$equiv\ A1\ r1 \Longrightarrow equiv\ A2\ r2$
$\Longrightarrow (\bigwedge y\ z\ w.\ w \in A2 \Longrightarrow (y,z) \in r1 \Longrightarrow f\ y\ w = f\ z\ w)$
$\Longrightarrow (\bigwedge y\ z\ w.\ w \in A1 \Longrightarrow (y,z) \in r2 \Longrightarrow f\ w\ y = f\ w\ z)$
$\Longrightarrow congruent2\ r1\ r2\ f$
— 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*: $\bigwedge y\ z.\ y \in A \Longrightarrow z \in A \Longrightarrow f\ y\ z = f\ z\ y$
and *congt*: $\bigwedge y\ z\ w.\ w \in A \Longrightarrow (y,z) \in r \Longrightarrow f\ w\ y = f\ w\ z$
shows $f\ respects2\ r$
⟨*proof*⟩

44.7 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*: $finite\ A \Longrightarrow r \subseteq A \times A \Longrightarrow finite\ (A//r)$
— recall $equiv\ ?A\ ?r \Longrightarrow ?r \subseteq ?A \times ?A$
⟨*proof*⟩

lemma *finite-equiv-class*: $finite\ A \Longrightarrow r \subseteq A \times A \Longrightarrow X \in A//r \Longrightarrow finite\ X$
⟨*proof*⟩

lemma *equiv-imp-dvd-card*: $finite\ A \Longrightarrow equiv\ A\ r \Longrightarrow \forall X \in A//r.\ k\ dvd\ card\ X$

$\Longrightarrow$ *k dvd card A*
⟨*proof*⟩

lemma *card-quotient-disjoint*: *finite A* $\Longrightarrow$ *inj-on* ($\lambda x.$ $\{x\}$ // *r*) *A* $\Longrightarrow$ *card* (*A*//*r*) = *card A*
⟨*proof*⟩

44.8 Projection

definition *proj* :: ($'b \times 'a$) *set* $\Rightarrow$ $'b \Rightarrow$ $'a$ *set*
where *proj r x* = *r* `` $\{x\}$

lemma *proj-preserves*: $x \in A \Longrightarrow$ *proj r x* $\in$ *A*//*r*
⟨*proof*⟩

lemma *proj-in-iff*:
assumes *equiv A r*
shows *proj r x* $\in$ *A*//*r* $\longleftrightarrow$ $x \in A$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *proj-iff*: *equiv A r* $\Longrightarrow$ $\{x, y\} \subseteq A \Longrightarrow$ *proj r x* = *proj r y* $\longleftrightarrow$ $(x, y) \in r$
⟨*proof*⟩

lemma *proj-image*: *proj r* ` *A* = *A*//*r*
⟨*proof*⟩

lemma *in-quotient-imp-non-empty*: *equiv A r* $\Longrightarrow$ $X \in$ *A*//*r* $\Longrightarrow$ $X \neq \{\}$
⟨*proof*⟩

lemma *in-quotient-imp-in-rel*: *equiv A r* $\Longrightarrow$ $X \in$ *A*//*r* $\Longrightarrow$ $\{x, y\} \subseteq X \Longrightarrow (x, y) \in r$
⟨*proof*⟩

lemma *in-quotient-imp-closed*: *equiv A r* $\Longrightarrow$ $X \in$ *A*//*r* $\Longrightarrow$ $x \in X \Longrightarrow (x, y) \in r \Longrightarrow y \in X$
⟨*proof*⟩

lemma *in-quotient-imp-subset*: *equiv A r* $\Longrightarrow$ $X \in$ *A*//*r* $\Longrightarrow$ $X \subseteq A$
⟨*proof*⟩

44.9 Equivalence relations – predicate version

Partial equivalences.

definition *part-equivp* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ *bool*
where *part-equivp R* $\longleftrightarrow$ ($\exists x.$ *R x x*) $\wedge$ ($\forall x\ y.$ *R x y* $\longleftrightarrow$ *R x x* $\wedge$ *R y y* $\wedge$ *R x* = *R y*)

— John-Harrison-style characterization

lemma *part-equivpI*: $\exists x.\ R\ x\ x \Longrightarrow symp\ R \Longrightarrow transp\ R \Longrightarrow part\text{-}equivp\ R$
⟨*proof*⟩

lemma *part-equivpE*:
assumes *part-equivp R*
obtains x **where** $R\ x\ x$ **and** *symp R* **and** *transp R*
⟨*proof*⟩

lemma *part-equivp-refl-symp-transp*: $part\text{-}equivp\ R \longleftrightarrow (\exists x.\ R\ x\ x) \wedge symp\ R \wedge transp\ R$
⟨*proof*⟩

lemma *part-equivp-symp*: $part\text{-}equivp\ R \Longrightarrow R\ x\ y \Longrightarrow R\ y\ x$
⟨*proof*⟩

lemma *part-equivp-transp*: $part\text{-}equivp\ R \Longrightarrow R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
⟨*proof*⟩

lemma *part-equivp-typedef*: $part\text{-}equivp\ R \Longrightarrow \exists d.\ d \in \{c.\ \exists x.\ R\ x\ x \wedge c = Collect\ (R\ x)\}$
⟨*proof*⟩

Total equivalences.

definition $equivp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $equivp\ R \longleftrightarrow (\forall x\ y.\ R\ x\ y = (R\ x = R\ y))$ — John-Harrison-style characterization

lemma *equivpI*: $reflp\ R \Longrightarrow symp\ R \Longrightarrow transp\ R \Longrightarrow equivp\ R$
⟨*proof*⟩

lemma *equivpE*:
assumes *equivp R*
obtains *reflp R* **and** *symp R* **and** *transp R*
⟨*proof*⟩

lemma *equivp-implies-part-equivp*: $equivp\ R \Longrightarrow part\text{-}equivp\ R$
⟨*proof*⟩

lemma *equivp-equiv*: $equiv\ UNIV\ A \longleftrightarrow equivp\ (\lambda x\ y.\ (x,\ y) \in A)$
⟨*proof*⟩

lemma *equivp-reflp-symp-transp*: $equivp\ R \longleftrightarrow reflp\ R \wedge symp\ R \wedge transp\ R$
⟨*proof*⟩

lemma *identity-equivp*: $equivp\ (op\ =)$
⟨*proof*⟩

lemma *equivp-reflp*: $equivp\ R \Longrightarrow R\ x\ x$
⟨*proof*⟩

lemma *equivp-symp*: $equivp\ R \Longrightarrow R\ x\ y \Longrightarrow R\ y\ x$
⟨*proof*⟩

lemma *equivp-transp*: $equivp\ R \Longrightarrow R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
⟨*proof*⟩

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

end

45 Lifting package

theory *Lifting*
imports *Equiv-Relations Transfer*
keywords
parametric **and**
print-quot-maps print-quotients :: *diag* **and**
lift-definition :: *thy-goal* **and**
setup-lifting lifting-forget lifting-update :: *thy-decl*
begin

45.1 Function map

context includes *lifting-syntax*
begin

lemma *map-fun-id*:
$(id\ {-}{-}{-}{>}\ id) = id$
⟨*proof*⟩

45.2 Quotient Predicate

definition
$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
$(\forall\, a.\ Abs\ (Rep\ a) = a) \wedge$
$(\forall\, a.\ R\ (Rep\ a)\ (Rep\ a)) \wedge$
$(\forall\, r\ s.\ R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s) \wedge$
$T = (\lambda x\ y.\ R\ x\ x \wedge Abs\ x = y)$

lemma *QuotientI*:
assumes $\bigwedge a.\ Abs\ (Rep\ a) = a$
and $\bigwedge a.\ R\ (Rep\ a)\ (Rep\ a)$
and $\bigwedge r\ s.\ R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s$
and $T = (\lambda x\ y.\ R\ x\ x \wedge Abs\ x = y)$
shows $Quotient\ R\ Abs\ Rep\ T$
⟨*proof*⟩

context
fixes R Abs Rep T
assumes a: $Quotient\ R\ Abs\ Rep\ T$
begin

lemma *Quotient-abs-rep*: $Abs\ (Rep\ a) = a$
⟨*proof*⟩

lemma *Quotient-rep-reflp*: $R\ (Rep\ a)\ (Rep\ a)$
⟨*proof*⟩

lemma *Quotient-rel*:
$R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s \longleftrightarrow R\ r\ s$ — orientation does not loop on rewriting
⟨*proof*⟩

lemma *Quotient-cr-rel*: $T = (\lambda x\ y.\ R\ x\ x \wedge Abs\ x = y)$
⟨*proof*⟩

lemma *Quotient-refl1*: $R\ r\ s \Longrightarrow R\ r\ r$
⟨*proof*⟩

lemma *Quotient-refl2*: $R\ r\ s \Longrightarrow R\ s\ s$
⟨*proof*⟩

lemma *Quotient-rel-rep*: $R\ (Rep\ a)\ (Rep\ b) \longleftrightarrow a = b$
⟨*proof*⟩

lemma *Quotient-rep-abs*: $R\ r\ r \Longrightarrow R\ (Rep\ (Abs\ r))\ r$
⟨*proof*⟩

lemma *Quotient-rep-abs-eq*: $R\ t\ t \Longrightarrow R \leq op{=} \Longrightarrow Rep\ (Abs\ t) = t$
⟨*proof*⟩

lemma *Quotient-rep-abs-fold-unmap*:
assumes $x' \equiv Abs\ x$ **and** $R\ x\ x$ **and** $Rep\ x' \equiv Rep'\ x'$
shows $R\ (Rep'\ x')\ x$
⟨*proof*⟩

lemma *Quotient-Rep-eq*:
assumes $x' \equiv Abs\ x$
shows $Rep\ x' \equiv Rep\ x'$
⟨*proof*⟩

lemma *Quotient-rel-abs*: $R\ r\ s \Longrightarrow Abs\ r = Abs\ s$
⟨*proof*⟩

lemma *Quotient-rel-abs2*:

assumes $R\ (Rep\ x)\ y$
shows $x = Abs\ y$
⟨*proof*⟩

lemma *Quotient-symp*: *symp R*
⟨*proof*⟩

lemma *Quotient-transp*: *transp R*
⟨*proof*⟩

lemma *Quotient-part-equivp*: *part-equivp R*
⟨*proof*⟩

end

lemma *identity-quotient*: $Quotient\ (op\ =)\ id\ id\ (op\ =)$
⟨*proof*⟩

TODO: Use one of these alternatives as the real definition.

lemma *Quotient-alt-def*:
$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
$(\forall a\ b.\ T\ a\ b \longrightarrow Abs\ a = b) \wedge$
$(\forall b.\ T\ (Rep\ b)\ b) \wedge$
$(\forall x\ y.\ R\ x\ y \longleftrightarrow T\ x\ (Abs\ x) \wedge T\ y\ (Abs\ y) \wedge Abs\ x = Abs\ y)$
⟨*proof*⟩

lemma *Quotient-alt-def2*:
$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
$(\forall a\ b.\ T\ a\ b \longrightarrow Abs\ a = b) \wedge$
$(\forall b.\ T\ (Rep\ b)\ b) \wedge$
$(\forall x\ y.\ R\ x\ y \longleftrightarrow T\ x\ (Abs\ y) \wedge T\ y\ (Abs\ x))$
⟨*proof*⟩

lemma *Quotient-alt-def3*:
$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
$(\forall a\ b.\ T\ a\ b \longrightarrow Abs\ a = b) \wedge (\forall b.\ T\ (Rep\ b)\ b) \wedge$
$(\forall x\ y.\ R\ x\ y \longleftrightarrow (\exists z.\ T\ x\ z \wedge T\ y\ z))$
⟨*proof*⟩

lemma *Quotient-alt-def4*:
$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
$(\forall a\ b.\ T\ a\ b \longrightarrow Abs\ a = b) \wedge (\forall b.\ T\ (Rep\ b)\ b) \wedge R = T\ OO\ conversep\ T$
⟨*proof*⟩

lemma *Quotient-alt-def5*:
$Quotient\ R\ Abs\ Rep\ T \longleftrightarrow$
$T \leq BNF\text{-}Def.Grp\ UNIV\ Abs \wedge BNF\text{-}Def.Grp\ UNIV\ Rep \leq T^{-1-1} \wedge R = T$
$OO\ T^{-1-1}$
⟨*proof*⟩

lemma *fun-quotient*:
assumes *1*: *Quotient R1 abs1 rep1 T1*
assumes *2*: *Quotient R2 abs2 rep2 T2*
shows *Quotient* (*R1* ===> *R2*) (*rep1* ---> *abs2*) (*abs1* ---> *rep2*) (*T1* ===> *T2*)
⟨*proof*⟩

lemma *apply-rsp*:
fixes *f g*::*'a* ⇒ *'c*
assumes *q*: *Quotient R1 Abs1 Rep1 T1*
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*⟩

lemma *apply-rsp''*:
assumes *Quotient R Abs Rep T*
and (*R* ===> *S*) *f f*
shows *S* (*f* (*Rep x*)) (*f* (*Rep x*))
⟨*proof*⟩

45.3 Quotient composition

lemma *Quotient-compose*:
assumes *1*: *Quotient R1 Abs1 Rep1 T1*
assumes *2*: *Quotient R2 Abs2 Rep2 T2*
shows *Quotient* (*T1 OO R2 OO conversep T1*) (*Abs2* ∘ *Abs1*) (*Rep1* ∘ *Rep2*) (*T1 OO T2*)
⟨*proof*⟩

lemma *equivp-reflp2*:
equivp R ⟹ *reflp R*
⟨*proof*⟩

45.4 Respects predicate

definition *Respects* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a set*
where *Respects R* = {*x*. *R x x*}

lemma *in-respects*: *x* ∈ *Respects R* ⟷ *R x x*
⟨*proof*⟩

lemma *UNIV-typedef-to-Quotient*:
assumes *type-definition Rep Abs UNIV*
and *T-def*: *T* ≡ (λ*x y*. *x* = *Rep y*)

shows *Quotient* (*op* =) *Abs Rep T*
⟨*proof*⟩

lemma *UNIV-typedef-to-equivp*:
fixes *Abs* :: $'a \Rightarrow 'b$
and *Rep* :: $'b \Rightarrow 'a$
assumes *type-definition Rep Abs* (*UNIV*::$'a$ *set*)
shows *equivp* (*op*=::$'a\Rightarrow'a\Rightarrow$*bool*)
⟨*proof*⟩

lemma *typedef-to-Quotient*:
assumes *type-definition Rep Abs S*
and *T-def*: $T \equiv (\lambda x\ y.\ x = Rep\ y)$
shows *Quotient* (*eq-onp* ($\lambda x.\ x \in S$)) *Abs Rep T*
⟨*proof*⟩

lemma *typedef-to-part-equivp*:
assumes *type-definition Rep Abs S*
shows *part-equivp* (*eq-onp* ($\lambda x.\ x \in S$))
⟨*proof*⟩

lemma *open-typedef-to-Quotient*:
assumes *type-definition Rep Abs* $\{x.\ P\ x\}$
and *T-def*: $T \equiv (\lambda x\ y.\ x = Rep\ y)$
shows *Quotient* (*eq-onp P*) *Abs Rep T*
⟨*proof*⟩

lemma *open-typedef-to-part-equivp*:
assumes *type-definition Rep Abs* $\{x.\ P\ x\}$
shows *part-equivp* (*eq-onp P*)
⟨*proof*⟩

lemma *type-definition-Quotient-not-empty*: *Quotient* (*eq-onp P*) *Abs Rep T* $\Longrightarrow$ $\exists x.\ P\ x$
⟨*proof*⟩

lemma *type-definition-Quotient-not-empty-witness*: *Quotient* (*eq-onp P*) *Abs Rep T* $\Longrightarrow$ *P* (*Rep undefined*)
⟨*proof*⟩

Generating transfer rules for quotients.

context
fixes *R Abs Rep T*
assumes *1*: *Quotient R Abs Rep T*
begin

lemma *Quotient-right-unique*: *right-unique T*
⟨*proof*⟩

lemma *Quotient-right-total*: *right-total T*
⟨*proof*⟩

lemma *Quotient-rel-eq-transfer*: $(T\ {=}{=}{=}{>}\ T\ {=}{=}{=}{>}\ op\ {=})\ R\ (op\ {=})$
⟨*proof*⟩

lemma *Quotient-abs-induct*:
assumes $\bigwedge y.\ R\ y\ y \Longrightarrow P\ (Abs\ y)$ **shows** $P\ x$
⟨*proof*⟩

end

Generating transfer rules for total quotients.

context
fixes *R Abs Rep T*
assumes *1*: *Quotient R Abs Rep T* **and** *2*: *reflp R*
begin

lemma *Quotient-left-total*: *left-total T*
⟨*proof*⟩

lemma *Quotient-bi-total*: *bi-total T*
⟨*proof*⟩

lemma *Quotient-id-abs-transfer*: $(op\ {=}\ {=}{=}{=}{>}\ T)\ (\lambda x.\ x)\ Abs$
⟨*proof*⟩

lemma *Quotient-total-abs-induct*: $(\bigwedge y.\ P\ (Abs\ y)) \Longrightarrow P\ x$
⟨*proof*⟩

lemma *Quotient-total-abs-eq-iff*: $Abs\ x = Abs\ y \longleftrightarrow R\ x\ y$
⟨*proof*⟩

end

Generating transfer rules for a type defined with *typedef*.

context
fixes *Rep Abs A T*
assumes *type*: *type-definition Rep Abs A*
assumes *T-def*: $T \equiv (\lambda(x{::}'a)\ (y{::}'b).\ x = Rep\ y)$
begin

lemma *typedef-left-unique*: *left-unique T*
⟨*proof*⟩

lemma *typedef-bi-unique*: *bi-unique T*
⟨*proof*⟩

lemma *typedef-right-unique*: *right-unique* T
⟨*proof*⟩

lemma *typedef-right-total*: *right-total* T
⟨*proof*⟩

lemma *typedef-rep-transfer*: $(T$ ===> *op* =) $(\lambda x.\ x)$ *Rep*
⟨*proof*⟩

end

Generating the correspondence rule for a constant defined with *lift-definition*.

lemma *Quotient-to-transfer*:
assumes *Quotient* R *Abs Rep* T **and** $R\ c\ c$ **and** $c' \equiv$ *Abs* c
shows $T\ c\ c'$
⟨*proof*⟩

Proving reflexivity

lemma *Quotient-to-left-total*:
assumes *q*: *Quotient* R *Abs Rep* T
and *r-R*: *reflp* R
shows *left-total* T
⟨*proof*⟩

lemma *Quotient-composition-ge-eq*:
assumes *left-total* T
assumes $R \geq$ *op*=
shows $(T\ OO\ R\ OO\ T^{-1-1}) \geq$ *op*=
⟨*proof*⟩

lemma *Quotient-composition-le-eq*:
assumes *left-unique* T
assumes $R \leq$ *op*=
shows $(T\ OO\ R\ OO\ T^{-1-1}) \leq$ *op*=
⟨*proof*⟩

lemma *eq-onp-le-eq*:
eq-onp $P \leq$ *op*= ⟨*proof*⟩

lemma *reflp-ge-eq*:
reflp $R \Longrightarrow R \geq$ *op*= ⟨*proof*⟩

Proving a parametrized correspondence relation

definition *POS* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
$POS\ A\ B \equiv A \leq B$

definition *NEG* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
$NEG\ A\ B \equiv B \leq A$

lemma *pos-OO-eq*:
shows *POS* (*A OO op*=) *A*
⟨*proof*⟩

lemma *pos-eq-OO*:
shows *POS* (*op*= *OO A*) *A*
⟨*proof*⟩

lemma *neg-OO-eq*:
shows *NEG* (*A OO op*=) *A*
⟨*proof*⟩

lemma *neg-eq-OO*:
shows *NEG* (*op*= *OO A*) *A*
⟨*proof*⟩

lemma *POS-trans*:
assumes *POS A B*
assumes *POS B C*
shows *POS A C*
⟨*proof*⟩

lemma *NEG-trans*:
assumes *NEG A B*
assumes *NEG B C*
shows *NEG A C*
⟨*proof*⟩

lemma *POS-NEG*:
POS A B ≡ *NEG B A*
⟨*proof*⟩

lemma *NEG-POS*:
NEG A B ≡ *POS B A*
⟨*proof*⟩

lemma *POS-pcr-rule*:
assumes *POS* (*A OO B*) *C*
shows *POS* (*A OO B OO X*) (*C OO X*)
⟨*proof*⟩

lemma *NEG-pcr-rule*:
assumes *NEG* (*A OO B*) *C*
shows *NEG* (*A OO B OO X*) (*C OO X*)
⟨*proof*⟩

lemma *POS-apply*:
assumes *POS R R′*

assumes $R\ f\ g$
shows $R'\ f\ g$
⟨*proof*⟩

Proving a parametrized correspondence relation

lemma *fun-mono*:
assumes $A \geq C$
assumes $B \leq D$
shows $(A \ ===> B) \leq (C \ ===> D)$
⟨*proof*⟩

lemma *pos-fun-distr*: $((R \ ===> S)\ OO\ (R' \ ===> S')) \leq ((R\ OO\ R') \ ===> (S\ OO\ S'))$
⟨*proof*⟩

lemma *functional-relation*: *right-unique* $R \Longrightarrow$ *left-total* $R \Longrightarrow \forall\, x.\ \exists!y.\ R\ x\ y$
⟨*proof*⟩

lemma *functional-converse-relation*: *left-unique* $R \Longrightarrow$ *right-total* $R \Longrightarrow \forall\, y.\ \exists!x.\ R\ x\ y$
⟨*proof*⟩

lemma *neg-fun-distr1*:
assumes *1*: *left-unique* R *right-total* R
assumes *2*: *right-unique* R' *left-total* R'
shows $(R\ OO\ R' \ ===> S\ OO\ S') \leq ((R \ ===> S)\ OO\ (R' \ ===> S'))$
⟨*proof*⟩

lemma *neg-fun-distr2*:
assumes *1*: *right-unique* R' *left-total* R'
assumes *2*: *left-unique* S' *right-total* S'
shows $(R\ OO\ R' \ ===> S\ OO\ S') \leq ((R \ ===> S)\ OO\ (R' \ ===> S'))$
⟨*proof*⟩

45.5 Domains

lemma *composed-equiv-rel-eq-onp*:
assumes *left-unique* R
assumes $(R \ ===> op=)\ P\ P'$
assumes *Domainp* $R = P''$
shows $(R\ OO$ *eq-onp* $P'\ OO\ R^{-1-1}) =$ *eq-onp* (*inf* $P''\ P$)
⟨*proof*⟩

lemma *composed-equiv-rel-eq-eq-onp*:
assumes *left-unique* R
assumes *Domainp* $R = P$
shows $(R\ OO\ op{=}\ OO\ R^{-1-1}) =$ *eq-onp* P
⟨*proof*⟩

lemma *pcr-Domainp-par-left-total*:
assumes *Domainp B* = *P*
assumes *left-total A*
assumes (*A* ===> *op*=) *P′ P*
shows *Domainp* (*A OO B*) = *P′*
⟨*proof*⟩

lemma *pcr-Domainp-par*:
assumes *Domainp B* = *P2*
assumes *Domainp A* = *P1*
assumes (*A* ===> *op*=) *P2′ P2*
shows *Domainp* (*A OO B*) = (*inf P1 P2′*)
⟨*proof*⟩

definition *rel-pred-comp* :: (′*a* => ′*b* => *bool*) => (′*b* => *bool*) => ′*a* => *bool*
where *rel-pred-comp R P* ≡ λ*x*. ∃ *y*. *R x y* ∧ *P y*

lemma *pcr-Domainp*:
assumes *Domainp B* = *P*
shows *Domainp* (*A OO B*) = (λ*x*. ∃ *y*. *A x y* ∧ *P y*)
⟨*proof*⟩

lemma *pcr-Domainp-total*:
assumes *left-total B*
assumes *Domainp A* = *P*
shows *Domainp* (*A OO B*) = *P*
⟨*proof*⟩

lemma *Quotient-to-Domainp*:
assumes *Quotient R Abs Rep T*
shows *Domainp T* = (λ*x*. *R x x*)
⟨*proof*⟩

lemma *eq-onp-to-Domainp*:
assumes *Quotient* (*eq-onp P*) *Abs Rep T*
shows *Domainp T* = *P*
⟨*proof*⟩

end

lemma *right-total-UNIV-transfer*:
assumes *right-total A*
shows (*rel-set A*) (*Collect* (*Domainp A*)) *UNIV*
⟨*proof*⟩

45.6 ML setup

⟨*ML*⟩

named-theorems *relator-eq-onp*
theorems that a relator of an eq-onp is an eq-onp of the corresponding predicate
⟨*ML*⟩

declare *fun-quotient*[*quot-map*]
declare *fun-mono*[*relator-mono*]
lemmas [*relator-distr*] = *pos-fun-distr neg-fun-distr1 neg-fun-distr2*

⟨*ML*⟩

lemma *pred-prod-beta*: *pred-prod P Q xy* ⟷ *P* (*fst xy*) ∧ *Q* (*snd xy*)
⟨*proof*⟩

lemma *pred-prod-split*: *P* (*pred-prod Q R xy*) ⟷ (∀ *x y*. *xy* = (*x*, *y*) ⟶ *P* (*Q x* ∧ *R y*))
⟨*proof*⟩

hide-const (**open**) *POS NEG*

end

46 Definition of Quotient Types

theory *Quotient*
imports *Lifting*
keywords
print-quotmapsQ3 print-quotientsQ3 print-quotconsts :: *diag* **and**
quotient-type :: *thy-goal* **and** / **and**
quotient-definition :: *thy-goal*
begin

Basic definition for equivalence relations that are represented by predicates.

Composition of Relations

abbreviation
rel-conj :: (′*a* ⇒ ′*b* ⇒ *bool*) ⇒ (′*b* ⇒ ′*a* ⇒ *bool*) ⇒ ′*a* ⇒ ′*b* ⇒ *bool* (**infixr** *OOO* *75*)
where
r1 OOO r2 ≡ *r1 OO r2 OO r1*

lemma *eq-comp-r*:
shows ((*op* =) *OOO R*) = *R*
⟨*proof*⟩

context includes *lifting-syntax*
begin

46.1 Quotient Predicate

definition
  Quotient3 R Abs Rep $\longleftrightarrow$
    $(\forall a.\ Abs\ (Rep\ a) = a) \wedge (\forall a.\ R\ (Rep\ a)\ (Rep\ a)) \wedge$
    $(\forall r\ s.\ R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s)$

lemma *Quotient3I*:
  assumes $\bigwedge a.\ Abs\ (Rep\ a) = a$
    and $\bigwedge a.\ R\ (Rep\ a)\ (Rep\ a)$
    and $\bigwedge r\ s.\ R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s$
  shows *Quotient3 R Abs Rep*
  ⟨*proof*⟩

context
  fixes *R Abs Rep*
  assumes *a*: *Quotient3 R Abs Rep*
begin

lemma *Quotient3-abs-rep*:
  $Abs\ (Rep\ a) = a$
  ⟨*proof*⟩

lemma *Quotient3-rep-reflp*:
  $R\ (Rep\ a)\ (Rep\ a)$
  ⟨*proof*⟩

lemma *Quotient3-rel*:
  $R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s \longleftrightarrow R\ r\ s$ — orientation does not loop on rewriting
  ⟨*proof*⟩

lemma *Quotient3-refl1*:
  $R\ r\ s \Longrightarrow R\ r\ r$
  ⟨*proof*⟩

lemma *Quotient3-refl2*:
  $R\ r\ s \Longrightarrow R\ s\ s$
  ⟨*proof*⟩

lemma *Quotient3-rel-rep*:
  $R\ (Rep\ a)\ (Rep\ b) \longleftrightarrow a = b$
  ⟨*proof*⟩

lemma *Quotient3-rep-abs*:
  $R\ r\ r \Longrightarrow R\ (Rep\ (Abs\ r))\ r$
  ⟨*proof*⟩

lemma *Quotient3-rel-abs*:
  $R\ r\ s \Longrightarrow Abs\ r = Abs\ s$

⟨*proof*⟩

lemma *Quotient3-symp*:
symp R
⟨*proof*⟩

lemma *Quotient3-transp*:
transp R
⟨*proof*⟩

lemma *Quotient3-part-equivp*:
part-equivp R
⟨*proof*⟩

lemma *abs-o-rep*:
Abs o Rep = id
⟨*proof*⟩

lemma *equals-rsp*:
assumes *b*: *R xa xb R ya yb*
shows *R xa ya = R xb yb*
⟨*proof*⟩

lemma *rep-abs-rsp*:
assumes *b*: *R x1 x2*
shows *R x1* (*Rep* (*Abs x2*))
⟨*proof*⟩

lemma *rep-abs-rsp-left*:
assumes *b*: *R x1 x2*
shows *R* (*Rep* (*Abs x1*)) *x2*
⟨*proof*⟩

end

lemma *identity-quotient3*:
Quotient3 (*op* =) *id id*
⟨*proof*⟩

lemma *fun-quotient3*:
assumes *q1*: *Quotient3 R1 abs1 rep1*
and *q2*: *Quotient3 R2 abs2 rep2*
shows *Quotient3* (*R1* ===> *R2*) (*rep1* −−−> *abs2*) (*abs1* −−−> *rep2*)
⟨*proof*⟩

lemma *lambda-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* −−−> *Abs2*) ($\lambda x.$ *Rep2* (*f* (*Abs1 x*))) = ($\lambda x.$ *f x*)

⟨*proof*⟩

lemma *lambda-prs1*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows $(Rep1\ {-}{-}{-}{>}\ Abs2)\ (\lambda x.\ (Abs1\ {-}{-}{-}{>}\ Rep2)\ f\ x) = (\lambda x.\ f\ x)$
⟨*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-rspQ3*:
fixes $f\ g{::}'a \Rightarrow 'c$
assumes *q*: *Quotient3 R1 Abs1 Rep1*
and *a*: $(R1\ {=}{=}{=}{>}\ R2)\ f\ g\ R1\ x\ y$
shows $R2\ (f\ x)\ (g\ y)$
⟨*proof*⟩

lemma *apply-rspQ3''*:
assumes *Quotient3 R Abs Rep*
and $(R\ {=}{=}{=}{>}\ S)\ f\ f$
shows $S\ (f\ (Rep\ x))\ (f\ (Rep\ x))$
⟨*proof*⟩

46.2 lemmas for regularisation of ball and bex

lemma *ball-reg-eqv*:
fixes $P :: 'a \Rightarrow bool$
assumes *a*: *equivp R*
shows $Ball\ (Respects\ R)\ P = (All\ P)$
⟨*proof*⟩

lemma *bex-reg-eqv*:
fixes $P :: 'a \Rightarrow bool$
assumes *a*: *equivp R*
shows $Bex\ (Respects\ R)\ P = (Ex\ P)$
⟨*proof*⟩

lemma *ball-reg-right*:
assumes *a*: $\bigwedge x.\ x \in R \Longrightarrow P\ x \longrightarrow Q\ x$
shows $All\ P \longrightarrow Ball\ R\ Q$
⟨*proof*⟩

lemma *bex-reg-left*:
assumes *a*: $\bigwedge x.\ x \in R \Longrightarrow Q\ x \longrightarrow P\ x$
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: $!x :: 'a.\ (x \in R\ {-}{-}{>}\ P\ x\ {-}{-}{>}\ Q\ x)$
and b: *Ball R P*
shows *Ball R Q*
⟨*proof*⟩

lemma *bex-reg*:
assumes a: $!x :: 'a.\ (x \in R\ {-}{-}{>}\ P\ x\ {-}{-}{>}\ Q\ x)$
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*⟩

46.3 Bounded abstraction

definition
$Babs :: 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where
$x \in p \Longrightarrow Babs\ p\ m\ x = m\ x$

lemma *babs-rsp*:
assumes *q*: *Quotient3 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*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows $((Rep1 ---> Abs2)\ (Babs\ (Respects\ R1)\ ((Abs1 ---> Rep2)\ f))) = f$
⟨*proof*⟩

lemma *babs-simp*:
assumes *q*: *Quotient3 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 $Ex1\ (\lambda x.\ x \in Respects\ R \wedge f\ x) = Ex1\ (\lambda x.\ x \in Respects\ R \wedge g\ x)$
⟨*proof*⟩

lemma *all-prs*:
assumes a: *Quotient3 R absf repf*
shows $Ball\ (Respects\ R)\ ((absf\ {---}{>}\ id)\ f) = All\ f$
⟨*proof*⟩

lemma *ex-prs*:
assumes a: *Quotient3 R absf repf*
shows $Bex\ (Respects\ R)\ ((absf\ {---}{>}\ id)\ f) = Ex\ f$
⟨*proof*⟩

46.4 *Bex1-rel* quantifier

definition
$Bex1\text{-}rel :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where
$Bex1\text{-}rel\ R\ P \longleftrightarrow (\exists\, x \in Respects\ R.\ P\ x) \wedge (\forall\, x \in Respects\ R.\ \forall\, y \in Respects\ R.\ ((P\ x \wedge P\ y) \longrightarrow (R\ x\ y)))$

lemma *bex1-rel-aux*:
$[\![\forall\, xa\ ya.\ R\ xa\ ya \longrightarrow x\ xa = y\ ya;\ Bex1\text{-}rel\ R\ x]\!] \Longrightarrow Bex1\text{-}rel\ R\ y$
⟨*proof*⟩

lemma *bex1-rel-aux2*:
$[\![\forall\, xa\ ya.\ R\ xa\ ya \longrightarrow x\ xa = y\ ya;\ Bex1\text{-}rel\ R\ y]\!] \Longrightarrow Bex1\text{-}rel\ R\ x$
⟨*proof*⟩

lemma *bex1-rel-rsp*:
assumes a: *Quotient3 R absf repf*
shows $((R\ {===}{>}\ op\ =)\ {===}{>}\ op\ =)\ (Bex1\text{-}rel\ R)\ (Bex1\text{-}rel\ R)$
⟨*proof*⟩

lemma *ex1-prs*:
assumes a: *Quotient3 R absf repf*
shows $((absf\ {---}{>}\ id)\ {---}{>}\ id)\ (Bex1\text{-}rel\ R)\ f = Ex1\ f$
⟨*proof*⟩

lemma *bex1-bexeq-reg*:
shows $(\exists!x{\in}Respects\ R.\ P\ x) \longrightarrow (Bex1\text{-}rel\ R\ (\lambda x.\ P\ x))$
⟨*proof*⟩

lemma *bex1-bexeq-reg-eqv*:
assumes *a*: *equivp R*
shows $(\exists!x.\ P\ x) \longrightarrow$ *Bex1-rel R P*
⟨*proof*⟩

46.5 Various respects and preserve lemmas

lemma *quot-rel-rsp*:
assumes *a*: *Quotient3 R Abs Rep*
shows (*R* ===> *R* ===> *op* =) *R R*
⟨*proof*⟩

lemma *o-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
and *q3*: *Quotient3 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*: *Quotient3 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*: *Quotient3 R Abs Rep*
shows (*id* −−−> *Rep* −−−> *Rep* −−−> *Abs*) *If* = *If*
⟨*proof*⟩

lemma *if-rsp*:
assumes *q*: *Quotient3 R Abs Rep*
shows (*op* = ===> *R* ===> *R* ===> *R*) *If If*
⟨*proof*⟩

lemma *let-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep2* −−−> (*Abs2* −−−> *Rep1*) −−−> *Abs1*) *Let* = *Let*
⟨*proof*⟩

lemma *let-rsp*:

shows $(R1 ===> (R1 ===> R2) ===> R2)$ *Let Let*
⟨*proof*⟩

lemma *id-rsp*:
shows $(R ===> R)$ *id id*
⟨*proof*⟩

lemma *id-prs*:
assumes *a*: *Quotient3 R Abs Rep*
shows $(Rep\ {-}{-}{-}{>}\ Abs)\ id = id$
⟨*proof*⟩

end

locale *quot-type* =
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$
and $Abs :: 'a\ set \Rightarrow 'b$
and $Rep :: 'b \Rightarrow 'a\ set$
assumes *equivp*: *part-equivp R*
and *rep-prop*: $\bigwedge y.\ \exists x.\ R\ x\ x \wedge Rep\ y = Collect\ (R\ x)$
and *rep-inverse*: $\bigwedge x.\ Abs\ (Rep\ x) = x$
and *abs-inverse*: $\bigwedge c.\ (\exists x.\ ((R\ x\ x) \wedge (c = Collect\ (R\ x)))) \Longrightarrow (Rep\ (Abs\ c)) = c$
and *rep-inject*: $\bigwedge x\ y.\ (Rep\ x = Rep\ y) = (x = y)$
begin

definition
$abs :: 'a \Rightarrow 'b$
where
$abs\ x = Abs\ (Collect\ (R\ x))$

definition
$rep :: 'b \Rightarrow 'a$
where
$rep\ a = (SOME\ x.\ x \in Rep\ a)$

lemma *some-collect*:
assumes *R r r*
shows $R\ (SOME\ x.\ x \in Collect\ (R\ r)) = R\ r$
⟨*proof*⟩

lemma *Quotient*:
shows *Quotient3 R abs rep*
⟨*proof*⟩

end

46.6 Quotient composition

lemma *OOO-quotient3*:
  fixes $R1 :: 'a \Rightarrow 'a \Rightarrow bool$
  fixes $Abs1 :: 'a \Rightarrow 'b$ **and** $Rep1 :: 'b \Rightarrow 'a$
  fixes $Abs2 :: 'b \Rightarrow 'c$ **and** $Rep2 :: 'c \Rightarrow 'b$
  fixes $R2' :: 'a \Rightarrow 'a \Rightarrow bool$
  fixes $R2 :: 'b \Rightarrow 'b \Rightarrow bool$
  assumes *R1*: *Quotient3 R1 Abs1 Rep1*
  assumes *R2*: *Quotient3 R2 Abs2 Rep2*
  assumes *Abs1*: $\bigwedge x\ y.\ R2'\ x\ y \Longrightarrow R1\ x\ x \Longrightarrow R1\ y\ y \Longrightarrow R2\ (Abs1\ x)\ (Abs1\ y)$
  assumes *Rep1*: $\bigwedge x\ y.\ R2\ x\ y \Longrightarrow R2'\ (Rep1\ x)\ (Rep1\ y)$
  shows $Quotient3\ (R1\ OO\ R2'\ OO\ R1)\ (Abs2 \circ Abs1)\ (Rep1 \circ Rep2)$
⟨*proof*⟩

lemma *OOO-eq-quotient3*:
  fixes $R1 :: 'a \Rightarrow 'a \Rightarrow bool$
  fixes $Abs1 :: 'a \Rightarrow 'b$ **and** $Rep1 :: 'b \Rightarrow 'a$
  fixes $Abs2 :: 'b \Rightarrow 'c$ **and** $Rep2 :: 'c \Rightarrow 'b$
  assumes *R1*: *Quotient3 R1 Abs1 Rep1*
  assumes *R2*: *Quotient3 op= Abs2 Rep2*
  shows $Quotient3\ (R1\ OOO\ op=)\ (Abs2 \circ Abs1)\ (Rep1 \circ Rep2)$
⟨*proof*⟩

46.7 Quotient3 to Quotient

lemma *Quotient3-to-Quotient*:
assumes *Quotient3 R Abs Rep*
and $T \equiv \lambda x\ y.\ R\ x\ x \wedge Abs\ x = y$
shows *Quotient R Abs Rep T*
⟨*proof*⟩

lemma *Quotient3-to-Quotient-equivp*:
assumes *q*: *Quotient3 R Abs Rep*
and *T-def*: $T \equiv \lambda x\ y.\ Abs\ x = y$
and *eR*: *equivp R*
shows *Quotient R Abs Rep T*
⟨*proof*⟩

46.8 ML setup

Auxiliary data for the quotient package

named-theorems *quot-equiv equivalence relation theorems*
  and *quot-respect respectfulness theorems*
  and *quot-preserve preservation theorems*
  and *id-simps identity simp rules for maps*
  and *quot-thm quotient theorems*
⟨*ML*⟩

declare [[*mapQ3 fun* = (*rel-fun*, *fun-quotient3*)]]

lemmas [*quot-thm*] = *fun-quotient3*
lemmas [*quot-respect*] = *quot-rel-rsp if-rsp o-rsp let-rsp id-rsp*
lemmas [*quot-preserve*] = *if-prs o-prs let-prs id-prs*
lemmas [*quot-equiv*] = *identity-equivp*

Lemmas about simplifying id's.

lemmas [*id-simps*] =
id-def [*symmetric*]
map-fun-id
id-apply
id-o
o-id
eq-comp-r
vimage-id

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
$Quot\text{-}True :: 'a \Rightarrow bool$
where
$Quot\text{-}True\ x \longleftrightarrow True$

lemma
shows $QT\text{-}all$: $Quot\text{-}True\ (All\ P) \Longrightarrow Quot\text{-}True\ P$
and $QT\text{-}ex$: $Quot\text{-}True\ (Ex\ P) \Longrightarrow Quot\text{-}True\ P$
and $QT\text{-}ex1$: $Quot\text{-}True\ (Ex1\ P) \Longrightarrow Quot\text{-}True\ P$
and $QT\text{-}lam$: $Quot\text{-}True\ (\lambda x.\ P\ x) \Longrightarrow (\bigwedge x.\ Quot\text{-}True\ (P\ x))$
and $QT\text{-}ext$: $(\bigwedge x.\ Quot\text{-}True\ (a\ x) \Longrightarrow f\ x = g\ x) \Longrightarrow (Quot\text{-}True\ a \Longrightarrow f = g)$
⟨*proof*⟩

lemma $QT\text{-}imp$: $Quot\text{-}True\ a \equiv Quot\text{-}True\ b$
⟨*proof*⟩

context includes *lifting-syntax*
begin

Tactics for proving the lifted theorems

⟨*ML*⟩

end

46.9 Methods / Interface

⟨*ML*⟩

no-notation
rel-conj (**infixr** *OOO 75*)

end

47 Chain-complete partial orders and their fixpoints

theory *Complete-Partial-Order*
imports *Product-Type*
begin

47.1 Monotone functions

Dictionary-passing version of *mono*.

definition *monotone* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where $monotone\ orda\ ordb\ f \longleftrightarrow (\forall x\ y.\ orda\ x\ y \longrightarrow ordb\ (f\ x)\ (f\ y))$

lemma *monotoneI*[*intro?*]: $(\bigwedge x\ y.\ orda\ x\ y \Longrightarrow ordb\ (f\ x)\ (f\ y)) \Longrightarrow monotone\ orda\ ordb\ f$
⟨*proof*⟩

lemma *monotoneD*[*dest?*]: $monotone\ orda\ ordb\ f \Longrightarrow orda\ x\ y \Longrightarrow ordb\ (f\ x)\ (f\ y)$
⟨*proof*⟩

47.2 Chains

A chain is a totally-ordered set. Chains are parameterized over the order for maximal flexibility, since type classes are not enough.

definition *chain* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$
where $chain\ ord\ S \longleftrightarrow (\forall x{\in}S.\ \forall y{\in}S.\ ord\ x\ y \lor ord\ y\ x)$

lemma *chainI*:
assumes $\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow ord\ x\ y \lor ord\ y\ x$
shows *chain ord S*
⟨*proof*⟩

lemma *chainD*:
  assumes *chain ord S* **and** $x \in S$ **and** $y \in S$
  shows *ord x y* $\vee$ *ord y x*
  ⟨*proof*⟩

lemma *chainE*:
  assumes *chain ord S* **and** $x \in S$ **and** $y \in S$
  obtains *ord x y* | *ord y x*
  ⟨*proof*⟩

lemma *chain-empty*: *chain ord* {}
  ⟨*proof*⟩

lemma *chain-equality*: *chain op* $=$ $A \longleftrightarrow (\forall x{\in}A.\ \forall y{\in}A.\ x = y)$
  ⟨*proof*⟩

lemma *chain-subset*: *chain ord A* $\Longrightarrow B \subseteq A \Longrightarrow$ *chain ord B*
  ⟨*proof*⟩

lemma *chain-imageI*:
  assumes *chain*: *chain le-a Y*
    and *mono*: $\bigwedge x\ y.\ x \in Y \Longrightarrow y \in Y \Longrightarrow$ *le-a x y* $\Longrightarrow$ *le-b* $(f\ x)\ (f\ y)$
  shows *chain le-b* $(f$ ' $Y)$
  ⟨*proof*⟩

47.3 Chain-complete partial orders

A *ccpo* has a least upper bound for any chain. In particular, the empty set is a chain, so every *ccpo* must have a bottom element.

class *ccpo* = *order* + *Sup* +
  assumes *ccpo-Sup-upper*: *chain* $(op \leq)$ $A \Longrightarrow x \in A \Longrightarrow x \leq$ *Sup A*
  assumes *ccpo-Sup-least*: *chain* $(op \leq)$ $A \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow x \leq z) \Longrightarrow$ *Sup* $A \leq z$
begin

lemma *chain-singleton*: *Complete-Partial-Order.chain op* $\leq \{x\}$
  ⟨*proof*⟩

lemma *ccpo-Sup-singleton* [*simp*]: $\bigsqcup\{x\} = x$
  ⟨*proof*⟩

47.4 Transfinite iteration of a function

context notes [[*inductive-internals*]]
begin

inductive-set *iterates* :: $('a \Rightarrow 'a) \Rightarrow 'a$ *set*
  for $f :: 'a \Rightarrow 'a$
  where

  step: $x \in$ *iterates* $f \Longrightarrow f\, x \in$ *iterates* f
| *Sup*: *chain* (*op* $\leq$) $M \Longrightarrow \forall\, x{\in}M.\ x \in$ *iterates* $f \Longrightarrow$ *Sup* $M \in$ *iterates* f

end

lemma *iterates-le-f*: $x \in$ *iterates* $f \Longrightarrow$ *monotone* (*op* $\leq$) (*op* $\leq$) $f \Longrightarrow x \leq f\, x$
⟨*proof*⟩

lemma *chain-iterates*:
assumes f: *monotone* (*op* $\leq$) (*op* $\leq$) f
shows *chain* (*op* $\leq$) (*iterates* f) (**is** *chain* - *?C*)
⟨*proof*⟩

lemma *bot-in-iterates*: *Sup* $\{\} \in$ *iterates* f
⟨*proof*⟩

47.5 Fixpoint combinator

definition *fixp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where *fixp* f = *Sup* (*iterates* f)

lemma *iterates-fixp*:
assumes f: *monotone* (*op* $\leq$) (*op* $\leq$) f
shows *fixp* $f \in$ *iterates* f
⟨*proof*⟩

lemma *fixp-unfold*:
assumes f: *monotone* (*op* $\leq$) (*op* $\leq$) f
shows *fixp* $f = f$ (*fixp* f)
⟨*proof*⟩

lemma *fixp-lowerbound*:
assumes f: *monotone* (*op* $\leq$) (*op* $\leq$) f
and z: $f\, z \leq z$
shows *fixp* $f \leq z$
⟨*proof*⟩

end

47.6 Fixpoint induction

⟨*ML*⟩

definition *admissible* :: $('a\ set \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where *admissible lub ord P* $\longleftrightarrow (\forall\, A.$ *chain ord* $A \longrightarrow A \neq \{\} \longrightarrow (\forall\, x{\in}A.\ P\ x) \longrightarrow P\ (lub\ A))$

lemma *admissibleI*:
assumes $\bigwedge A.$ *chain ord* $A \Longrightarrow A \neq \{\} \Longrightarrow \forall\, x{\in}A.\ P\ x \Longrightarrow P\ (lub\ A)$
shows *ccpo.admissible lub ord P*

⟨proof⟩

lemma *admissibleD*:
assumes *ccpo.admissible lub ord P*
assumes *chain ord A*
assumes $A \neq \{\}$
assumes $\bigwedge x.\ x \in A \Longrightarrow P\ x$
shows $P\ (lub\ A)$
⟨proof⟩

⟨ML⟩

lemma (**in** *ccpo*) *fixp-induct*:
assumes *adm*: *ccpo.admissible Sup* $(op \leq)$ *P*
assumes *mono*: *monotone* $(op \leq)$ $(op \leq)$ *f*
assumes *bot*: $P\ (Sup\ \{\})$
assumes *step*: $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$
shows $P\ (fixp\ f)$
⟨proof⟩

lemma *admissible-True*: *ccpo.admissible lub ord* $(\lambda x.\ True)$
⟨proof⟩

lemma *admissible-const*: *ccpo.admissible lub ord* $(\lambda x.\ t)$
⟨proof⟩

lemma *admissible-conj*:
assumes *ccpo.admissible lub ord* $(\lambda x.\ P\ x)$
assumes *ccpo.admissible lub ord* $(\lambda x.\ Q\ x)$
shows *ccpo.admissible lub ord* $(\lambda x.\ P\ x \wedge Q\ x)$
⟨proof⟩

lemma *admissible-all*:
assumes $\bigwedge y.$ *ccpo.admissible lub ord* $(\lambda x.\ P\ x\ y)$
shows *ccpo.admissible lub ord* $(\lambda x.\ \forall y.\ P\ x\ y)$
⟨proof⟩

lemma *admissible-ball*:
assumes $\bigwedge y.\ y \in A \Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ P\ x\ y)$
shows *ccpo.admissible lub ord* $(\lambda x.\ \forall y{\in}A.\ P\ x\ y)$
⟨proof⟩

lemma *chain-compr*: *chain ord A* $\Longrightarrow$ *chain ord* $\{x \in A.\ P\ x\}$
⟨proof⟩

context *ccpo*
begin

lemma *admissible-disj*:
fixes $P\ Q :: 'a \Rightarrow bool$
assumes P: *ccpo.admissible Sup* $(op \leq)$ $(\lambda x.\ P\ x)$
assumes Q: *ccpo.admissible Sup* $(op \leq)$ $(\lambda x.\ Q\ x)$
shows *ccpo.admissible Sup* $(op \leq)$ $(\lambda x.\ P\ x \vee Q\ x)$
⟨*proof*⟩

end

instance *complete-lattice* $\subseteq$ *ccpo*
⟨*proof*⟩

lemma *lfp-eq-fixp*:
assumes *mono*: *mono f*
shows *lfp f* = *fixp f*
⟨*proof*⟩

hide-const (**open**) *iterates fixp*

end

48 Datatype option

theory *Option*
imports *Lifting*
begin

datatype $'a$ *option* =
None
| *Some* (*the*: $'a$)

datatype-compat *option*

lemma [*case-names None Some*, *cases type*: *option*]:
— for backward compatibility – names of variables differ
$(y = None \Longrightarrow P) \Longrightarrow (\bigwedge a.\ y = Some\ a \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma [*case-names None Some*, *induct type*: *option*]:
— for backward compatibility – names of variables differ
$P\ None \Longrightarrow (\bigwedge option.\ P\ (Some\ option)) \Longrightarrow P\ option$
⟨*proof*⟩

Compatibility:

⟨*ML*⟩
lemmas *inducts* = *option.induct*
lemmas *cases* = *option.case*
⟨*ML*⟩

lemma *not-None-eq* [*iff*]: $x \neq None \longleftrightarrow (\exists y.\ x = Some\ y)$
⟨*proof*⟩

lemma *not-Some-eq* [*iff*]: $(\forall y.\ x \neq Some\ y) \longleftrightarrow x = None$
⟨*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 *case-optionE*:
assumes *c*: $(case\ x\ of\ None \Rightarrow P \mid Some\ y \Rightarrow Q\ y)$
obtains
(*None*) $x = None$ **and** P
| (*Some*) y **where** $x = Some\ y$ **and** $Q\ y$
⟨*proof*⟩

lemma *split-option-all*: $(\forall x.\ P\ x) \longleftrightarrow P\ None \wedge (\forall x.\ P\ (Some\ x))$
⟨*proof*⟩

lemma *split-option-ex*: $(\exists x.\ P\ x) \longleftrightarrow P\ None \vee (\exists x.\ P\ (Some\ x))$
⟨*proof*⟩

lemma *UNIV-option-conv*: $UNIV = insert\ None\ (range\ Some)$
⟨*proof*⟩

lemma *rel-option-None1* [*simp*]: $rel\text{-}option\ P\ None\ x \longleftrightarrow x = None$
⟨*proof*⟩

lemma *rel-option-None2* [*simp*]: $rel\text{-}option\ P\ x\ None \longleftrightarrow x = None$
⟨*proof*⟩

lemma *option-rel-Some1*: $rel\text{-}option\ A\ (Some\ x)\ y \longleftrightarrow (\exists y'.\ y = Some\ y' \wedge A\ x\ y')$
⟨*proof*⟩

lemma *option-rel-Some2*: $rel\text{-}option\ A\ x\ (Some\ y) \longleftrightarrow (\exists x'.\ x = Some\ x' \wedge A\ x'\ y)$
⟨*proof*⟩

lemma *rel-option-inf*: $inf\ (rel\text{-}option\ A)\ (rel\text{-}option\ B) = rel\text{-}option\ (inf\ A\ B)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *rel-option-reflI*:
$(\bigwedge x.\ x \in set\text{-}option\ y \Longrightarrow P\ x\ x) \Longrightarrow rel\text{-}option\ P\ y\ y$
⟨*proof*⟩

48.0.1 Operations

lemma *ospec* [*dest*]: $(\forall x \in \textit{set-option } A.\ P\ x) \Longrightarrow A = \textit{Some } x \Longrightarrow P\ x$
⟨*proof*⟩

⟨*ML*⟩

lemma *elem-set* [*iff*]: $(x \in \textit{set-option } xo) = (xo = \textit{Some } x)$
⟨*proof*⟩

lemma *set-empty-eq* [*simp*]: $(\textit{set-option } xo = \{\}) = (xo = \textit{None})$
⟨*proof*⟩

lemma *map-option-case*: $\textit{map-option } f\ y = (\textit{case } y \textit{ of None} \Rightarrow \textit{None} \mid \textit{Some } x \Rightarrow \textit{Some } (f\ x))$
⟨*proof*⟩

lemma *map-option-is-None* [*iff*]: $(\textit{map-option } f\ opt = \textit{None}) = (opt = \textit{None})$
⟨*proof*⟩

lemma *None-eq-map-option-iff* [*iff*]: $\textit{None} = \textit{map-option } f\ x \longleftrightarrow x = \textit{None}$
⟨*proof*⟩

lemma *map-option-eq-Some* [*iff*]: $(\textit{map-option } f\ xo = \textit{Some } y) = (\exists z.\ xo = \textit{Some } z \wedge f\ z = y)$
⟨*proof*⟩

lemma *map-option-o-case-sum* [*simp*]:
$\textit{map-option } f \circ \textit{case-sum } g\ h = \textit{case-sum } (\textit{map-option } f \circ g)\ (\textit{map-option } f \circ h)$
⟨*proof*⟩

lemma *map-option-cong*: $x = y \Longrightarrow (\bigwedge a.\ y = \textit{Some } a \Longrightarrow f\ a = g\ a) \Longrightarrow \textit{map-option } f\ x = \textit{map-option } g\ y$
⟨*proof*⟩

lemma *map-option-idI*: $(\bigwedge y.\ y \in \textit{set-option } x \Longrightarrow f\ y = y) \Longrightarrow \textit{map-option } f\ x = x$
⟨*proof*⟩

functor *map-option*: *map-option*
⟨*proof*⟩

lemma *case-map-option* [*simp*]: $\textit{case-option } g\ h\ (\textit{map-option } f\ x) = \textit{case-option } g\ (h \circ f)\ x$
⟨*proof*⟩

lemma *None-notin-image-Some* [*simp*]: $\textit{None} \notin \textit{Some}\ `\ A$
⟨*proof*⟩

lemma *notin-range-Some*: $x \notin \textit{range Some} \longleftrightarrow x = \textit{None}$

⟨proof⟩

lemma *rel-option-iff*:
  $rel\text{-}option\ R\ x\ y = (case\ (x,\ y)\ of\ (None,\ None) \Rightarrow True$
    $|\ (Some\ x,\ Some\ y) \Rightarrow R\ x\ y$
    $|\ \text{-} \Rightarrow False)$
  ⟨proof⟩

definition $combine\text{-}options :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ option \Rightarrow 'a\ option \Rightarrow 'a\ option$
  where $combine\text{-}options\ f\ x\ y =$
    $(case\ x\ of\ None \Rightarrow y\ |\ Some\ x \Rightarrow (case\ y\ of\ None \Rightarrow Some\ x\ |\ Some\ y \Rightarrow Some\ (f\ x\ y)))$

lemma *combine-options-simps* [*simp*]:
  $combine\text{-}options\ f\ None\ y = y$
  $combine\text{-}options\ f\ x\ None = x$
  $combine\text{-}options\ f\ (Some\ a)\ (Some\ b) = Some\ (f\ a\ b)$
  ⟨proof⟩

lemma *combine-options-cases* [*case-names None1 None2 Some*]:
  $(x = None \Longrightarrow P\ x\ y) \Longrightarrow (y = None \Longrightarrow P\ x\ y) \Longrightarrow$
    $(\bigwedge a\ b.\ x = Some\ a \Longrightarrow y = Some\ b \Longrightarrow P\ x\ y) \Longrightarrow P\ x\ y$
  ⟨proof⟩

lemma *combine-options-commute*:
  $(\bigwedge x\ y.\ f\ x\ y = f\ y\ x) \Longrightarrow combine\text{-}options\ f\ x\ y = combine\text{-}options\ f\ y\ x$
  ⟨proof⟩

lemma *combine-options-assoc*:
  $(\bigwedge x\ y\ z.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)) \Longrightarrow$
    $combine\text{-}options\ f\ (combine\text{-}options\ f\ x\ y)\ z =$
    $combine\text{-}options\ f\ x\ (combine\text{-}options\ f\ y\ z)$
  ⟨proof⟩

lemma *combine-options-left-commute*:
  $(\bigwedge x\ y.\ f\ x\ y = f\ y\ x) \Longrightarrow (\bigwedge x\ y\ z.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)) \Longrightarrow$
    $combine\text{-}options\ f\ y\ (combine\text{-}options\ f\ x\ z) =$
    $combine\text{-}options\ f\ x\ (combine\text{-}options\ f\ y\ z)$
  ⟨proof⟩

lemmas *combine-options-ac* =
  combine-options-commute combine-options-assoc combine-options-left-commute

context
begin

qualified definition $is\text{-}none :: 'a\ option \Rightarrow bool$

where [*code-post*]: $\textit{is-none } x \longleftrightarrow x = \textit{None}$

lemma *is-none-simps* [*simp*]:
$\textit{is-none None}$
$\neg \textit{ is-none } (\textit{Some } x)$
⟨*proof*⟩

lemma *is-none-code* [*code*]:
$\textit{is-none None} = \textit{True}$
$\textit{is-none } (\textit{Some } x) = \textit{False}$
⟨*proof*⟩

lemma *rel-option-unfold*:
$\textit{rel-option } R\ x\ y \longleftrightarrow$
$(\textit{is-none } x \longleftrightarrow \textit{is-none } y) \wedge (\neg \textit{ is-none } x \longrightarrow \neg \textit{ is-none } y \longrightarrow R\ (\textit{the } x)\ (\textit{the } y))$
⟨*proof*⟩

lemma *rel-optionI*:
$[\![\textit{is-none } x \longleftrightarrow \textit{is-none } y; [\![\neg \textit{ is-none } x; \neg \textit{ is-none } y]\!] \Longrightarrow P\ (\textit{the } x)\ (\textit{the } y)]\!]$
$\Longrightarrow \textit{rel-option } P\ x\ y$
⟨*proof*⟩

lemma *is-none-map-option* [*simp*]: $\textit{is-none } (\textit{map-option } f\ x) \longleftrightarrow \textit{is-none } x$
⟨*proof*⟩

lemma *the-map-option*: $\neg \textit{ is-none } x \Longrightarrow \textit{the } (\textit{map-option } f\ x) = f\ (\textit{the } x)$
⟨*proof*⟩ **primrec** $\textit{bind} :: {}'a\ \textit{option} \Rightarrow ({}'a \Rightarrow {}'b\ \textit{option}) \Rightarrow {}'b\ \textit{option}$
where
bind-lzero: $\textit{bind None } f = \textit{None}$
| *bind-lunit*: $\textit{bind } (\textit{Some } x)\ f = f\ x$

lemma *is-none-bind*: $\textit{is-none } (\textit{bind } f\ g) \longleftrightarrow \textit{is-none } f \vee \textit{is-none } (g\ (\textit{the } f))$
⟨*proof*⟩

lemma *bind-runit*[*simp*]: $\textit{bind } x\ \textit{Some} = x$
⟨*proof*⟩

lemma *bind-assoc*[*simp*]: $\textit{bind } (\textit{bind } x\ f)\ g = \textit{bind } x\ (\lambda y.\ \textit{bind } (f\ y)\ g)$
⟨*proof*⟩

lemma *bind-rzero*[*simp*]: $\textit{bind } x\ (\lambda x.\ \textit{None}) = \textit{None}$
⟨*proof*⟩ **lemma** *bind-cong*: $x = y \Longrightarrow (\bigwedge a.\ y = \textit{Some } a \Longrightarrow f\ a = g\ a) \Longrightarrow \textit{bind } x\ f = \textit{bind } y\ g$
⟨*proof*⟩

lemma *bind-split*: $P\ (\textit{bind } m\ f) \longleftrightarrow (m = \textit{None} \longrightarrow P\ \textit{None}) \wedge (\forall v.\ m = \textit{Some } v \longrightarrow P\ (f\ v))$
⟨*proof*⟩

lemma *bind-split-asm*: P (*bind* m f) $\longleftrightarrow \neg$ (m = *None* $\wedge \neg$ P *None* $\vee$ ($\exists x$. m = *Some* x $\wedge \neg$ P (f x)))
⟨*proof*⟩

lemmas *bind-splits* = *bind-split bind-split-asm*

lemma *bind-eq-Some-conv*: *bind* f g = *Some* x $\longleftrightarrow$ ($\exists y$. f = *Some* y $\wedge$ g y = *Some* x)
⟨*proof*⟩

lemma *bind-eq-None-conv*: *Option.bind* a f = *None* $\longleftrightarrow$ a = *None* $\vee$ f (*the* a) = *None*
⟨*proof*⟩

lemma *map-option-bind*: *map-option* f (*bind* x g) = *bind* x (*map-option* $f \circ g$)
⟨*proof*⟩

lemma *bind-option-cong*:
$[\![$ $x = y$; $\bigwedge z$. $z \in$ *set-option* $y \Longrightarrow f\,z = g\,z$ $]\!] \Longrightarrow$ *bind* x f = *bind* y g
⟨*proof*⟩

lemma *bind-option-cong-simp*:
$[\![$ $x = y$; $\bigwedge z$. $z \in$ *set-option* y =*simp*=> $f\,z = g\,z$ $]\!] \Longrightarrow$ *bind* x f = *bind* y g
⟨*proof*⟩

lemma *bind-option-cong-code*: $x = y \Longrightarrow$ *bind* x f = *bind* y f
⟨*proof*⟩

lemma *bind-map-option*: *bind* (*map-option* f x) g = *bind* x ($g \circ f$)
⟨*proof*⟩

lemma *set-bind-option* [*simp*]: *set-option* (*bind* x f) = *UNION* (*set-option* x) (*set-option* $\circ f$)
⟨*proof*⟩

lemma *map-conv-bind-option*: *map-option* f x = *Option.bind* x (*Some* $\circ f$)
⟨*proof*⟩

end

⟨*ML*⟩

context
begin

qualified definition *these* :: *'a option set* $\Rightarrow$ *'a set*
where *these* A = *the* ‘ $\{x \in A.\ x \neq$ *None*$\}$

lemma *these-empty* [*simp*]: *these* {} = {}
⟨*proof*⟩

lemma *these-insert-None* [*simp*]: *these* (*insert None A*) = *these A*
⟨*proof*⟩

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

lemma *in-these-eq*: $x \in$ *these A* $\longleftrightarrow$ *Some* $x \in A$
⟨*proof*⟩

lemma *these-image-Some-eq* [*simp*]: *these* (*Some* ‘ *A*) = *A*
⟨*proof*⟩

lemma *Some-image-these-eq*: *Some* ‘ *these A* = $\{x \in A.\ x \neq None\}$
⟨*proof*⟩

lemma *these-empty-eq*: *these B* = {} $\longleftrightarrow$ *B* = {} $\vee$ *B* = {*None*}
⟨*proof*⟩

lemma *these-not-empty-eq*: *these B* $\neq$ {} $\longleftrightarrow$ *B* $\neq$ {} $\wedge$ *B* $\neq$ {*None*}
⟨*proof*⟩

end

48.1 Transfer rules for the Transfer package

context includes *lifting-syntax*
begin

lemma *option-bind-transfer* [*transfer-rule*]:
(*rel-option A* ===> (*A* ===> *rel-option B*) ===> *rel-option B*)
Option.bind Option.bind
⟨*proof*⟩

lemma *pred-option-parametric* [*transfer-rule*]:
((*A* ===> *op* =) ===> *rel-option A* ===> *op* =) *pred-option pred-option*
⟨*proof*⟩

end

48.1.1 Interaction with finite sets

lemma *finite-option-UNIV* [*simp*]:
finite (*UNIV* :: 'a *option set*) = *finite* (*UNIV* :: 'a *set*)
⟨*proof*⟩

instance *option* :: (*finite*) *finite*

⟨*proof*⟩

48.1.2 Code generator setup

lemma *equal-None-code-unfold* [*code-unfold*]:
HOL.equal x None ⟷ *Option.is-none x*
HOL.equal None = *Option.is-none*
⟨*proof*⟩

code-printing
type-constructor *option* ⇀
(*SML*) - *option*
and (*OCaml*) - *option*
and (*Haskell*) *Maybe* -
and (*Scala*) !*Option*[(-)]
| **constant** *None* ⇀
(*SML*) *NONE*
and (*OCaml*) *None*
and (*Haskell*) *Nothing*
and (*Scala*) !*None*
| **constant** *Some* ⇀
(*SML*) *SOME*
and (*OCaml*) *Some* -
and (*Haskell*) *Just*
and (*Scala*) *Some*
| **class-instance** *option* :: *equal* ⇀
(*Haskell*) −
| **constant** *HOL.equal* :: *'a option* ⇒ *'a option* ⇒ *bool* ⇀
(*Haskell*) **infix** *4* ==

code-reserved *SML*
option NONE SOME

code-reserved *OCaml*
option None Some

code-reserved *Scala*
Option None Some

end

49 Partial Function Definitions

theory *Partial-Function*
imports *Complete-Partial-Order Option*
keywords *partial-function* :: *thy-decl*
begin

named-theorems *partial-function-mono monotonicity rules for partial function*

definitions
⟨*ML*⟩

lemma (**in** *ccpo*) *in-chain-finite*:
  assumes *Complete-Partial-Order.chain op* $\leq$ *A finite A A* $\neq \{\}$
  shows $\bigsqcup A \in A$
⟨*proof*⟩

lemma (**in** *ccpo*) *admissible-chfin*:
  $(\forall S.$ *Complete-Partial-Order.chain op* $\leq S \longrightarrow$ *finite S*$)$
  $\Longrightarrow$ *ccpo.admissible Sup op* $\leq P$
⟨*proof*⟩

49.1 Axiomatic setup

This techical locale constains the requirements for function definitions with ccpo fixed points.

definition *fun-ord ord f g* $\longleftrightarrow (\forall x.\ ord\ (f\ x)\ (g\ x))$
definition *fun-lub L A* $= (\lambda x.\ L\ \{y.\ \exists f{\in}A.\ y = f\ x\})$
definition *img-ord f ord* $= (\lambda x\ y.\ ord\ (f\ x)\ (f\ y))$
definition *img-lub f g Lub* $= (\lambda A.\ g\ (Lub\ (f\ `\ A)))$

lemma *chain-fun*:
  assumes *A*: *chain* (*fun-ord ord*) *A*
  shows *chain ord* $\{y.\ \exists f{\in}A.\ y = f\ a\}$ (**is** *chain ord ?C*)
⟨*proof*⟩

lemma *call-mono*[*partial-function-mono*]: *monotone* (*fun-ord ord*) *ord* $(\lambda f.\ f\ t)$
⟨*proof*⟩

lemma *let-mono*[*partial-function-mono*]:
  $(\bigwedge x.$ *monotone orda ordb* $(\lambda f.\ b\ f\ x))$
  $\Longrightarrow$ *monotone orda ordb* $(\lambda f.\ Let\ t\ (b\ f))$
⟨*proof*⟩

lemma *if-mono*[*partial-function-mono*]: *monotone orda ordb F*
$\Longrightarrow$ *monotone orda ordb G*
$\Longrightarrow$ *monotone orda ordb* $(\lambda f.$ *if c then F f else G f*$)$
⟨*proof*⟩

definition *mk-less R* $= (\lambda x\ y.\ R\ x\ y \wedge \neg\ R\ y\ x)$

locale *partial-function-definitions* =
  fixes *leq* :: $'a \Rightarrow 'a \Rightarrow bool$
  fixes *lub* :: $'a\ set \Rightarrow 'a$
  assumes *leq-refl*: *leq x x*
  assumes *leq-trans*: *leq x y* $\Longrightarrow$ *leq y z* $\Longrightarrow$ *leq x z*
  assumes *leq-antisym*: *leq x y* $\Longrightarrow$ *leq y x* $\Longrightarrow x = y$
  assumes *lub-upper*: *chain leq A* $\Longrightarrow x \in A \Longrightarrow$ *leq x* (*lub A*)

assumes *lub-least*: *chain leq A* $\Longrightarrow$ ($\bigwedge x.\ x \in A \Longrightarrow$ *leq x z*) $\Longrightarrow$ *leq* (*lub A*) *z*

lemma *partial-function-lift*:
assumes *partial-function-definitions ord lb*
shows *partial-function-definitions* (*fun-ord ord*) (*fun-lub lb*) (**is** *partial-function-definitions ?ordf ?lubf*)
⟨*proof*⟩

lemma *ccpo*: **assumes** *partial-function-definitions ord lb*
shows *class.ccpo lb ord* (*mk-less ord*)
⟨*proof*⟩

lemma *partial-function-image*:
assumes *partial-function-definitions ord Lub*
assumes *inj*: $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
assumes *inv*: $\bigwedge x.\ f\ (g\ x) = x$
shows *partial-function-definitions* (*img-ord f ord*) (*img-lub f g Lub*)
⟨*proof*⟩

context *partial-function-definitions*
begin

abbreviation *le-fun* ≡ *fun-ord leq*
abbreviation *lub-fun* ≡ *fun-lub lub*
abbreviation *fixp-fun* ≡ *ccpo.fixp lub-fun le-fun*
abbreviation *mono-body* ≡ *monotone le-fun leq*
abbreviation *admissible* ≡ *ccpo.admissible lub-fun le-fun*

Interpret manually, to avoid flooding everything with facts about orders

lemma *ccpo*: *class.ccpo lub-fun le-fun* (*mk-less le-fun*)
⟨*proof*⟩

The crucial fixed-point theorem

lemma *mono-body-fixp*:
($\bigwedge x.$ *mono-body* ($\lambda f.\ F\ f\ x$)) $\Longrightarrow$ *fixp-fun* $F = F$ (*fixp-fun* F)
⟨*proof*⟩

Version with curry/uncurry combinators, to be used by package

lemma *fixp-rule-uc*:
fixes F :: $'c \Rightarrow 'c$ **and**
U :: $'c \Rightarrow 'b \Rightarrow 'a$ **and**
C :: $('b \Rightarrow 'a) \Rightarrow 'c$
assumes *mono*: $\bigwedge x.$ *mono-body* ($\lambda f.\ U\ (F\ (C\ f))\ x$)
assumes *eq*: $f \equiv C$ (*fixp-fun* ($\lambda f.\ U\ (F\ (C\ f))$))
assumes *inverse*: $\bigwedge f.\ C\ (U\ f) = f$
shows $f = F\ f$
⟨*proof*⟩

Fixpoint induction rule

lemma *fixp-induct-uc*:
  fixes $F :: 'c \Rightarrow 'c$
    and $U :: 'c \Rightarrow 'b \Rightarrow 'a$
    and $C :: ('b \Rightarrow 'a) \Rightarrow 'c$
    and $P :: ('b \Rightarrow 'a) \Rightarrow bool$
  assumes *mono*: $\bigwedge x.\ mono\text{-}body\ (\lambda f.\ U\ (F\ (C\ f))\ x)$
    and *eq*: $f \equiv C\ (fixp\text{-}fun\ (\lambda f.\ U\ (F\ (C\ f))))$
    and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
    and *adm*: *ccpo.admissible lub-fun le-fun P*
    and *bot*: $P\ (\lambda\text{-}.\ lub\ \{\})$
    and *step*: $\bigwedge f.\ P\ (U\ f) \Longrightarrow P\ (U\ (F\ f))$
  shows $P\ (U\ f)$
⟨*proof*⟩

Rules for *mono-body*:

lemma *const-mono*[*partial-function-mono*]: *monotone ord leq* $(\lambda f.\ c)$
⟨*proof*⟩

end

49.2 Flat interpretation: tailrec and option

definition
  flat-ord $b\ x\ y \longleftrightarrow x = b \lor x = y$

definition
  flat-lub $b\ A = (if\ A \subseteq \{b\}\ then\ b\ else\ (THE\ x.\ x \in A - \{b\}))$

lemma *flat-interpretation*:
  partial-function-definitions (*flat-ord* b) (*flat-lub* b)
⟨*proof*⟩

lemma *flat-ordI*: $(x \neq a \Longrightarrow x = y) \Longrightarrow$ *flat-ord* $a\ x\ y$
⟨*proof*⟩

lemma *flat-ord-antisym*: $\llbracket$ *flat-ord* $a\ x\ y$; *flat-ord* $a\ y\ x$ $\rrbracket \Longrightarrow x = y$
⟨*proof*⟩

lemma *antisymp-flat-ord*: *antisymp* (*flat-ord* a)
⟨*proof*⟩

interpretation *tailrec*:
  partial-function-definitions flat-ord undefined flat-lub undefined
  rewrites *flat-lub undefined* $\{\} \equiv$ *undefined*
⟨*proof*⟩

interpretation *option*:
  partial-function-definitions flat-ord None flat-lub None
  rewrites *flat-lub None* $\{\} \equiv$ *None*

⟨*proof*⟩

abbreviation *tailrec-ord* ≡ *flat-ord undefined*
abbreviation *mono-tailrec* ≡ *monotone* (*fun-ord tailrec-ord*) *tailrec-ord*

lemma *tailrec-admissible*:
ccpo.admissible (*fun-lub* (*flat-lub c*)) (*fun-ord* (*flat-ord c*))
(λ*a*. ∀ *x*. *a x* ≠ *c* ⟶ *P x* (*a x*))
⟨*proof*⟩

lemma *fixp-induct-tailrec*:
fixes *F* :: ′*c* ⇒ ′*c* **and**
U :: ′*c* ⇒ ′*b* ⇒ ′*a* **and**
C :: (′*b* ⇒ ′*a*) ⇒ ′*c* **and**
P :: ′*b* ⇒ ′*a* ⇒ *bool* **and**
x :: ′*b*
assumes *mono*: ⋀*x*. *monotone* (*fun-ord* (*flat-ord c*)) (*flat-ord c*) (λ*f*. *U* (*F* (*C* *f*)) *x*)
assumes *eq*: *f* ≡ *C* (*ccpo.fixp* (*fun-lub* (*flat-lub c*)) (*fun-ord* (*flat-ord c*)) (λ*f*. *U* (*F* (*C f*))))
assumes *inverse2*: ⋀*f*. *U* (*C f*) = *f*
assumes *step*: ⋀*f x y*. (⋀*x y*. *U f x* = *y* ⟹ *y* ≠ *c* ⟹ *P x y*) ⟹ *U* (*F f*) *x* = *y* ⟹ *y* ≠ *c* ⟹ *P x y*
assumes *result*: *U f x* = *y*
assumes *defined*: *y* ≠ *c*
shows *P x y*
⟨*proof*⟩

lemma *admissible-image*:
assumes *pfun*: *partial-function-definitions le lub*
assumes *adm*: *ccpo.admissible lub le* (*P o g*)
assumes *inj*: ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
assumes *inv*: ⋀*x*. *f* (*g x*) = *x*
shows *ccpo.admissible* (*img-lub f g lub*) (*img-ord f le*) *P*
⟨*proof*⟩

lemma *admissible-fun*:
assumes *pfun*: *partial-function-definitions le lub*
assumes *adm*: ⋀*x*. *ccpo.admissible lub le* (*Q x*)
shows *ccpo.admissible* (*fun-lub lub*) (*fun-ord le*) (λ*f*. ∀ *x*. *Q x* (*f x*))
⟨*proof*⟩

abbreviation *option-ord* ≡ *flat-ord None*
abbreviation *mono-option* ≡ *monotone* (*fun-ord option-ord*) *option-ord*

lemma *bind-mono*[*partial-function-mono*]:
assumes *mf*: *mono-option B* **and** *mg*: ⋀*y*. *mono-option* (λ*f*. *C y f*)

shows *mono-option* $(\lambda f.\ Option.bind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
⟨*proof*⟩

lemma *flat-lub-in-chain*:
assumes *ch*: *chain* (*flat-ord* b) A
assumes *lub*: *flat-lub* $b\ A = a$
shows $a = b \lor a \in A$
⟨*proof*⟩

lemma *option-admissible*: *option.admissible* $(\%(f::'a \Rightarrow 'b\ option).$
$(\forall x\ y.\ f\ x = Some\ y \longrightarrow P\ x\ y))$
⟨*proof*⟩

lemma *fixp-induct-option*:
fixes $F :: 'c \Rightarrow 'c$ **and**
$U :: 'c \Rightarrow 'b \Rightarrow 'a\ option$ **and**
$C :: ('b \Rightarrow 'a\ option) \Rightarrow 'c$ **and**
$P :: 'b \Rightarrow 'a \Rightarrow bool$
assumes *mono*: $\bigwedge x.\ mono\text{-}option\ (\lambda f.\ U\ (F\ (C\ f))\ x)$
assumes *eq*: $f \equiv C\ (ccpo.fixp\ (fun\text{-}lub\ (flat\text{-}lub\ None))\ (fun\text{-}ord\ option\text{-}ord)\ (\lambda f.\ U\ (F\ (C\ f))))$
assumes *inverse2*: $\bigwedge f.\ U\ (C\ f) = f$
assumes *step*: $\bigwedge f\ x\ y.\ (\bigwedge x\ y.\ U\ f\ x = Some\ y \Longrightarrow P\ x\ y) \Longrightarrow U\ (F\ f)\ x = Some\ y \Longrightarrow P\ x\ y$
assumes *defined*: $U\ f\ x = Some\ y$
shows $P\ x\ y$
⟨*proof*⟩

⟨*ML*⟩

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

end

theory *Argo*
imports *HOL*
begin

⟨*ML*⟩

end

50 Reconstructing external resolution proofs for propositional logic

theory *SAT*
imports *Argo*

begin

⟨*ML*⟩

end

51 Function Definitions and Termination Proofs

theory *Fun-Def*
  imports *Basic-BNF-LFPs Partial-Function SAT*
  keywords
    function termination :: *thy-goal* **and**
    fun fun-cases :: *thy-decl*
begin

51.1 Definitions with default value

definition *THE-default* :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$
  where $THE\text{-}default\ d\ P = (if\ (\exists!x.\ P\ x)\ then\ (THE\ x.\ P\ x)\ else\ d)$

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
  ⟨*proof*⟩

lemma *THE-default1-equality*: $\exists!x.\ P\ x \Longrightarrow P\ a \Longrightarrow THE\text{-}default\ d\ P = a$
  ⟨*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*: $f \equiv (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
  assumes *ex1*: $\exists!y.\ G\ x\ y$
  shows $G\ x\ (f\ x)$
  ⟨*proof*⟩

lemma *fundef-ex1-uniqueness*:
  assumes *f-def*: $f \equiv (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
  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*: $f \equiv (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
  assumes *ex1*: $\exists!y.\ G\ x\ y$
  shows $(G\ x\ y) = (f\ x = y)$
  ⟨*proof*⟩

lemma *fundef-default-value*:
  assumes *f-def*: $f \equiv (\lambda x::'a.\ \textit{THE-default}\ (d\ x)\ (\lambda y.\ G\ x\ y))$
  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*]: $\textit{in-rel}\ R\ x\ y \equiv (x,\ y) \in R$

lemma *wf-in-rel*: $\textit{wf}\ R \Longrightarrow \textit{wfP}\ (\textit{in-rel}\ R)$
  ⟨*proof*⟩

⟨*ML*⟩

51.2 Measure functions

inductive *is-measure* :: $('a \Rightarrow \textit{nat}) \Rightarrow \textit{bool}$
  where *is-measure-trivial*: *is-measure f*

named-theorems *measure-function rules that guide the heuristic generation of measure functions*
⟨*ML*⟩

lemma *measure-size*[*measure-function*]: *is-measure size*
  ⟨*proof*⟩

lemma *measure-fst*[*measure-function*]: $\textit{is-measure}\ f \Longrightarrow \textit{is-measure}\ (\lambda p.\ f\ (\textit{fst}\ p))$
  ⟨*proof*⟩

lemma *measure-snd*[*measure-function*]: $\textit{is-measure}\ f \Longrightarrow \textit{is-measure}\ (\lambda p.\ f\ (\textit{snd}\ p))$
  ⟨*proof*⟩

⟨*ML*⟩

51.3 Congruence rules

lemma *let-cong* [*fundef-cong*]: $M = N \Longrightarrow (\bigwedge x.\ x = N \Longrightarrow f\ x = g\ x) \Longrightarrow \textit{Let}\ M\ f = \textit{Let}\ N\ g$
  ⟨*proof*⟩

lemmas [*fundef-cong*] =
  if-cong image-cong INF-cong SUP-cong
  bex-cong ball-cong imp-cong map-option-cong Option.bind-cong

lemma *split-cong* [*fundef-cong*]:
  $(\bigwedge x\ y.\ (x,\ y) = q \Longrightarrow f\ x\ y = g\ x\ y) \Longrightarrow p = q \Longrightarrow \textit{case-prod}\ f\ p = \textit{case-prod}\ g\ q$
  ⟨*proof*⟩

lemma *comp-cong* [*fundef-cong*]: $f\ (g\ x) = f'\ (g'\ x') \Longrightarrow (f \circ g)\ x = (f' \circ g')\ x'$
⟨*proof*⟩

51.4 Simp rules for termination proofs

declare
trans-less-add1[*termination-simp*]
trans-less-add2[*termination-simp*]
trans-le-add1[*termination-simp*]
trans-le-add2[*termination-simp*]
less-imp-le-nat[*termination-simp*]
le-imp-less-Suc[*termination-simp*]

lemma *size-prod-simp*[*termination-simp*]: *size-prod f g p* = *f* (*fst p*) + *g* (*snd p*) + *Suc 0*
⟨*proof*⟩

51.5 Decomposition

lemma *less-by-empty*: $A = \{\} \Longrightarrow A \subseteq B$
and *union-comp-emptyL*: $A\ O\ C = \{\} \Longrightarrow B\ O\ C = \{\} \Longrightarrow (A \cup B)\ O\ C = \{\}$
and *union-comp-emptyR*: $A\ O\ B = \{\} \Longrightarrow A\ O\ C = \{\} \Longrightarrow A\ O\ (B \cup C) = \{\}$
and *wf-no-loop*: $R\ O\ R = \{\} \Longrightarrow$ *wf R*
⟨*proof*⟩

51.6 Reduction pairs

definition *reduction-pair P* $\longleftrightarrow$ *wf* (*fst P*) $\wedge$ *fst P O snd P* $\subseteq$ *fst P*

lemma *reduction-pairI*[*intro*]: *wf R* $\Longrightarrow$ $R\ O\ S \subseteq R \Longrightarrow$ *reduction-pair* (*R*, *S*)
⟨*proof*⟩

lemma *reduction-pair-lemma*:
assumes *rp*: *reduction-pair P*
assumes $R \subseteq$ *fst P*
assumes $S \subseteq$ *snd P*
assumes *wf S*
shows *wf* ($R \cup S$)
⟨*proof*⟩

definition *rp-inv-image* = (λ(*R*,*S*) *f*. (*inv-image R f*, *inv-image S f*))

lemma *rp-inv-image-rp*: *reduction-pair P* $\Longrightarrow$ *reduction-pair* (*rp-inv-image P f*)
⟨*proof*⟩

51.7 Concrete orders for SCNP termination proofs

definition *pair-less* = *less-than* <∗*lex*∗> *less-than*
definition *pair-leq* = *pair-less*^=
definition *max-strict* = *max-ext pair-less*

definition *max-weak* = *max-ext pair-leq* $\cup$ {({}, {})}
definition *min-strict* = *min-ext pair-less*
definition *min-weak* = *min-ext pair-leq* $\cup$ {({}, {})}

lemma *wf-pair-less*[*simp*]: *wf pair-less*
⟨*proof*⟩

Introduction rules for *pair-less*/*pair-leq*

lemma *pair-leqI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in$ *pair-leq*
and *pair-leqI2*: $a \leq b \Longrightarrow s \leq t \Longrightarrow ((a, s), (b, t)) \in$ *pair-leq*
and *pair-lessI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in$ *pair-less*
and *pair-lessI2*: $a \leq b \Longrightarrow s < t \Longrightarrow ((a, s), (b, t)) \in$ *pair-less*
⟨*proof*⟩

Introduction rules for max

lemma *smax-emptyI*: *finite* $Y \Longrightarrow Y \neq \{\} \Longrightarrow (\{\}, Y) \in$ *max-strict*
and *smax-insertI*:
$y \in Y \Longrightarrow (x, y) \in$ *pair-less* $\Longrightarrow (X, Y) \in$ *max-strict* $\Longrightarrow$ (*insert* x X, Y) $\in$ *max-strict*
and *wmax-emptyI*: *finite* $X \Longrightarrow (\{\}, X) \in$ *max-weak*
and *wmax-insertI*:
$y \in YS \Longrightarrow (x, y) \in$ *pair-leq* $\Longrightarrow (XS, YS) \in$ *max-weak* $\Longrightarrow$ (*insert* x XS, YS) $\in$ *max-weak*
⟨*proof*⟩

Introduction rules for min

lemma *smin-emptyI*: $X \neq \{\} \Longrightarrow (X, \{\}) \in$ *min-strict*
and *smin-insertI*:
$x \in XS \Longrightarrow (x, y) \in$ *pair-less* $\Longrightarrow (XS, YS) \in$ *min-strict* $\Longrightarrow$ (XS, *insert* y YS) $\in$ *min-strict*
and *wmin-emptyI*: $(X, \{\}) \in$ *min-weak*
and *wmin-insertI*:
$x \in XS \Longrightarrow (x, y) \in$ *pair-leq* $\Longrightarrow (XS, YS) \in$ *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*⟩

51.8 Yet another induction principle on the natural numbers

lemma *nat-descend-induct* [*case-names base descend*]:
fixes $P :: nat \Rightarrow bool$
assumes *H1*: $\bigwedge k.\ k > n \Longrightarrow P\ k$
assumes *H2*: $\bigwedge k.\ k \leq n \Longrightarrow (\bigwedge i.\ i > k \Longrightarrow P\ i) \Longrightarrow P\ k$
shows $P\ m$
⟨*proof*⟩

51.9 Tool setup

⟨*ML*⟩

end

52 The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *Int*
imports *Equiv-Relations Power Quotient Fun-Def*
begin

52.1 Definition of integers as a quotient type

definition $intrel :: (nat \times nat) \Rightarrow (nat \times nat) \Rightarrow bool$
where $intrel = (\lambda(x, y)\ (u, v).\ x + v = u + y)$

lemma *intrel-iff* [*simp*]: $intrel\ (x, y)\ (u, v) \longleftrightarrow x + v = u + y$
⟨*proof*⟩

quotient-type $int = nat \times nat\ /\ intrel$
morphisms *Rep-Integ Abs-Integ*
⟨*proof*⟩

lemma *eq-Abs-Integ* [*case-names Abs-Integ*, *cases type*: *int*]:
$(\bigwedge x\ y.\ z = Abs\text{-}Integ\ (x, y) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

52.2 Integers form a commutative ring

instantiation *int* :: *comm-ring-1*
begin

lift-definition *zero-int* :: *int* **is** (*0*, *0*) ⟨*proof*⟩

lift-definition *one-int* :: *int* **is** $(1,\ 0)$ ⟨*proof*⟩

lift-definition *plus-int* :: *int* ⇒ *int* ⇒ *int*
is $\lambda(x,\ y)\ (u,\ v).\ (x + u,\ y + v)$
⟨*proof*⟩

lift-definition *uminus-int* :: *int* ⇒ *int*
is $\lambda(x,\ y).\ (y,\ x)$
⟨*proof*⟩

lift-definition *minus-int* :: *int* ⇒ *int* ⇒ *int*
is $\lambda(x,\ y)\ (u,\ v).\ (x + v,\ y + u)$
⟨*proof*⟩

lift-definition *times-int* :: *int* ⇒ *int* ⇒ *int*
is $\lambda(x,\ y)\ (u,\ v).\ (x*u + y*v,\ x*v + y*u)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

abbreviation *int* :: *nat* ⇒ *int*
where *int* ≡ *of-nat*

lemma *int-def*: *int* n = *Abs-Integ* $(n,\ 0)$
⟨*proof*⟩

lemma *int-transfer* [*transfer-rule*]: (*rel-fun* (*op* =) *pcr-int*) $(\lambda n.\ (n,\ 0))$ *int*
⟨*proof*⟩

lemma *int-diff-cases*: **obtains** (*diff*) m n **where** z = *int* m − *int* n
⟨*proof*⟩

52.3 Integers are totally ordered

instantiation *int* :: *linorder*
begin

lift-definition *less-eq-int* :: *int* ⇒ *int* ⇒ *bool*
is $\lambda(x,\ y)\ (u,\ v).\ x + v \leq u + y$
⟨*proof*⟩

lift-definition *less-int* :: *int* ⇒ *int* ⇒ *bool*
is $\lambda(x,\ y)\ (u,\ v).\ x + v < u + y$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *int* :: *distrib-lattice*
begin

definition $(inf :: int \Rightarrow int \Rightarrow int) = min$

definition $(sup :: int \Rightarrow int \Rightarrow int) = max$

instance
⟨*proof*⟩

end

52.4 Ordering properties of arithmetic operations

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*: $i < j \Longrightarrow 0 < k \Longrightarrow int\ k * i < int\ k * j$
for $i\ j :: int$
⟨*proof*⟩

lemma *zero-le-imp-eq-int*: $0 \leq k \Longrightarrow \exists\, n.\ k = int\ n$
for $k :: int$
⟨*proof*⟩

lemma *zero-less-imp-eq-int*: $0 < k \Longrightarrow \exists\, n{>}0.\ k = int\ n$
for $k :: int$
⟨*proof*⟩

lemma *zmult-zless-mono2*: $i < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$
for $i\ j\ k :: int$
⟨*proof*⟩

The integers form an ordered integral domain.

instantiation *int* :: *linordered-idom*
begin

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

lemma *zless-imp-add1-zle*: $w < z \Longrightarrow w + 1 \leq z$
  for $w\ z$ :: *int*
  ⟨*proof*⟩

lemma *zless-iff-Suc-zadd*: $w < z \longleftrightarrow (\exists n.\ z = w + int\ (Suc\ n))$
  for $w\ z$ :: *int*
  ⟨*proof*⟩

lemma *zabs-less-one-iff* [*simp*]: $|z| < 1 \longleftrightarrow z = 0$ (**is** *?lhs* ⟷ *?rhs*)
  for z :: *int*
⟨*proof*⟩

lemmas *int-distrib* =
  distrib-right [*of z1 z2 w*]
  distrib-left [*of w z1 z2*]
  left-diff-distrib [*of z1 z2 w*]
  right-diff-distrib [*of w z1 z2*]
  for *z1 z2 w* :: *int*

52.5 Embedding of the Integers into any *ring-1*: *of-int*

context *ring-1*
begin

lift-definition *of-int* :: *int* ⇒ *'a*
  is $\lambda(i, j).$ *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*⟩

lemma *mult-of-int-commute*: *of-int* $x * y = y *$ *of-int* x
⟨*proof*⟩

Collapse nested embeddings.

lemma *of-int-of-nat-eq* [*simp*]: *of-int* (*int* n) = *of-nat* n
⟨*proof*⟩

lemma *of-int-numeral* [*simp*, *code-post*]: *of-int* (*numeral* k) = *numeral* k
⟨*proof*⟩

lemma *of-int-neg-numeral* [*code-post*]: *of-int* (− *numeral* k) = − *numeral* k
⟨*proof*⟩

lemma *of-int-power* [*simp*]: *of-int* (z ^ n) = *of-int* z ^ n
⟨*proof*⟩

end

context *ring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]: *of-int* w = *of-int* $z \longleftrightarrow w = z$
⟨*proof*⟩

Special cases where either operand is zero.

lemma *of-int-eq-0-iff* [*simp*]: *of-int* $z = 0 \longleftrightarrow z = 0$
⟨*proof*⟩

lemma *of-int-0-eq-iff* [*simp*]: 0 = *of-int* $z \longleftrightarrow z = 0$
⟨*proof*⟩

lemma *of-int-eq-1-iff* [*iff*]: *of-int* $z = 1 \longleftrightarrow z = 1$
⟨*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* $w \leq$ *of-int* $z \longleftrightarrow w \leq z$
⟨*proof*⟩

lemma *of-int-less-iff* [*simp*]: *of-int* $w <$ *of-int* $z \longleftrightarrow w < z$
⟨*proof*⟩

lemma *of-int-0-le-iff* [*simp*]: $0 \leq$ *of-int* $z \longleftrightarrow 0 \leq z$
⟨*proof*⟩

lemma *of-int-le-0-iff* [*simp*]: *of-int* $z \leq 0 \longleftrightarrow z \leq 0$
⟨*proof*⟩

lemma *of-int-0-less-iff* [*simp*]: $0 <$ *of-int* $z \longleftrightarrow 0 < z$
⟨*proof*⟩

lemma *of-int-less-0-iff* [*simp*]: *of-int* $z < 0 \longleftrightarrow z < 0$
⟨*proof*⟩

lemma *of-int-1-le-iff* [*simp*]: $1 \leq$ *of-int* $z \longleftrightarrow 1 \leq z$
⟨*proof*⟩

lemma *of-int-le-1-iff* [*simp*]: *of-int* $z \leq 1 \longleftrightarrow z \leq 1$
⟨*proof*⟩

lemma *of-int-1-less-iff* [*simp*]: $1 <$ *of-int* $z \longleftrightarrow 1 < z$
⟨*proof*⟩

lemma *of-int-less-1-iff* [*simp*]: *of-int* $z < 1 \longleftrightarrow z < 1$
⟨*proof*⟩

lemma *of-int-pos*: $z > 0 \Longrightarrow$ *of-int* $z > 0$
⟨*proof*⟩

lemma *of-int-nonneg*: $z \geq 0 \Longrightarrow$ *of-int* $z \geq 0$
⟨*proof*⟩

lemma *of-int-abs* [*simp*]: *of-int* $|x|$ = |*of-int* x|
⟨*proof*⟩

lemma *of-int-lessD*:
assumes |*of-int* n| $< x$
shows $n = 0 \lor x > 1$
⟨*proof*⟩

lemma *of-int-leD*:
assumes |*of-int* n| $\leq x$
shows $n = 0 \lor 1 \leq x$
⟨*proof*⟩

end

Comparisons involving *of-int*.

lemma *of-int-eq-numeral-iff* [*iff*]: *of-int* z = (*numeral* n :: *'a::ring-char-0*) $\longleftrightarrow z$ = *numeral* n

⟨*proof*⟩

lemma *of-int-le-numeral-iff* [*simp*]:
$\mathit{of\text{-}int}\ z \le (\mathit{numeral}\ n :: {'a}::\mathit{linordered\text{-}idom}) \longleftrightarrow z \le \mathit{numeral}\ n$
⟨*proof*⟩

lemma *of-int-numeral-le-iff* [*simp*]:
$(\mathit{numeral}\ n :: {'a}::\mathit{linordered\text{-}idom}) \le \mathit{of\text{-}int}\ z \longleftrightarrow \mathit{numeral}\ n \le z$
⟨*proof*⟩

lemma *of-int-less-numeral-iff* [*simp*]:
$\mathit{of\text{-}int}\ z < (\mathit{numeral}\ n :: {'a}::\mathit{linordered\text{-}idom}) \longleftrightarrow z < \mathit{numeral}\ n$
⟨*proof*⟩

lemma *of-int-numeral-less-iff* [*simp*]:
$(\mathit{numeral}\ n :: {'a}::\mathit{linordered\text{-}idom}) < \mathit{of\text{-}int}\ z \longleftrightarrow \mathit{numeral}\ n < z$
⟨*proof*⟩

lemma *of-nat-less-of-int-iff*: $(\mathit{of\text{-}nat}\ n::{'a}::\mathit{linordered\text{-}idom}) < \mathit{of\text{-}int}\ x \longleftrightarrow \mathit{int}\ n < x$
⟨*proof*⟩

lemma *of-int-eq-id* [*simp*]: $\mathit{of\text{-}int} = \mathit{id}$
⟨*proof*⟩

instance *int* :: *no-top*
⟨*proof*⟩

instance *int* :: *no-bot*
⟨*proof*⟩

52.6 Magnitude of an Integer, as a Natural Number: *nat*

lift-definition *nat* :: *int* ⇒ *nat* **is** $\lambda(x, y).\ x - y$
⟨*proof*⟩

lemma *nat-int* [*simp*]: $\mathit{nat}\ (\mathit{int}\ n) = n$
⟨*proof*⟩

lemma *int-nat-eq* [*simp*]: $\mathit{int}\ (\mathit{nat}\ z) = (\mathit{if}\ 0 \le z\ \mathit{then}\ z\ \mathit{else}\ 0)$
⟨*proof*⟩

lemma *nat-0-le*: $0 \le z \Longrightarrow \mathit{int}\ (\mathit{nat}\ z) = z$
⟨*proof*⟩

lemma *nat-le-0* [*simp*]: $z \le 0 \Longrightarrow \mathit{nat}\ z = 0$
⟨*proof*⟩

lemma *nat-le-eq-zle*: $0 < w \lor 0 \le z \Longrightarrow \mathit{nat}\ w \le \mathit{nat}\ z \longleftrightarrow w \le z$

⟨*proof*⟩

An alternative condition is $(0::'a) \leq w$.

lemma *nat-mono-iff*: $0 < z \Longrightarrow nat\ w < nat\ z \longleftrightarrow w < z$
⟨*proof*⟩

lemma *nat-less-eq-zless*: $0 \leq w \Longrightarrow nat\ w < nat\ z \longleftrightarrow w < z$
⟨*proof*⟩

lemma *zless-nat-conj* [*simp*]: $nat\ w < nat\ z \longleftrightarrow 0 < z \wedge w < z$
⟨*proof*⟩

lemma *nonneg-int-cases*:
assumes $0 \leq k$
obtains n **where** $k = int\ n$
⟨*proof*⟩

lemma *pos-int-cases*:
assumes $0 < k$
obtains n **where** $k = int\ n$ **and** $n > 0$
⟨*proof*⟩

lemma *nonpos-int-cases*:
assumes $k \leq 0$
obtains n **where** $k = -\ int\ n$
⟨*proof*⟩

lemma *neg-int-cases*:
assumes $k < 0$
obtains n **where** $k = -\ int\ n$ **and** $n > 0$
⟨*proof*⟩

lemma *nat-eq-iff*: $nat\ w = m \longleftrightarrow (if\ 0 \leq w\ then\ w = int\ m\ else\ m = 0)$
⟨*proof*⟩

lemma *nat-eq-iff2*: $m = nat\ w \longleftrightarrow (if\ 0 \leq w\ then\ w = int\ m\ else\ m = 0)$
⟨*proof*⟩

lemma *nat-0* [*simp*]: $nat\ 0 = 0$
⟨*proof*⟩

lemma *nat-1* [*simp*]: $nat\ 1 = Suc\ 0$
⟨*proof*⟩

lemma *nat-numeral* [*simp*]: $nat\ (numeral\ k) = numeral\ k$
⟨*proof*⟩

lemma *nat-neg-numeral* [*simp*]: $nat\ (-\ numeral\ k) = 0$
⟨*proof*⟩

lemma *nat-2*: $nat\ 2 = Suc\ (Suc\ 0)$
⟨*proof*⟩

lemma *nat-less-iff*: $0 \le w \Longrightarrow nat\ w < m \longleftrightarrow w < of\text{-}nat\ m$
⟨*proof*⟩

lemma *nat-le-iff*: $nat\ x \le n \longleftrightarrow x \le int\ n$
⟨*proof*⟩

lemma *nat-mono*: $x \le y \Longrightarrow nat\ x \le nat\ y$
⟨*proof*⟩

lemma *nat-0-iff* [*simp*]: $nat\ i = 0 \longleftrightarrow i \le 0$
for $i :: int$
⟨*proof*⟩

lemma *int-eq-iff*: $of\text{-}nat\ m = z \longleftrightarrow m = nat\ z \land 0 \le z$
⟨*proof*⟩

lemma *zero-less-nat-eq* [*simp*]: $0 < nat\ z \longleftrightarrow 0 < z$
⟨*proof*⟩

lemma *nat-add-distrib*: $0 \le z \Longrightarrow 0 \le z' \Longrightarrow nat\ (z + z') = nat\ z + nat\ z'$
⟨*proof*⟩

lemma *nat-diff-distrib'*: $0 \le x \Longrightarrow 0 \le y \Longrightarrow nat\ (x - y) = nat\ x - nat\ y$
⟨*proof*⟩

lemma *nat-diff-distrib*: $0 \le z' \Longrightarrow z' \le z \Longrightarrow nat\ (z - z') = nat\ z - nat\ z'$
⟨*proof*⟩

lemma *nat-zminus-int* [*simp*]: $nat\ (-\ int\ n) = 0$
⟨*proof*⟩

lemma *le-nat-iff*: $k \ge 0 \Longrightarrow n \le nat\ k \longleftrightarrow int\ n \le k$
⟨*proof*⟩

lemma *zless-nat-eq-int-zless*: $m < nat\ z \longleftrightarrow int\ m < z$
⟨*proof*⟩

lemma (**in** *ring-1*) *of-nat-nat* [*simp*]: $0 \le z \Longrightarrow of\text{-}nat\ (nat\ z) = of\text{-}int\ z$
⟨*proof*⟩

lemma *diff-nat-numeral* [*simp*]: $(numeral\ v :: nat) - numeral\ v' = nat\ (numeral\ v - numeral\ v')$
⟨*proof*⟩

For termination proofs:

lemma *measure-function-int*[*measure-function*]: *is-measure* (*nat* ∘ *abs*) ⟨*proof*⟩

52.7 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: $-$ (*int* (*Suc n*)) < (*0* :: *int*)
⟨*proof*⟩

lemma *negative-zless* [*iff*]: $-$ (*int* (*Suc n*)) < *int m*
⟨*proof*⟩

lemma *negative-zle-0*: $-$ *int n* ≤ *0*
⟨*proof*⟩

lemma *negative-zle* [*iff*]: $-$ *int n* ≤ *int m*
⟨*proof*⟩

lemma *not-zle-0-negative* [*simp*]: ¬ *0* ≤ $-$ *int* (*Suc n*)
⟨*proof*⟩

lemma *int-zle-neg*: *int n* ≤ $-$ *int m* ⟷ *n* = *0* ∧ *m* = *0*
⟨*proof*⟩

lemma *not-int-zless-negative* [*simp*]: ¬ *int n* < $-$ *int m*
⟨*proof*⟩

lemma *negative-eq-positive* [*simp*]: $-$ *int n* = *of-nat m* ⟷ *n* = *0* ∧ *m* = *0*
⟨*proof*⟩

lemma *zle-iff-zadd*: *w* ≤ *z* ⟷ (∃ *n*. *z* = *w* + *int n*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *zadd-int-left*: *int m* + (*int n* + *z*) = *int* (*m* + *n*) + *z*
⟨*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* |*a*| ⟷ (*0* ≤ *a* ⟶ *P a*) ∧ (*a* < *0* ⟶ *P* ($-$ *a*))
for *a* :: '*a*::*linordered-idom*
⟨*proof*⟩

lemma *negD*: *x* < *0* ⟹ ∃ *n*. *x* = $-$ (*int* (*Suc n*))
⟨*proof*⟩

52.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

This version is symmetric in the two subgoals.

lemma *int-cases2* [*case-names nonneg nonpos*, *cases type*: *int*]:
$(\bigwedge n.\ z = int\ n \Longrightarrow P) \Longrightarrow (\bigwedge n.\ z = -\ (int\ n) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

This is the default, with a negative case.

lemma *int-cases* [*case-names nonneg neg*, *cases type*: *int*]:
$(\bigwedge n.\ z = int\ n \Longrightarrow P) \Longrightarrow (\bigwedge n.\ z = -\ (int\ (Suc\ n)) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *int-cases3* [*case-names zero pos neg*]:
fixes $k :: int$
assumes $k = 0 \Longrightarrow P$ **and** $\bigwedge n.\ k = int\ n \Longrightarrow n > 0 \Longrightarrow P$
and $\bigwedge n.\ k = -\ int\ n \Longrightarrow n > 0 \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *int-of-nat-induct* [*case-names nonneg neg*, *induct type*: *int*]:
$(\bigwedge n.\ P\ (int\ n)) \Longrightarrow (\bigwedge n.\ P\ (-\ (int\ (Suc\ n)))) \Longrightarrow P\ z$
⟨*proof*⟩

lemma *Let-numeral* [*simp*]: $Let\ (numeral\ v)\ f = f\ (numeral\ v)$
— Unfold all *let*s involving constants
⟨*proof*⟩

lemma *Let-neg-numeral* [*simp*]: $Let\ (-\ numeral\ v)\ f = f\ (-\ numeral\ v)$
— Unfold all *let*s involving constants
⟨*proof*⟩

Unfold *min* and *max* on numerals.

lemmas *max-number-of* [*simp*] =
max-def [*of numeral u numeral v*]
max-def [*of numeral u − numeral v*]
max-def [*of − numeral u numeral v*]
max-def [*of − numeral u − numeral v*] **for** *u v*

lemmas *min-number-of* [*simp*] =
min-def [*of numeral u numeral v*]
min-def [*of numeral u − numeral v*]
min-def [*of − numeral u numeral v*]
min-def [*of − numeral u − numeral v*] **for** *u v*

52.8.1 Binary comparisons

Preliminaries

lemma *le-imp-0-less*:
  fixes $z :: int$
  assumes *le*: $0 \leq z$
  shows $0 < 1 + z$
⟨*proof*⟩

lemma *odd-less-0-iff*: $1 + z + z < 0 \longleftrightarrow z < 0$
  for $z :: int$
⟨*proof*⟩

52.8.2 Comparisons, for Ordered Rings

lemmas *double-eq-0-iff* = *double-zero*

lemma *odd-nonzero*: $1 + z + z \neq 0$
  for $z :: int$
⟨*proof*⟩

52.9 The Set of Integers

context *ring-1*
begin

definition *Ints* :: *'a set* ($\mathbb{Z}$)
  where $\mathbb{Z}$ = *range of-int*

lemma *Ints-of-int* [*simp*]: *of-int* $z \in \mathbb{Z}$
  ⟨*proof*⟩

lemma *Ints-of-nat* [*simp*]: *of-nat* $n \in \mathbb{Z}$
  ⟨*proof*⟩

lemma *Ints-0* [*simp*]: $0 \in \mathbb{Z}$
  ⟨*proof*⟩

lemma *Ints-1* [*simp*]: $1 \in \mathbb{Z}$
  ⟨*proof*⟩

lemma *Ints-numeral* [*simp*]: *numeral* $n \in \mathbb{Z}$
  ⟨*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*]: $a \in \mathbb{Z} \Longrightarrow a \mathbin{\hat{}} n \in \mathbb{Z}$
⟨*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*⟩

lemma *Nats-subset-Ints*: $\mathbb{N} \subseteq \mathbb{Z}$
⟨*proof*⟩

lemma *Nats-altdef1*: $\mathbb{N} = \{\textit{of-int } n \mid n.\ n \geq 0\}$
⟨*proof*⟩

end

lemma (**in** *linordered-idom*) *Ints-abs* [*simp*]:
shows $a \in \mathbb{Z} \Longrightarrow \textit{abs } a \in \mathbb{Z}$
⟨*proof*⟩

lemma (**in** *linordered-idom*) *Nats-altdef2*: $\mathbb{N} = \{n \in \mathbb{Z}.\ n \geq 0\}$
⟨*proof*⟩

lemma (**in** *idom-divide*) *of-int-divide-in-Ints*:
$\textit{of-int } a \textit{ div of-int } b \in \mathbb{Z}$ **if** $b \textit{ dvd } a$
⟨*proof*⟩

The premise involving $\mathbb{Z}$ prevents $a = (1::'a) / (2::'a)$.

lemma *Ints-double-eq-0-iff*:
fixes $a :: 'a::\textit{ring-char-0}$
assumes *in-Ints*: $a \in \mathbb{Z}$
shows $a + a = 0 \longleftrightarrow a = 0$
(**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

lemma *Ints-odd-nonzero*:
fixes $a :: 'a::\textit{ring-char-0}$
assumes *in-Ints*: $a \in \mathbb{Z}$

shows $1 + a + a \neq 0$
⟨*proof*⟩

lemma *Nats-numeral* [*simp*]: *numeral* $w \in \mathbb{N}$
⟨*proof*⟩

lemma *Ints-odd-less-0*:
fixes a :: $'a$::*linordered-idom*
assumes *in-Ints*: $a \in \mathbb{Z}$
shows $1 + a + a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

52.10 *sum* and *prod*

lemma *of-nat-sum* [*simp*]: *of-nat* (*sum f A*) = ($\sum x{\in}A$. *of-nat*(*f x*))
⟨*proof*⟩

lemma *of-int-sum* [*simp*]: *of-int* (*sum f A*) = ($\sum x{\in}A$. *of-int*(*f x*))
⟨*proof*⟩

lemma *of-nat-prod* [*simp*]: *of-nat* (*prod f A*) = ($\prod x{\in}A$. *of-nat*(*f x*))
⟨*proof*⟩

lemma *of-int-prod* [*simp*]: *of-int* (*prod f A*) = ($\prod x{\in}A$. *of-int*(*f x*))
⟨*proof*⟩

Legacy theorems

lemmas *int-sum* = *of-nat-sum* [**where** *'a=int*]
lemmas *int-prod* = *of-nat-prod* [**where** *'a=int*]
lemmas *zle-int* = *of-nat-le-iff* [**where** *'a=int*]
lemmas *int-int-eq* = *of-nat-eq-iff* [**where** *'a=int*]
lemmas *nonneg-eq-int* = *nonneg-int-cases*

52.11 Setting up simplification procedures

lemmas *of-int-simps* =
of-int-0 of-int-1 of-int-add of-int-mult

⟨*ML*⟩

52.12 More Inequality Reasoning

lemma *zless-add1-eq*: $w < z + 1 \longleftrightarrow w < z \lor w = z$
for *w z* :: *int*
⟨*proof*⟩

lemma *add1-zle-eq*: $w + 1 \leq z \longleftrightarrow w < z$
for *w z* :: *int*
⟨*proof*⟩

lemma *zle-diff1-eq* [*simp*]: $w \leq z - 1 \longleftrightarrow w < z$
  for $w\ z$ :: *int*
  ⟨*proof*⟩

lemma *zle-add1-eq-le* [*simp*]: $w < z + 1 \longleftrightarrow w \leq z$
  for $w\ z$ :: *int*
  ⟨*proof*⟩

lemma *int-one-le-iff-zero-less*: $1 \leq z \longleftrightarrow 0 < z$
  for z :: *int*
  ⟨*proof*⟩

lemma *Ints-nonzero-abs-ge1*:
  fixes x:: $'a$ :: *linordered-idom*
    assumes $x \in Ints\ x \neq 0$
    shows $1 \leq abs\ x$
⟨*proof*⟩

lemma *Ints-nonzero-abs-less1*:
  fixes x:: $'a$ :: *linordered-idom*
  shows $[\![x \in Ints;\ abs\ x < 1]\!] \Longrightarrow x = 0$
    ⟨*proof*⟩

52.13 The functions *nat* and *int*

Simplify the term $w + - z$.

lemma *one-less-nat-eq* [*simp*]: $Suc\ 0 < nat\ z \longleftrightarrow 1 < z$
  ⟨*proof*⟩

This simplifies expressions of the form $int\ n = z$ where z is an integer literal.

lemmas *int-eq-iff-numeral* [*simp*] = *int-eq-iff* [*of - numeral v*] **for** v

lemma *split-nat* [*arith-split*]: $P\ (nat\ i) = ((\forall n.\ i = int\ n \longrightarrow P\ n) \wedge (i < 0 \longrightarrow P\ 0))$
  (**is** $?P = (?L \wedge ?R)$)
  for i :: *int*
⟨*proof*⟩

lemma *nat-abs-int-diff*: $nat\ |int\ a - int\ b| = (if\ a \leq b\ then\ b - a\ else\ a - b)$
  ⟨*proof*⟩

lemma *nat-int-add*: $nat\ (int\ a + int\ b) = a + b$
  ⟨*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 *transfer-rule-of-int*:
  fixes R :: *'a::ring-1* ⇒ *'b::ring-1* ⇒ *bool*
  assumes [*transfer-rule*]: R *0 0* R *1 1*
    rel-fun R (rel-fun R R) plus plus
    rel-fun R R uminus uminus
  shows *rel-fun HOL.eq R of-int of-int*
⟨*proof*⟩

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*: $z \leq 0 \Longrightarrow$ *nat* $(z * z')$ = *nat* $(- z)$ $*$ *nat* $(- z')$
  for z z' :: *int*
  ⟨*proof*⟩

lemma *nat-abs-mult-distrib*: *nat* $|w * z|$ = *nat* $|w|$ $*$ *nat* $|z|$
  ⟨*proof*⟩

lemma *int-in-range-abs* [*simp*]: *int* $n \in$ *range abs*
⟨*proof*⟩

lemma *range-abs-Nats* [*simp*]: *range abs* = ($\mathbb{N}$ :: *int set*)
⟨*proof*⟩

lemma *Suc-nat-eq-nat-zadd1*: $0 \leq z \Longrightarrow$ *Suc* (*nat* z) = *nat* $(1 + z)$
  for z :: *int*
  ⟨*proof*⟩

lemma *diff-nat-eq-if*:
  nat z − *nat* z' =
    (*if* $z' < 0$ *then nat* z
     else
      let $d = z - z'$
      in if $d < 0$ *then* 0 *else nat* d)
  ⟨*proof*⟩

lemma *nat-numeral-diff-1* [*simp*]: *numeral* v − (*1::nat*) = *nat* (*numeral* $v - 1$)
  ⟨*proof*⟩

52.14 Induction principles for int

Well-founded segments of the integers.

definition *int-ge-less-than* :: *int* $\Rightarrow$ (*int* $\times$ *int*) *set*
 where *int-ge-less-than* $d = \{(z', z).\ d \leq z' \wedge z' < z\}$

lemma *wf-int-ge-less-than*: *wf* (*int-ge-less-than* d)
⟨*proof*⟩

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition *int-ge-less-than2* :: *int* $\Rightarrow$ (*int* $\times$ *int*) *set*
 where *int-ge-less-than2* $d = \{(z',z).\ d \leq z \wedge z' < z\}$

lemma *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2* d)
⟨*proof*⟩

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*]:
 fixes $i\ k$ :: *int*
 assumes *gr*: $k < i$
  and *base*: $P\ (k + 1)$
  and *step*: $\bigwedge i.\ k < i \Longrightarrow P\ i \Longrightarrow P\ (i + 1)$
 shows $P\ i$
 ⟨*proof*⟩

theorem *int-le-induct* [*consumes 1*, *case-names base step*]:
 fixes $i\ k$ :: *int*
 assumes *le*: $i \leq k$
  and *base*: $P\ k$
  and *step*: $\bigwedge i.\ i \leq k \Longrightarrow P\ i \Longrightarrow P\ (i - 1)$
 shows $P\ i$
⟨*proof*⟩

theorem *int-less-induct* [*consumes 1*, *case-names base step*]:
 fixes $i\ k$ :: *int*
 assumes *less*: $i < k$
  and *base*: $P\ (k - 1)$
  and *step*: $\bigwedge i.\ i < k \Longrightarrow P\ i \Longrightarrow P\ (i - 1)$
 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*⟩

52.15 Intermediate value theorems

lemma *int-val-lemma*: $(\forall\, i<n.\ |f\ (i + 1) - f\ i| \leq 1) \longrightarrow f\ 0 \leq k \longrightarrow k \leq f\ n \longrightarrow (\exists\, i \leq n.\ f\ i = k)$
for n :: *nat* **and** k :: *int*
⟨*proof*⟩

lemmas *nat0-intermed-int-val* = *int-val-lemma* [*rule-format* (*no-asm*)]

lemma *nat-intermed-int-val*:
$\forall\, i.\ m \leq i \wedge i < n \longrightarrow |f\ (i + 1) - f\ i| \leq 1 \Longrightarrow m < n \Longrightarrow$
$f\ m \leq k \Longrightarrow k \leq f\ n \Longrightarrow \exists\, i.\ m \leq i \wedge i \leq n \wedge f\ i = k$
for f :: $nat \Rightarrow int$ **and** k :: *int*
⟨*proof*⟩

52.16 Products and 1, by T. M. Rasmussen

lemma *abs-zmult-eq-1*:
fixes $m\ n$ :: *int*
assumes *mn*: $|m * n| = 1$
shows $|m| = 1$
⟨*proof*⟩

lemma *pos-zmult-eq-1-iff-lemma*: $m * n = 1 \Longrightarrow m = 1 \vee m = -\ 1$
for $m\ n$ :: *int*
⟨*proof*⟩

lemma *pos-zmult-eq-1-iff*:
fixes $m\ n$ :: *int*
assumes $0 < m$
shows $m * n = 1 \longleftrightarrow m = 1 \wedge n = 1$
⟨*proof*⟩

lemma *zmult-eq-1-iff*: $m * n = 1 \longleftrightarrow (m = 1 \wedge n = 1) \vee (m = -\ 1 \wedge n = -\ 1)$
for $m\ n$ :: *int*
⟨*proof*⟩

lemma *infinite-UNIV-int*: $\neg$ *finite* (*UNIV*::*int set*)
⟨*proof*⟩

52.17 Further theorems on numerals

52.17.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

lemmas *distrib-right-numeral* [*simp*] = *distrib-right* [*of - - numeral v*] **for** *v*
lemmas *distrib-left-numeral* [*simp*] = *distrib-left* [*of numeral v*] **for** *v*
lemmas *left-diff-distrib-numeral* [*simp*] = *left-diff-distrib* [*of - - numeral v*] **for** *v*
lemmas *right-diff-distrib-numeral* [*simp*] = *right-diff-distrib* [*of numeral v*] **for** *v*

These are actually for fields, like real: but where else to put them?

lemmas *zero-less-divide-iff-numeral* [*simp*, *no-atp*] = *zero-less-divide-iff* [*of numeral w*] **for** *w*
lemmas *divide-less-0-iff-numeral* [*simp*, *no-atp*] = *divide-less-0-iff* [*of numeral w*] **for** *w*
lemmas *zero-le-divide-iff-numeral* [*simp*, *no-atp*] = *zero-le-divide-iff* [*of numeral w*] **for** *w*
lemmas *divide-le-0-iff-numeral* [*simp*, *no-atp*] = *divide-le-0-iff* [*of numeral w*] **for** *w*

Replaces *inverse #nn* by *1/#nn*. It looks strange, but then other simprocs simplify the quotient.

lemmas *inverse-eq-divide-numeral* [*simp*] =
inverse-eq-divide [*of numeral w*] **for** *w*

lemmas *inverse-eq-divide-neg-numeral* [*simp*] =
inverse-eq-divide [*of* − *numeral w*] **for** *w*

These laws simplify inequalities, moving unary minus from a term into the literal.

lemmas *equation-minus-iff-numeral* [*no-atp*] =
equation-minus-iff [*of numeral v*] **for** *v*

lemmas *minus-equation-iff-numeral* [*no-atp*] =
minus-equation-iff [*of - numeral v*] **for** *v*

lemmas *le-minus-iff-numeral* [*no-atp*] =
le-minus-iff [*of numeral v*] **for** *v*

lemmas *minus-le-iff-numeral* [*no-atp*] =
minus-le-iff [*of - numeral v*] **for** *v*

lemmas *less-minus-iff-numeral* [*no-atp*] =
less-minus-iff [*of numeral v*] **for** *v*

lemmas *minus-less-iff-numeral* [*no-atp*] =
minus-less-iff [*of - numeral v*] **for** *v*

Cancellation of constant factors in comparisons ($<$ and $\leq$)

lemmas *mult-less-cancel-left-numeral* [*simp*, *no-atp*] = *mult-less-cancel-left* [*of numeral v*] **for** *v*
lemmas *mult-less-cancel-right-numeral* [*simp*, *no-atp*] = *mult-less-cancel-right* [*of - numeral v*] **for** *v*
lemmas *mult-le-cancel-left-numeral* [*simp*, *no-atp*] = *mult-le-cancel-left* [*of numeral v*] **for** *v*
lemmas *mult-le-cancel-right-numeral* [*simp*, *no-atp*] = *mult-le-cancel-right* [*of - numeral v*] **for** *v*

Multiplying out constant divisors in comparisons ($<$, $\leq$ and $=$)

named-theorems *divide-const-simps simplification rules to simplify comparisons involving constant divisors*

lemmas *le-divide-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-le-divide-eq [*of numeral w*, *OF zero-less-numeral*]
neg-le-divide-eq [*of* − *numeral w*, *OF neg-numeral-less-zero*] **for** *w*

lemmas *divide-le-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-divide-le-eq [*of numeral w*, *OF zero-less-numeral*]
neg-divide-le-eq [*of* − *numeral w*, *OF neg-numeral-less-zero*] **for** *w*

lemmas *less-divide-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-less-divide-eq [*of numeral w*, *OF zero-less-numeral*]
neg-less-divide-eq [*of* − *numeral w*, *OF neg-numeral-less-zero*] **for** *w*

lemmas *divide-less-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-divide-less-eq [*of numeral w*, *OF zero-less-numeral*]
neg-divide-less-eq [*of* − *numeral w*, *OF neg-numeral-less-zero*] **for** *w*

lemmas *eq-divide-eq-numeral1* [*simp*,*divide-const-simps*] =
eq-divide-eq [*of - - numeral w*]
eq-divide-eq [*of - -* − *numeral w*] **for** *w*

lemmas *divide-eq-eq-numeral1* [*simp*,*divide-const-simps*] =
divide-eq-eq [*of - numeral w*]
divide-eq-eq [*of -* − *numeral w*] **for** *w*

52.17.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas *le-divide-eq-numeral* [*divide-const-simps*] =
le-divide-eq [*of numeral w*]
le-divide-eq [*of* − *numeral w*] **for** *w*

lemmas *divide-le-eq-numeral* [*divide-const-simps*] =
divide-le-eq [*of - - numeral w*]
divide-le-eq [*of - -* − *numeral w*] **for** *w*

lemmas *less-divide-eq-numeral* [*divide-const-simps*] =

less-divide-eq [*of numeral w*]
less-divide-eq [*of* − *numeral w*] **for** *w*

lemmas *divide-less-eq-numeral* [*divide-const-simps*] =
divide-less-eq [*of - - numeral w*]
divide-less-eq [*of - -* − *numeral w*] **for** *w*

lemmas *eq-divide-eq-numeral* [*divide-const-simps*] =
eq-divide-eq [*of numeral w*]
eq-divide-eq [*of* − *numeral w*] **for** *w*

lemmas *divide-eq-eq-numeral* [*divide-const-simps*] =
divide-eq-eq [*of - - numeral w*]
divide-eq-eq [*of - -* − *numeral w*] **for** *w*

Not good as automatic simprules because they cause case splits.

lemmas [*divide-const-simps*] =
le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1

52.18 The divides relation

lemma *zdvd-antisym-nonneg*: $0 \le m \Longrightarrow 0 \le n \Longrightarrow m\ dvd\ n \Longrightarrow n\ dvd\ m \Longrightarrow m = n$
for *m n* :: *int*
⟨*proof*⟩

lemma *zdvd-antisym-abs*:
fixes *a b* :: *int*
assumes *a dvd b* **and** *b dvd a*
shows $|a| = |b|$
⟨*proof*⟩

lemma *zdvd-zdiffD*: $k\ dvd\ m - n \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ m$
for *k m n* :: *int*
⟨*proof*⟩

lemma *zdvd-reduce*: $k\ dvd\ n + k * m \longleftrightarrow k\ dvd\ n$
for *k m n* :: *int*
⟨*proof*⟩

lemma *dvd-imp-le-int*:
fixes *d i* :: *int*
assumes $i \neq 0$ **and** *d dvd i*
shows $|d| \le |i|$
⟨*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*:
fixes *k m n* :: *int*
assumes *d*: $k * m$ *dvd* $k * n$
and $k \neq 0$
shows *m dvd n*
⟨*proof*⟩

theorem *zdvd-int*: *x dvd y* $\longleftrightarrow$ *int x dvd int y*
⟨*proof*⟩

lemma *zdvd1-eq*[*simp*]: *x dvd 1* $\longleftrightarrow$ $|x| = 1$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
for *x* :: *int*
⟨*proof*⟩

lemma *zdvd-mult-cancel1*:
fixes *m* :: *int*
assumes *mp*: $m \neq 0$
shows $m * n$ *dvd m* $\longleftrightarrow$ $|n| = 1$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *int-dvd-iff*: *int m dvd z* $\longleftrightarrow$ *m dvd nat* $|z|$
⟨*proof*⟩

lemma *dvd-int-iff*: *z dvd int m* $\longleftrightarrow$ *nat* $|z|$ *dvd m*
⟨*proof*⟩

lemma *dvd-int-unfold-dvd-nat*: *k dvd l* $\longleftrightarrow$ *nat* $|k|$ *dvd nat* $|l|$
⟨*proof*⟩

lemma *nat-dvd-iff*: *nat z dvd m* $\longleftrightarrow$ (*if* $0 \leq z$ *then z dvd int m else* $m = 0$)
⟨*proof*⟩

lemma *eq-nat-nat-iff*: $0 \leq z \Longrightarrow 0 \leq z' \Longrightarrow$ *nat* $z =$ *nat* $z' \longleftrightarrow z = z'$
⟨*proof*⟩

lemma *nat-power-eq*: $0 \leq z \Longrightarrow$ *nat* $(z \hat{} n) =$ *nat* $z \hat{} n$
⟨*proof*⟩

lemma *zdvd-imp-le*: *z dvd n* $\Longrightarrow 0 < n \Longrightarrow z \leq n$
for *n z* :: *int*
⟨*proof*⟩

lemma *zdvd-period*:
fixes *a d* :: *int*

assumes *a dvd d*
shows $a\ dvd\ (x + t) \longleftrightarrow a\ dvd\ ((x + c * d) + t)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

52.19 Finiteness of intervals

lemma *finite-interval-int1* [*iff*]: *finite* $\{i :: int.\ a \leq i \wedge i \leq b\}$
⟨*proof*⟩

lemma *finite-interval-int2* [*iff*]: *finite* $\{i :: int.\ a \leq i \wedge i < b\}$
⟨*proof*⟩

lemma *finite-interval-int3* [*iff*]: *finite* $\{i :: int.\ a < i \wedge i \leq b\}$
⟨*proof*⟩

lemma *finite-interval-int4* [*iff*]: *finite* $\{i :: int.\ a < i \wedge i < b\}$
⟨*proof*⟩

52.20 Configuration of the code generator

Constructors

definition *Pos* :: *num* ⇒ *int*
where [*simp*, *code-abbrev*]: *Pos* = *numeral*

definition *Neg* :: *num* ⇒ *int*
where [*simp*, *code-abbrev*]: *Neg n* = − (*Pos n*)

code-datatype *0::int Pos Neg*

Auxiliary operations.

definition *dup* :: *int* ⇒ *int*
where [*simp*]: *dup k* = *k* + *k*

lemma *dup-code* [*code*]:
dup 0 = *0*
dup (*Pos n*) = *Pos* (*Num.Bit0 n*)
dup (*Neg n*) = *Neg* (*Num.Bit0 n*)
⟨*proof*⟩

definition *sub* :: *num* ⇒ *num* ⇒ *int*
where [*simp*]: *sub m n* = *numeral m* − *numeral n*

lemma *sub-code* [*code*]:
sub Num.One Num.One = *0*
sub (*Num.Bit0 m*) *Num.One* = *Pos* (*Num.BitM m*)
sub (*Num.Bit1 m*) *Num.One* = *Pos* (*Num.Bit0 m*)
sub Num.One (*Num.Bit0 n*) = *Neg* (*Num.BitM n*)
sub Num.One (*Num.Bit1 n*) = *Neg* (*Num.Bit0 n*)

$sub\ (Num.Bit0\ m)\ (Num.Bit0\ n) = dup\ (sub\ m\ n)$
$sub\ (Num.Bit1\ m)\ (Num.Bit1\ n) = dup\ (sub\ m\ n)$
$sub\ (Num.Bit1\ m)\ (Num.Bit0\ n) = dup\ (sub\ m\ n) + 1$
$sub\ (Num.Bit0\ m)\ (Num.Bit1\ n) = dup\ (sub\ m\ n) - 1$
⟨*proof*⟩

Implementations.

lemma *one-int-code* [*code*]: $1 = Pos\ Num.One$
⟨*proof*⟩

lemma *plus-int-code* [*code*]:
$k + 0 = k$
$0 + l = l$
$Pos\ m + Pos\ n = Pos\ (m + n)$
$Pos\ m + Neg\ n = sub\ m\ n$
$Neg\ m + Pos\ n = sub\ n\ m$
$Neg\ m + Neg\ n = Neg\ (m + n)$
for $k\ l :: int$
⟨*proof*⟩

lemma *uminus-int-code* [*code*]:
$uminus\ 0 = (0::int)$
$uminus\ (Pos\ m) = Neg\ m$
$uminus\ (Neg\ m) = Pos\ m$
⟨*proof*⟩

lemma *minus-int-code* [*code*]:
$k - 0 = k$
$0 - l = uminus\ l$
$Pos\ m - Pos\ n = sub\ m\ n$
$Pos\ m - Neg\ n = Pos\ (m + n)$
$Neg\ m - Pos\ n = Neg\ (m + n)$
$Neg\ m - Neg\ n = sub\ n\ m$
for $k\ l :: int$
⟨*proof*⟩

lemma *times-int-code* [*code*]:
$k * 0 = 0$
$0 * l = 0$
$Pos\ m * Pos\ n = Pos\ (m * n)$
$Pos\ m * Neg\ n = Neg\ (m * n)$
$Neg\ m * Pos\ n = Neg\ (m * n)$
$Neg\ m * Neg\ n = Pos\ (m * n)$
for $k\ l :: int$
⟨*proof*⟩

instantiation *int* :: *equal*
begin

definition $HOL.equal\ k\ l \longleftrightarrow k = (l::int)$

instance
⟨*proof*⟩

end

lemma *equal-int-code* [*code*]:
$HOL.equal\ 0\ (0::int) \longleftrightarrow True$
$HOL.equal\ 0\ (Pos\ l) \longleftrightarrow False$
$HOL.equal\ 0\ (Neg\ l) \longleftrightarrow False$
$HOL.equal\ (Pos\ k)\ 0 \longleftrightarrow False$
$HOL.equal\ (Pos\ k)\ (Pos\ l) \longleftrightarrow HOL.equal\ k\ l$
$HOL.equal\ (Pos\ k)\ (Neg\ l) \longleftrightarrow False$
$HOL.equal\ (Neg\ k)\ 0 \longleftrightarrow False$
$HOL.equal\ (Neg\ k)\ (Pos\ l) \longleftrightarrow False$
$HOL.equal\ (Neg\ k)\ (Neg\ l) \longleftrightarrow HOL.equal\ k\ l$
⟨*proof*⟩

lemma *equal-int-refl* [*code nbe*]: $HOL.equal\ k\ k \longleftrightarrow True$
for $k :: int$
⟨*proof*⟩

lemma *less-eq-int-code* [*code*]:
$0 \leq (0::int) \longleftrightarrow True$
$0 \leq Pos\ l \longleftrightarrow True$
$0 \leq Neg\ l \longleftrightarrow False$
$Pos\ k \leq 0 \longleftrightarrow False$
$Pos\ k \leq Pos\ l \longleftrightarrow k \leq l$
$Pos\ k \leq Neg\ l \longleftrightarrow False$
$Neg\ k \leq 0 \longleftrightarrow True$
$Neg\ k \leq Pos\ l \longleftrightarrow True$
$Neg\ k \leq Neg\ l \longleftrightarrow l \leq k$
⟨*proof*⟩

lemma *less-int-code* [*code*]:
$0 < (0::int) \longleftrightarrow False$
$0 < Pos\ l \longleftrightarrow True$
$0 < Neg\ l \longleftrightarrow False$
$Pos\ k < 0 \longleftrightarrow False$
$Pos\ k < Pos\ l \longleftrightarrow k < l$
$Pos\ k < Neg\ l \longleftrightarrow False$
$Neg\ k < 0 \longleftrightarrow True$
$Neg\ k < Pos\ l \longleftrightarrow True$
$Neg\ k < Neg\ l \longleftrightarrow l < k$
⟨*proof*⟩

lemma *nat-code* [*code*]:
$nat\ (Int.Neg\ k) = 0$

nat 0 = 0
nat (Int.Pos k) = nat-of-num k
⟨*proof*⟩

lemma (**in** *ring-1*) *of-int-code* [*code*]:
of-int (Int.Neg k) = − numeral k
of-int 0 = 0
of-int (Int.Pos k) = numeral k
⟨*proof*⟩

Serializer setup.

code-identifier
code-module *Int* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

quickcheck-params [*default-type = int*]

hide-const (**open**) *Pos Neg sub dup*

De-register *int* as a quotient type:

lifting-update *int.lifting*
lifting-forget *int.lifting*

end

53 Generic transfer machinery; specific transfer from nats to ints and back.

theory *Nat-Transfer*
imports *Int*
begin

53.1 Generic transfer machinery

definition *transfer-morphism*:: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow bool) \Rightarrow bool$
where *transfer-morphism f A* ⟷ *True*

lemma *transfer-morphismI* [*intro*]: *transfer-morphism f A*
⟨*proof*⟩

⟨*ML*⟩

53.2 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*:
$nat\text{-}set\ \{\}$
$nat\text{-}set\ A \Longrightarrow nat\text{-}set\ B \Longrightarrow nat\text{-}set\ (A\ Un\ B)$
$nat\text{-}set\ A \Longrightarrow nat\text{-}set\ B \Longrightarrow nat\text{-}set\ (A\ Int\ B)$

nat-set $\{x.\ x >= 0\ \&\ P\ x\}$
nat-set $(int\ `\ C)$
nat-set $A \Longrightarrow x : A \Longrightarrow x >= 0$
⟨*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* $A \Longrightarrow$
$(int\ `\ nat\ `\ A = A)$
⟨*proof*⟩

lemma *transfer-nat-int-set-cong*: $(!!x.\ x >= 0 \Longrightarrow P\ x = P'\ x) \Longrightarrow$
$\{(x::int).\ x >= 0\ \&\ P\ x\} = \{x.\ x >= 0\ \&\ P'\ x\}$
⟨*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*
]

sum and prod

lemma *transfer-nat-int-sum-prod*:
$sum\ f\ A = sum\ (\%x.\ f\ (nat\ x))\ (int\ `\ A)$
$prod\ f\ A = prod\ (\%x.\ f\ (nat\ x))\ (int\ `\ A)$
⟨*proof*⟩

lemma *transfer-nat-int-sum-prod2*:
$sum\ f\ A = nat(sum\ (\%x.\ int\ (f\ x))\ A)$
$prod\ f\ A = nat(prod\ (\%x.\ int\ (f\ x))\ A)$
⟨*proof*⟩

lemma *transfer-nat-int-sum-prod-closure*:
nat-set $A \Longrightarrow (!!x.\ x >= 0 \Longrightarrow f\ x >= (0::int)) \Longrightarrow sum\ f\ A >= 0$
nat-set $A \Longrightarrow (!!x.\ x >= 0 \Longrightarrow f\ x >= (0::int)) \Longrightarrow prod\ f\ A >= 0$
⟨*proof*⟩

lemma *transfer-nat-int-sum-prod-cong*:
$A = B \Longrightarrow$ *nat-set* $B \Longrightarrow (!!x.\ x >= 0 \Longrightarrow f\ x = g\ x) \Longrightarrow$
sum $f\ A =$ *sum* $g\ B$
$A = B \Longrightarrow$ *nat-set* $B \Longrightarrow (!!x.\ x >= 0 \Longrightarrow f\ x = g\ x) \Longrightarrow$
prod $f\ A =$ *prod* $g\ B$
⟨*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*]

53.3 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*:
$(int\ x) + (int\ y) = int\ (x + y)$
$(int\ x) * (int\ y) = int\ (x * y)$
$tsub\ (int\ x)\ (int\ y) = int\ (x - y)$
$(int\ x)\hat{}n = int\ (x\hat{}n)$
⟨*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* $(x\hat{}n)$

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 ⟹ *card A* = *card* (*nat* ' *A*)
{} = *int* ' ({}::*nat set*)
nat-set A ⟹ *nat-set B* ⟹ *A Un B* = *int* ' (*nat* ' *A Un nat* ' *B*)
nat-set A ⟹ *nat-set B* ⟹ *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 $\Longrightarrow$ *nat-set B* $\Longrightarrow$ *nat-set* (*A Un B*)
nat-set A $\Longrightarrow$ *nat-set B* $\Longrightarrow$ *nat-set* (*A Int B*)
nat-set {*x*. *x* >= *0* & *P x*}
nat-set (*int* ‘ *C*)
nat-set A $\Longrightarrow$ *x* : *A* $\Longrightarrow$ *is-nat x*
⟨*proof*⟩

lemma *transfer-int-nat-set-relations*:
nat-set A $\Longrightarrow$ *finite A* = *finite* (*nat* ‘ *A*)
is-nat x $\Longrightarrow$ *nat-set A* $\Longrightarrow$ (*x* : *A*) = (*nat x* : *nat* ‘ *A*)
nat-set A $\Longrightarrow$ *nat-set B* $\Longrightarrow$ (*A* = *B*) = (*nat* ‘ *A* = *nat* ‘ *B*)
nat-set A $\Longrightarrow$ *nat-set B* $\Longrightarrow$ (*A* < *B*) = (*nat* ‘ *A* < *nat* ‘ *B*)
nat-set A $\Longrightarrow$ *nat-set B* $\Longrightarrow$ (*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*) $\Longrightarrow$
{(*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*
]

sum and prod

lemma *transfer-int-nat-sum-prod*:
nat-set A $\Longrightarrow$ *sum f A* = *sum* (%*x*. *f* (*int x*)) (*nat* ‘ *A*)
nat-set A $\Longrightarrow$ *prod f A* = *prod* (%*x*. *f* (*int x*)) (*nat* ‘ *A*)
⟨*proof*⟩

lemma *transfer-int-nat-sum-prod2*:
(!!*x*. *x*:*A* $\Longrightarrow$ *is-nat* (*f x*)) $\Longrightarrow$ *sum f A* = *int*(*sum* (%*x*. *nat* (*f x*)) *A*)
(!!*x*. *x*:*A* $\Longrightarrow$ *is-nat* (*f x*)) $\Longrightarrow$
prod f A = *int*(*prod* (%*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: *sum.cong prod.cong*]

end

54 Uniquely determined division in euclidean (semi)rings

theory *Euclidean-Division*
  imports *Nat-Transfer*
begin

54.1 Quotient and remainder in integral domains

class *semidom-modulo* = *algebraic-semidom* + *semiring-modulo*
begin

lemma *mod-0* [*simp*]: *0 mod a* = *0*
  ⟨*proof*⟩

lemma *mod-by-0* [*simp*]: *a mod 0* = *a*
  ⟨*proof*⟩

lemma *mod-by-1* [*simp*]:
  a mod 1 = *0*
⟨*proof*⟩

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

lemma *dvd-imp-mod-0* [*simp*]:
  assumes *a dvd b*
  shows *b mod a* = *0*
  ⟨*proof*⟩

lemma *mod-0-imp-dvd*:
  assumes *a mod b* = *0*
  shows *b dvd a*
⟨*proof*⟩

lemma *mod-eq-0-iff-dvd*:
  a mod b = *0* ⟷ *b dvd a*
  ⟨*proof*⟩

lemma *dvd-eq-mod-eq-0* [*nitpick-unfold*, *code*]:
  a dvd b ⟷ *b mod a* = *0*
  ⟨*proof*⟩

lemma *dvd-mod-iff*:
  assumes *c dvd b*
  shows *c dvd a mod b* ⟷ *c dvd a*
⟨*proof*⟩

lemma *dvd-mod-imp-dvd*:
  assumes *c dvd a mod b* **and** *c dvd b*

shows *c dvd a*
⟨*proof*⟩

end

class *idom-modulo* = *idom* + *semidom-modulo*
begin

subclass *idom-divide* ⟨*proof*⟩

lemma *div-diff* [*simp*]:
$c\ dvd\ a \Longrightarrow c\ dvd\ b \Longrightarrow (a - b)\ div\ c = a\ div\ c - b\ div\ c$
⟨*proof*⟩

end

54.2 Euclidean (semi)rings with explicit division and remainder

class *euclidean-semiring* = *semidom-modulo* + *normalization-semidom* +
fixes *euclidean-size* :: $'a \Rightarrow nat$
assumes *size-0* [*simp*]: *euclidean-size* $0 = 0$
assumes *mod-size-less*:
$b \neq 0 \Longrightarrow$ *euclidean-size* $(a\ mod\ b) <$ *euclidean-size* b
assumes *size-mult-mono*:
$b \neq 0 \Longrightarrow$ *euclidean-size* $a \leq$ *euclidean-size* $(a * b)$
begin

lemma *size-mult-mono'*: $b \neq 0 \Longrightarrow$ *euclidean-size* $a \leq$ *euclidean-size* $(b * a)$
⟨*proof*⟩

lemma *euclidean-size-normalize* [*simp*]:
euclidean-size (*normalize* a) = *euclidean-size* a
⟨*proof*⟩

lemma *dvd-euclidean-size-eq-imp-dvd*:
assumes $a \neq 0$ **and** *euclidean-size* a = *euclidean-size* b
and *b dvd a*
shows *a dvd b*
⟨*proof*⟩

lemma *euclidean-size-times-unit*:
assumes *is-unit a*
shows *euclidean-size* $(a * b)$ = *euclidean-size* b
⟨*proof*⟩

lemma *euclidean-size-unit*:
is-unit $a \Longrightarrow$ *euclidean-size* a = *euclidean-size* 1
⟨*proof*⟩

lemma *unit-iff-euclidean-size*:
is-unit a $\longleftrightarrow$ *euclidean-size a* = *euclidean-size 1* $\wedge$ *a* $\neq$ *0*
⟨*proof*⟩

lemma *euclidean-size-times-nonunit*:
assumes *a* $\neq$ *0 b* $\neq$ *0* $\neg$ *is-unit a*
shows *euclidean-size b* < *euclidean-size* (*a* ∗ *b*)
⟨*proof*⟩

lemma *dvd-imp-size-le*:
assumes *a dvd b b* $\neq$ *0*
shows *euclidean-size a* $\leq$ *euclidean-size b*
⟨*proof*⟩

lemma *dvd-proper-imp-size-less*:
assumes *a dvd b* $\neg$ *b dvd a b* $\neq$ *0*
shows *euclidean-size a* < *euclidean-size b*
⟨*proof*⟩

end

class *euclidean-ring* = *idom-modulo* + *euclidean-semiring*

54.3 Uniquely determined division

class *unique-euclidean-semiring* = *euclidean-semiring* +
fixes *uniqueness-constraint* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *size-mono-mult*:
b $\neq$ *0* $\Longrightarrow$ *euclidean-size a* < *euclidean-size c*
$\Longrightarrow$ *euclidean-size* (*a* ∗ *b*) < *euclidean-size* (*c* ∗ *b*)
— FIXME justify
assumes *uniqueness-constraint-mono-mult*:
uniqueness-constraint a b $\Longrightarrow$ *uniqueness-constraint* (*a* ∗ *c*) (*b* ∗ *c*)
assumes *uniqueness-constraint-mod*:
b $\neq$ *0* $\Longrightarrow$ $\neg$ *b dvd a* $\Longrightarrow$ *uniqueness-constraint* (*a mod b*) *b*
assumes *div-bounded*:
b $\neq$ *0* $\Longrightarrow$ *uniqueness-constraint r b*
$\Longrightarrow$ *euclidean-size r* < *euclidean-size b*
$\Longrightarrow$ (*q* ∗ *b* + *r*) *div b* = *q*
begin

lemma *divmod-cases* [*case-names divides remainder by0*]:
obtains
(*divides*) *q* **where** *b* $\neq$ *0*
and *a div b* = *q*
and *a mod b* = *0*
and *a* = *q* ∗ *b*
| (*remainder*) *q r* **where** *b* $\neq$ *0* **and** *r* $\neq$ *0*

and *uniqueness-constraint r b*
and *euclidean-size r* $<$ *euclidean-size b*
and a *div* $b = q$
and a *mod* $b = r$
and $a = q * b + r$
| (*by0*) $b = 0$
⟨*proof*⟩

lemma *div-eqI*:
a *div* $b = q$ **if** $b \neq 0$ *uniqueness-constraint r b*
euclidean-size r $<$ *euclidean-size b* $q * b + r = a$
⟨*proof*⟩

lemma *mod-eqI*:
a *mod* $b = r$ **if** $b \neq 0$ *uniqueness-constraint r b*
euclidean-size r $<$ *euclidean-size b* $q * b + r = a$
⟨*proof*⟩

end

class *unique-euclidean-ring* = *euclidean-ring* + *unique-euclidean-semiring*

end

55 Parity in rings and semirings

theory *Parity*
imports *Nat-Transfer Euclidean-Division*
begin

55.1 Ring structures with parity and *even*/*odd* predicates

class *semiring-parity* = *comm-semiring-1-cancel* + *numeral* +
assumes *odd-one* [*simp*]: $\neg$ *2 dvd 1*
assumes *odd-even-add*: $\neg$ *2 dvd a* $\Longrightarrow$ $\neg$ *2 dvd b* $\Longrightarrow$ *2 dvd a* + *b*
assumes *even-multD*: *2 dvd a* $*$ *b* $\Longrightarrow$ *2 dvd a* $\vee$ *2 dvd b*
assumes *odd-ex-decrement*: $\neg$ *2 dvd a* $\Longrightarrow$ $\exists b.\ a = b + 1$
begin

subclass *semiring-numeral* ⟨*proof*⟩

abbreviation *even* :: $'a \Rightarrow bool$
where *even a* $\equiv$ *2 dvd a*

abbreviation *odd* :: $'a \Rightarrow bool$
where *odd a* $\equiv$ $\neg$ *2 dvd a*

lemma *even-zero* [*simp*]: *even 0*
⟨*proof*⟩

lemma *even-plus-one-iff* [*simp*]: $even\ (a + 1) \longleftrightarrow odd\ a$
⟨*proof*⟩

lemma *evenE* [*elim?*]:
assumes $even\ a$
obtains b **where** $a = 2 * b$
⟨*proof*⟩

lemma *oddE* [*elim?*]:
assumes $odd\ a$
obtains b **where** $a = 2 * b + 1$
⟨*proof*⟩

lemma *even-times-iff* [*simp*]: $even\ (a * b) \longleftrightarrow even\ a \lor even\ b$
⟨*proof*⟩

lemma *even-numeral* [*simp*]: $even\ (numeral\ (Num.Bit0\ n))$
⟨*proof*⟩

lemma *odd-numeral* [*simp*]: $odd\ (numeral\ (Num.Bit1\ n))$
⟨*proof*⟩

lemma *even-add* [*simp*]: $even\ (a + b) \longleftrightarrow (even\ a \longleftrightarrow even\ b)$
⟨*proof*⟩

lemma *odd-add* [*simp*]: $odd\ (a + b) \longleftrightarrow (\neg\ (odd\ a \longleftrightarrow odd\ b))$
⟨*proof*⟩

lemma *even-power* [*simp*]: $even\ (a\ \hat{}\ n) \longleftrightarrow even\ a \land n > 0$
⟨*proof*⟩

end

class *ring-parity* = *ring* + *semiring-parity*
begin

subclass *comm-ring-1* ⟨*proof*⟩

lemma *even-minus* [*simp*]: $even\ (-\ a) \longleftrightarrow even\ a$
⟨*proof*⟩

lemma *even-diff* [*simp*]: $even\ (a - b) \longleftrightarrow even\ (a + b)$
⟨*proof*⟩

end

55.2 Instances for *nat* and *int*

lemma *even-Suc-Suc-iff* [*simp*]: $2\ dvd\ Suc\ (Suc\ n) \longleftrightarrow 2\ dvd\ n$
⟨*proof*⟩

lemma *even-Suc* [*simp*]: $2\ dvd\ Suc\ n \longleftrightarrow \neg\ 2\ dvd\ n$
⟨*proof*⟩

lemma *even-diff-nat* [*simp*]: $2\ dvd\ (m - n) \longleftrightarrow m < n \vee 2\ dvd\ (m + n)$
for $m\ n :: nat$
⟨*proof*⟩

instance *nat* :: *semiring-parity*
⟨*proof*⟩

lemma *odd-pos*: $odd\ n \Longrightarrow 0 < n$
for $n :: nat$
⟨*proof*⟩

lemma *Suc-double-not-eq-double*: $Suc\ (2 * m) \neq 2 * n$
for $m\ n :: nat$
⟨*proof*⟩

lemma *double-not-eq-Suc-double*: $2 * m \neq Suc\ (2 * n)$
for $m\ n :: nat$
⟨*proof*⟩

lemma *even-diff-iff* [*simp*]: $2\ dvd\ (k - l) \longleftrightarrow 2\ dvd\ (k + l)$
for $k\ l :: int$
⟨*proof*⟩

lemma *even-abs-add-iff* [*simp*]: $2\ dvd\ (|k| + l) \longleftrightarrow 2\ dvd\ (k + l)$
for $k\ l :: int$
⟨*proof*⟩

lemma *even-add-abs-iff* [*simp*]: $2\ dvd\ (k + |l|) \longleftrightarrow 2\ dvd\ (k + l)$
for $k\ l :: int$
⟨*proof*⟩

lemma *odd-Suc-minus-one* [*simp*]: $odd\ n \Longrightarrow Suc\ (n - Suc\ 0) = n$
⟨*proof*⟩

instance *int* :: *ring-parity*
⟨*proof*⟩

lemma *even-int-iff* [*simp*]: $even\ (int\ n) \longleftrightarrow even\ n$
⟨*proof*⟩

lemma *even-nat-iff*: $0 \leq k \Longrightarrow even\ (nat\ k) \longleftrightarrow even\ k$
⟨*proof*⟩

55.3 Parity and powers

context *ring-1*
begin

lemma *power-minus-even* [*simp*]: $even\ n \Longrightarrow (-\ a)\ \hat{}\ n = a\ \hat{}\ n$
⟨*proof*⟩

lemma *power-minus-odd* [*simp*]: $odd\ n \Longrightarrow (-\ a)\ \hat{}\ n = -\ (a\ \hat{}\ n)$
⟨*proof*⟩

lemma *neg-one-even-power* [*simp*]: $even\ n \Longrightarrow (-\ 1)\ \hat{}\ n = 1$
⟨*proof*⟩

lemma *neg-one-odd-power* [*simp*]: $odd\ n \Longrightarrow (-\ 1)\ \hat{}\ n = -\ 1$
⟨*proof*⟩

lemma *neg-one-power-add-eq-neg-one-power-diff*: $k \leq n \Longrightarrow (-\ 1)\ \hat{}\ (n + k) = (-\ 1)\ \hat{}\ (n - k)$
⟨*proof*⟩

end

context *linordered-idom*
begin

lemma *zero-le-even-power*: $even\ n \Longrightarrow 0 \leq a\ \hat{}\ n$
⟨*proof*⟩

lemma *zero-le-odd-power*: $odd\ n \Longrightarrow 0 \leq a\ \hat{}\ n \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-le-power-eq*: $0 \leq a\ \hat{}\ n \longleftrightarrow even\ n \vee odd\ n \wedge 0 \leq a$
⟨*proof*⟩

lemma *zero-less-power-eq*: $0 < a\ \hat{}\ n \longleftrightarrow n = 0 \vee even\ n \wedge a \neq 0 \vee odd\ n \wedge 0 < a$
⟨*proof*⟩

lemma *power-less-zero-eq* [*simp*]: $a\ \hat{}\ n < 0 \longleftrightarrow odd\ n \wedge a < 0$
⟨*proof*⟩

lemma *power-le-zero-eq*: $a\ \hat{}\ n \leq 0 \longleftrightarrow n > 0 \wedge (odd\ n \wedge a \leq 0 \vee even\ n \wedge a = 0)$
⟨*proof*⟩

lemma *power-even-abs*: $even\ n \Longrightarrow |a|\ \hat{}\ n = a\ \hat{}\ n$
⟨*proof*⟩

lemma *power-mono-even*:

assumes *even n* **and** $|a| \leq |b|$
shows $a \hat{} n \leq b \hat{} n$
⟨*proof*⟩

lemma *power-mono-odd*:
assumes *odd n* **and** $a \leq b$
shows $a \hat{} n \leq b \hat{} n$
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *uminus-power-if*: $(- x) \hat{} n = ($*if even n then* $x\hat{}n$ *else* $- (x \hat{} n))$
⟨*proof*⟩

Simplify, when the exponent is a numeral

lemma *zero-le-power-eq-numeral* [*simp*]:
$0 \leq a \hat{}$ *numeral w* $\longleftrightarrow$ *even* (*numeral w* :: *nat*) $\vee$ *odd* (*numeral w* :: *nat*) $\wedge$ $0 \leq a$
⟨*proof*⟩

lemma *zero-less-power-eq-numeral* [*simp*]:
$0 < a \hat{}$ *numeral w* $\longleftrightarrow$
numeral w = (*0* :: *nat*) $\vee$
even (*numeral w* :: *nat*) $\wedge$ $a \neq 0$ $\vee$
odd (*numeral w* :: *nat*) $\wedge$ $0 < a$
⟨*proof*⟩

lemma *power-le-zero-eq-numeral* [*simp*]:
$a \hat{}$ *numeral w* $\leq 0 \longleftrightarrow$
(*0* :: *nat*) < *numeral w* $\wedge$
(*odd* (*numeral w* :: *nat*) $\wedge$ $a \leq 0$ $\vee$ *even* (*numeral w* :: *nat*) $\wedge$ $a = 0$)
⟨*proof*⟩

lemma *power-less-zero-eq-numeral* [*simp*]:
$a \hat{}$ *numeral w* $< 0 \longleftrightarrow$ *odd* (*numeral w* :: *nat*) $\wedge$ $a < 0$
⟨*proof*⟩

lemma *power-even-abs-numeral* [*simp*]:
even (*numeral w* :: *nat*) $\Longrightarrow$ $|a| \hat{}$ *numeral w* = $a \hat{}$ *numeral w*
⟨*proof*⟩

end

55.3.1 Tool setup

declare *transfer-morphism-int-nat* [*transfer add return*: *even-int-iff*]

end

56 More on quotient and remainder

theory *Divides*
imports *Parity*
begin

56.1 Quotient and remainder in integral domains with additional properties

class *semiring-div* = *semidom-modulo* +
  assumes *div-mult-self1* [*simp*]: $b \neq 0 \Longrightarrow (a + c * b)$ *div* $b = c + a$ *div* b
    and *div-mult-mult1* [*simp*]: $c \neq 0 \Longrightarrow (c * a)$ *div* $(c * b) = a$ *div* b
begin

lemma *div-mult-self2* [*simp*]:
  assumes $b \neq 0$
  shows $(a + b * c)$ *div* $b = c + a$ *div* b
  ⟨*proof*⟩

lemma *div-mult-self3* [*simp*]:
  assumes $b \neq 0$
  shows $(c * b + a)$ *div* $b = c + a$ *div* b
  ⟨*proof*⟩

lemma *div-mult-self4* [*simp*]:
  assumes $b \neq 0$
  shows $(b * c + a)$ *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 *mod-mult-self3* [*simp*]:
  $(c * b + a)$ *mod* $b = a$ *mod* b
  ⟨*proof*⟩

lemma *mod-mult-self4* [*simp*]:
  $(b * c + a)$ *mod* $b = a$ *mod* b
  ⟨*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-add-self1*:
assumes $b \neq 0$
shows $(b + a) \ div \ b = a \ div \ b + 1$
⟨proof⟩

lemma *div-add-self2*:
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-trivial* [*simp*]:
$a \ mod \ b \ div \ b = 0$
⟨proof⟩

lemma *mod-mod-trivial* [*simp*]:
$a \ mod \ b \ mod \ b = a \ mod \ b$
⟨proof⟩

lemma *mod-mod-cancel*:
assumes $c \ dvd \ b$
shows $a \ mod \ b \ mod \ c = a \ mod \ c$
⟨proof⟩

lemma *div-mult-mult2* [*simp*]:
$c \neq 0 \Longrightarrow (a * c) \ div \ (b * c) = a \ div \ b$
⟨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 *mult-mod-left*: $(a\ mod\ b) * c = (a * c)\ mod\ (b * c)$
⟨*proof*⟩

lemma *mult-mod-right*: $c * (a\ mod\ b) = (c * a)\ mod\ (c * b)$
⟨*proof*⟩

lemma *dvd-mod*: $k\ dvd\ m \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ (m\ mod\ n)$
⟨*proof*⟩

lemma *div-plus-div-distrib-dvd-left*:
$c\ dvd\ a \Longrightarrow (a + b)\ div\ c = a\ div\ c + b\ div\ c$
⟨*proof*⟩

lemma *div-plus-div-distrib-dvd-right*:
$c\ dvd\ b \Longrightarrow (a + b)\ div\ c = a\ div\ c + b\ div\ c$
⟨*proof*⟩

named-theorems *mod-simps*

Addition respects modular equivalence.

lemma *mod-add-left-eq* [*mod-simps*]:
$(a\ mod\ c + b)\ mod\ c = (a + b)\ mod\ c$
⟨*proof*⟩

lemma *mod-add-right-eq* [*mod-simps*]:
$(a + b\ mod\ c)\ mod\ c = (a + b)\ mod\ c$
⟨*proof*⟩

lemma *mod-add-eq*:
$(a\ mod\ c + b\ mod\ c)\ mod\ c = (a + b)\ mod\ c$
⟨*proof*⟩

lemma *mod-sum-eq* [*mod-simps*]:
$(\sum i \in A.\ f\ i\ mod\ a)\ mod\ a = sum\ f\ A\ mod\ a$
⟨*proof*⟩

lemma *mod-add-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*⟩

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq* [*mod-simps*]:
$((a\ mod\ c) * b)\ mod\ c = (a * b)\ mod\ c$
⟨*proof*⟩

lemma *mod-mult-right-eq* [*mod-simps*]:

$(a * (b \ mod \ c)) \ mod \ c = (a * b) \ mod \ c$
⟨*proof*⟩

lemma *mod-mult-eq*:
$((a \ mod \ c) * (b \ mod \ c)) \ mod \ c = (a * b) \ mod \ c$
⟨*proof*⟩

lemma *mod-prod-eq* [*mod-simps*]:
$(\prod i{\in}A. \ f \ i \ mod \ a) \ mod \ a = prod \ f \ A \ mod \ a$
⟨*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*⟩

Exponentiation respects modular equivalence.

lemma *power-mod* [*mod-simps*]:
$((a \ mod \ b) \ \hat{} \ n) \ mod \ b = (a \ \hat{} \ n) \ mod \ b$
⟨*proof*⟩

end

class *ring-div* = *comm-ring-1* + *semiring-div*
begin

subclass *idom-divide* ⟨*proof*⟩

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

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

lemma *div-minus-right*: $a \ div \ (- \ b) = (- \ a) \ div \ b$
⟨*proof*⟩

lemma *mod-minus-right*: $a \ mod \ (- \ b) = - \ ((- \ a) \ mod \ b)$
⟨*proof*⟩

lemma *div-minus1-right* [*simp*]: $a \ div \ (- \ 1) = - \ a$
⟨*proof*⟩

lemma *mod-minus1-right* [*simp*]: $a \ mod \ (- \ 1) = 0$
⟨*proof*⟩

Negation respects modular equivalence.

lemma *mod-minus-eq* [*mod-simps*]:

$(- (a \ mod \ b)) \ mod \ b = (- \ a) \ mod \ b$
⟨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* [*mod-simps*]:
$(a \ mod \ c - b) \ mod \ c = (a - b) \ mod \ c$
⟨proof⟩

lemma *mod-diff-right-eq* [*mod-simps*]:
$(a - b \ mod \ c) \ mod \ c = (a - b) \ mod \ c$
⟨proof⟩

lemma *mod-diff-eq*:
$(a \ mod \ c - b \ mod \ c) \ mod \ c = (a - b) \ mod \ c$
⟨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 *minus-mod-self2* [*simp*]:
$(a - b) \ mod \ b = a \ mod \ b$
⟨proof⟩

lemma *minus-mod-self1* [*simp*]:
$(b - a) \ mod \ b = - \ a \ mod \ b$
⟨proof⟩

end

56.2 Euclidean (semi)rings with cancel rules

class *euclidean-semiring-cancel* = *euclidean-semiring* + *semiring-div*

class *euclidean-ring-cancel* = *euclidean-ring* + *ring-div*

context *unique-euclidean-semiring*
begin

subclass *euclidean-semiring-cancel*
⟨proof⟩

end

context *unique-euclidean-ring*
begin

subclass *euclidean-ring-cancel* ⟨*proof*⟩

end

56.3 Parity

class *semiring-div-parity* = *semiring-div* + *comm-semiring-1-cancel* + *numeral* +
  assumes *parity*: $a\ mod\ 2 = 0 \lor a\ mod\ 2 = 1$
  assumes *one-mod-two-eq-one* [*simp*]: $1\ mod\ 2 = 1$
  assumes *zero-not-eq-two*: $0 \neq 2$
begin

lemma *parity-cases* [*case-names even odd*]:
  assumes $a\ mod\ 2 = 0 \Longrightarrow P$
  assumes $a\ mod\ 2 = 1 \Longrightarrow P$
  shows P
  ⟨*proof*⟩

lemma *one-div-two-eq-zero* [*simp*]:
  $1\ div\ 2 = 0$
⟨*proof*⟩

lemma *not-mod-2-eq-0-eq-1* [*simp*]:
  $a\ mod\ 2 \neq 0 \longleftrightarrow a\ mod\ 2 = 1$
  ⟨*proof*⟩

lemma *not-mod-2-eq-1-eq-0* [*simp*]:
  $a\ mod\ 2 \neq 1 \longleftrightarrow a\ mod\ 2 = 0$
  ⟨*proof*⟩

subclass *semiring-parity*
⟨*proof*⟩

lemma *even-iff-mod-2-eq-zero*:
  $even\ a \longleftrightarrow a\ mod\ 2 = 0$
  ⟨*proof*⟩

lemma *odd-iff-mod-2-eq-one*:
  $odd\ a \longleftrightarrow a\ mod\ 2 = 1$
  ⟨*proof*⟩

lemma *even-succ-div-two* [*simp*]:
  $even\ a \Longrightarrow (a + 1)\ div\ 2 = a\ div\ 2$
  ⟨*proof*⟩

lemma *odd-succ-div-two* [*simp*]:
  $odd\ a \Longrightarrow (a + 1)\ div\ 2 = a\ div\ 2 + 1$
  ⟨*proof*⟩

lemma *even-two-times-div-two*:
  $even\ a \Longrightarrow 2 * (a\ div\ 2) = a$
  ⟨*proof*⟩

lemma *odd-two-times-div-two-succ* [*simp*]:
  $odd\ a \Longrightarrow 2 * (a\ div\ 2) + 1 = a$
  ⟨*proof*⟩

end

56.4 Numeral division with a pragmatic type class

The following type class contains everything necessary to formulate a division algorithm in ring structures with numerals, restricted to its positive segments. This is its primary motiviation, and it could surely be formulated using a more fine-grained, more algebraic and less technical class hierarchy.

class *semiring-numeral-div* = *semiring-div* + *comm-semiring-1-cancel* + *linordered-semidom* +
  assumes *div-less*: $0 \leq a \Longrightarrow a < b \Longrightarrow a\ div\ b = 0$
    and *mod-less*: $0 \leq a \Longrightarrow a < b \Longrightarrow a\ mod\ b = a$
    and *div-positive*: $0 < b \Longrightarrow b \leq a \Longrightarrow a\ div\ b > 0$
    and *mod-less-eq-dividend*: $0 \leq a \Longrightarrow a\ mod\ b \leq a$
    and *pos-mod-bound*: $0 < b \Longrightarrow a\ mod\ b < b$
    and *pos-mod-sign*: $0 < b \Longrightarrow 0 \leq a\ mod\ b$
    and *mod-mult2-eq*: $0 \leq c \Longrightarrow a\ mod\ (b * c) = b * (a\ div\ b\ mod\ c) + a\ mod\ b$
    and *div-mult2-eq*: $0 \leq c \Longrightarrow a\ div\ (b * c) = a\ div\ b\ div\ c$
  assumes *discrete*: $a < b \longleftrightarrow a + 1 \leq b$
  fixes *divmod* :: $num \Rightarrow num \Rightarrow 'a \times 'a$
    and *divmod-step* :: $num \Rightarrow 'a \times 'a \Rightarrow 'a \times 'a$
  assumes *divmod-def*: $divmod\ m\ n = (numeral\ m\ div\ numeral\ n,\ numeral\ m\ mod\ numeral\ n)$
    and *divmod-step-def*: $divmod\text{-}step\ l\ qr = (let\ (q,\ r) = qr$
    $in\ if\ r \geq numeral\ l\ then\ (2 * q + 1,\ r - numeral\ l)$
    $else\ (2 * q,\ r))$
  — These are conceptually definitions but force generated code to be monomorphic wrt. particular instances of this class which yields a significant speedup.

begin

subclass *semiring-div-parity*
⟨*proof*⟩

lemma *divmod-digit-1*:

assumes $0 \leq a$ $0 < b$ **and** $b \leq a$ *mod* $(2 * b)$
shows $2 * (a$ *div* $(2 * b)) + 1 = a$ *div* b (**is** $?P$)
and a *mod* $(2 * b) - b = a$ *mod* b (**is** $?Q$)
⟨*proof*⟩

lemma *divmod-digit-0*:
assumes $0 < b$ **and** a *mod* $(2 * b) < b$
shows $2 * (a$ *div* $(2 * b)) = a$ *div* b (**is** $?P$)
and a *mod* $(2 * b) = a$ *mod* b (**is** $?Q$)
⟨*proof*⟩

lemma *fst-divmod*:
fst (*divmod m n*) = *numeral m div numeral n*
⟨*proof*⟩

lemma *snd-divmod*:
snd (*divmod m n*) = *numeral m mod numeral n*
⟨*proof*⟩

This is a formulation of one step (referring to one digit position) in school-method division: compare the dividend at the current digit position with the remainder from previous division steps and evaluate accordingly.

lemma *divmod-step-eq* [*simp*]:
divmod-step l $(q, r) =$ (*if numeral* $l \leq r$
then $(2 * q + 1, r -$ *numeral* $l)$ *else* $(2 * q, r))$
⟨*proof*⟩

This is a formulation of school-method division. If the divisor is smaller than the dividend, terminate. If not, shift the dividend to the right until termination occurs and then reiterate single division steps in the opposite direction.

lemma *divmod-divmod-step*:
divmod m n = (*if* $m < n$ *then* (*0*, *numeral m*)
else divmod-step n (*divmod m* (*Num.Bit0 n*)))
⟨*proof*⟩

The division rewrite proper – first, trivial results involving *1*

lemma *divmod-trivial* [*simp*]:
divmod Num.One Num.One = (*numeral Num.One*, *0*)
divmod (*Num.Bit0 m*) *Num.One* = (*numeral* (*Num.Bit0 m*), *0*)
divmod (*Num.Bit1 m*) *Num.One* = (*numeral* (*Num.Bit1 m*), *0*)
divmod num.One (*num.Bit0 n*) = (*0*, *Numeral1*)
divmod num.One (*num.Bit1 n*) = (*0*, *Numeral1*)
⟨*proof*⟩

Division by an even number is a right-shift

lemma *divmod-cancel* [*simp*]:

divmod (*Num.Bit0 m*) (*Num.Bit0 n*) = (*case divmod m n of* (*q*, *r*) ⇒ (*q*, *2* ∗ *r*)) (**is** *?P*)
divmod (*Num.Bit1 m*) (*Num.Bit0 n*) = (*case divmod m n of* (*q*, *r*) ⇒ (*q*, *2* ∗ *r* + *1*)) (**is** *?Q*)
⟨*proof*⟩

The really hard work

lemma *divmod-steps* [*simp*]:
divmod (*num.Bit0 m*) (*num.Bit1 n*) =
(*if* $m \leq n$ *then* (*0*, *numeral* (*num.Bit0 m*))
else divmod-step (*num.Bit1 n*)
(*divmod* (*num.Bit0 m*)
(*num.Bit0* (*num.Bit1 n*))))
divmod (*num.Bit1 m*) (*num.Bit1 n*) =
(*if* $m < n$ *then* (*0*, *numeral* (*num.Bit1 m*))
else divmod-step (*num.Bit1 n*)
(*divmod* (*num.Bit1 m*)
(*num.Bit0* (*num.Bit1 n*))))
⟨*proof*⟩

lemmas *divmod-algorithm-code* = *divmod-step-eq divmod-trivial divmod-cancel divmod-steps*

Special case: divisibility

definition *divides-aux* :: *'a* × *'a* ⇒ *bool*
where
divides-aux qr ⟷ *snd qr* = *0*

lemma *divides-aux-eq* [*simp*]:
divides-aux (*q*, *r*) ⟷ *r* = *0*
⟨*proof*⟩

lemma *dvd-numeral-simp* [*simp*]:
numeral m dvd numeral n ⟷ *divides-aux* (*divmod n m*)
⟨*proof*⟩

Generic computation of quotient and remainder

lemma *numeral-div-numeral* [*simp*]:
numeral k div numeral l = *fst* (*divmod k l*)
⟨*proof*⟩

lemma *numeral-mod-numeral* [*simp*]:
numeral k mod numeral l = *snd* (*divmod k l*)
⟨*proof*⟩

lemma *one-div-numeral* [*simp*]:
1 div numeral n = *fst* (*divmod num.One n*)
⟨*proof*⟩

lemma *one-mod-numeral* [*simp*]:

1 mod numeral n = snd (divmod num.One n)
⟨*proof*⟩

Computing congruences modulo *2 ˆ q*

lemma *cong-exp-iff-simps*:
numeral n mod numeral Num.One = 0
$\longleftrightarrow$ *True*
numeral (Num.Bit0 n) mod numeral (Num.Bit0 q) = 0
$\longleftrightarrow$ *numeral n mod numeral q = 0*
numeral (Num.Bit1 n) mod numeral (Num.Bit0 q) = 0
$\longleftrightarrow$ *False*
numeral m mod numeral Num.One = (numeral n mod numeral Num.One)
$\longleftrightarrow$ *True*
numeral Num.One mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *True*
numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *False*
numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *(numeral n mod numeral q) = 0*
numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *False*
numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *numeral m mod numeral q = (numeral n mod numeral q)*
numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *False*
numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *(numeral m mod numeral q) = 0*
numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *False*
numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
$\longleftrightarrow$ *numeral m mod numeral q = (numeral n mod numeral q)*
⟨*proof*⟩

end

56.5 Division on *nat*

context
begin

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).

inductive *eucl-rel-nat* :: $nat \Rightarrow nat \Rightarrow nat \times nat \Rightarrow bool$
where *eucl-rel-nat-by0*: *eucl-rel-nat* m 0 $(0, m)$
| *eucl-rel-natI*: $r < n \Longrightarrow m = q * n + r \Longrightarrow$ *eucl-rel-nat* m n (q, r)

eucl-rel-nat is total:

qualified lemma *eucl-rel-nat-ex*:
obtains q r **where** *eucl-rel-nat* m n (q, r)
⟨*proof*⟩

eucl-rel-nat is injective:

qualified lemma *eucl-rel-nat-unique-div*:
assumes *eucl-rel-nat* m n (q, r)
and *eucl-rel-nat* m n (q', r')
shows $q = q'$
⟨*proof*⟩ **lemma** *eucl-rel-nat-unique-mod*:
assumes *eucl-rel-nat* m n (q, r)
and *eucl-rel-nat* m n (q', r')
shows $r = r'$
⟨*proof*⟩

We instantiate divisibility on the natural numbers by means of *eucl-rel-nat*:

qualified definition *divmod-nat* :: $nat \Rightarrow nat \Rightarrow nat \times nat$ **where**
divmod-nat m n = (*THE* qr. *eucl-rel-nat* m n qr)

qualified lemma *eucl-rel-nat-divmod-nat*:
eucl-rel-nat m n (*divmod-nat* m n)
⟨*proof*⟩ **lemma** *divmod-nat-unique*:
divmod-nat m n = (q, r) **if** *eucl-rel-nat* m n (q, r)
⟨*proof*⟩ **lemma** *divmod-nat-zero*:
divmod-nat m 0 = $(0, m)$
⟨*proof*⟩ **lemma** *divmod-nat-zero-left*:
divmod-nat 0 n = $(0, 0)$
⟨*proof*⟩ **lemma** *divmod-nat-base*:
$m < n \Longrightarrow$ *divmod-nat* m n = $(0, m)$
⟨*proof*⟩ **lemma** *divmod-nat-step*:
assumes $0 < n$ **and** $n \leq m$
shows *divmod-nat* m n =
(*Suc* (*fst* (*divmod-nat* $(m - n)$ n)), *snd* (*divmod-nat* $(m - n)$ n))
⟨*proof*⟩

end

instantiation *nat* :: {*semidom-modulo*, *normalization-semidom*}
begin

definition *normalize-nat* :: $nat \Rightarrow nat$

where [*simp*]: *normalize* = (*id* :: *nat* ⇒ *nat*)

definition *unit-factor-nat* :: *nat* ⇒ *nat*
where *unit-factor n* = (*if n* = *0 then 0 else 1* :: *nat*)

lemma *unit-factor-simps* [*simp*]:
unit-factor 0 = (*0*::*nat*)
unit-factor (*Suc n*) = *1*
⟨*proof*⟩

definition *divide-nat* :: *nat* ⇒ *nat* ⇒ *nat*
where *div-nat-def*: *m div n* = *fst* (*Divides.divmod-nat m n*)

definition *modulo-nat* :: *nat* ⇒ *nat* ⇒ *nat*
where *mod-nat-def*: *m mod n* = *snd* (*Divides.divmod-nat m n*)

lemma *fst-divmod-nat* [*simp*]:
fst (*Divides.divmod-nat m n*) = *m div n*
⟨*proof*⟩

lemma *snd-divmod-nat* [*simp*]:
snd (*Divides.divmod-nat m n*) = *m mod n*
⟨*proof*⟩

lemma *divmod-nat-div-mod*:
Divides.divmod-nat m n = (*m div n*, *m mod n*)
⟨*proof*⟩

lemma *div-nat-unique*:
assumes *eucl-rel-nat m n* (*q*, *r*)
shows *m div n* = *q*
⟨*proof*⟩

lemma *mod-nat-unique*:
assumes *eucl-rel-nat m n* (*q*, *r*)
shows *m mod n* = *r*
⟨*proof*⟩

lemma *eucl-rel-nat*: *eucl-rel-nat m n* (*m div n*, *m mod n*)
⟨*proof*⟩

The "recursion" equations for *op div* and *op mod*

lemma *div-less* [*simp*]:
fixes *m n* :: *nat*
assumes *m* < *n*
shows *m div n* = *0*
⟨*proof*⟩

lemma *le-div-geq*:

fixes *m n* :: *nat*
assumes *0* < *n* **and** *n* ≤ *m*
shows *m div n* = *Suc* ((*m* − *n*) *div n*)
⟨*proof*⟩

lemma *mod-less* [*simp*]:
fixes *m n* :: *nat*
assumes *m* < *n*
shows *m mod n* = *m*
⟨*proof*⟩

lemma *le-mod-geq*:
fixes *m n* :: *nat*
assumes *n* ≤ *m*
shows *m mod n* = (*m* − *n*) *mod n*
⟨*proof*⟩

lemma *mod-less-divisor* [*simp*]:
fixes *m n* :: *nat*
assumes *n* > *0*
shows *m mod n* < *n*
⟨*proof*⟩

lemma *mod-le-divisor* [*simp*]:
fixes *m n* :: *nat*
assumes *n* > *0*
shows *m mod n* ≤ *n*
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *nat* :: *semiring-div*
⟨*proof*⟩

lemma *div-by-Suc-0* [*simp*]:
m div Suc 0 = *m*
⟨*proof*⟩

lemma *mod-by-Suc-0* [*simp*]:
m mod Suc 0 = *0*
⟨*proof*⟩

lemma *mod-greater-zero-iff-not-dvd*:
fixes *m n* :: *nat*
shows *m mod n* > *0* ⟷ ¬ *n dvd m*
⟨*proof*⟩

instantiation *nat* :: *unique-euclidean-semiring*
begin

definition [*simp*]:
euclidean-size-nat = (*id* :: *nat* ⇒ *nat*)

definition [*simp*]:
uniqueness-constraint-nat = (*top* :: *nat* ⇒ *nat* ⇒ *bool*)

instance
⟨*proof*⟩

end

Simproc for cancelling *op div* and *op mod*

lemma (**in** *semiring-modulo*) *cancel-div-mod-rules*:
((*a div b*) ∗ *b* + *a mod b*) + *c* = *a* + *c*
(*b* ∗ (*a div b*) + *a mod b*) + *c* = *a* + *c*
⟨*proof*⟩

⟨*ML*⟩

lemma *divmod-nat-if* [*code*]:
Divides.divmod-nat m n = (*if n* = *0* ∨ *m* < *n then* (*0*, *m*) *else*
let (*q*, *r*) = *Divides.divmod-nat* (*m* − *n*) *n in* (*Suc q*, *r*))
⟨*proof*⟩

lemma *mod-Suc-eq* [*mod-simps*]:
Suc (*m mod n*) *mod n* = *Suc m mod n*
⟨*proof*⟩

lemma *mod-Suc-Suc-eq* [*mod-simps*]:
Suc (*Suc* (*m mod n*)) *mod n* = *Suc* (*Suc m*) *mod n*
⟨*proof*⟩

56.5.1 Quotient

lemma *div-geq*: *0* < *n* ⟹ ¬ *m* < *n* ⟹ *m div n* = *Suc* ((*m* − *n*) *div n*)
⟨*proof*⟩

lemma *div-if*: *0* < *n* ⟹ *m div n* = (*if m* < *n then 0 else Suc* ((*m* − *n*) *div n*))
⟨*proof*⟩

lemma *div-mult-self-is-m* [*simp*]: *0*<*n* ==> (*m*∗*n*) *div n* = (*m*::*nat*)
⟨*proof*⟩

lemma *div-mult-self1-is-m* [*simp*]: *0*<*n* ==> (*n*∗*m*) *div n* = (*m*::*nat*)
⟨*proof*⟩

lemma *div-positive*:
fixes $m\ n :: nat$
assumes $n > 0$
assumes $m \geq n$
shows $m\ div\ n > 0$
⟨*proof*⟩

lemma *div-eq-0-iff*: $(a\ div\ b::nat) = 0 \longleftrightarrow a < b \lor b = 0$
⟨*proof*⟩

56.5.2 Remainder

lemma *mod-Suc-le-divisor* [*simp*]:
$m\ mod\ Suc\ n \leq n$
⟨*proof*⟩

lemma *mod-less-eq-dividend* [*simp*]:
fixes $m\ n :: nat$
shows $m\ mod\ n \leq m$
⟨*proof*⟩

lemma *mod-geq*: $\neg\ m < (n::nat) \Longrightarrow m\ mod\ n = (m - n)\ mod\ n$
⟨*proof*⟩

lemma *mod-if*: $m\ mod\ (n::nat) = (if\ m < n\ then\ m\ else\ (m - n)\ mod\ n)$
⟨*proof*⟩

56.5.3 Quotient and Remainder

lemma *div-mult1-eq*:
$(a * b)\ div\ c = a * (b\ div\ c) + a * (b\ mod\ c)\ div\ (c::nat)$
⟨*proof*⟩

lemma *eucl-rel-nat-add1-eq*:
$eucl\text{-}rel\text{-}nat\ a\ c\ (aq,\ ar) \Longrightarrow eucl\text{-}rel\text{-}nat\ b\ c\ (bq,\ br)$
$\Longrightarrow eucl\text{-}rel\text{-}nat\ (a + b)\ c\ (aq + bq + (ar + br)\ div\ c,\ (ar + br)\ mod\ c)$
⟨*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 *eucl-rel-nat-mult2-eq*:
assumes $eucl\text{-}rel\text{-}nat\ a\ b\ (q,\ r)$
shows $eucl\text{-}rel\text{-}nat\ a\ (b * c)\ (q\ div\ c,\ b *(q\ mod\ c) + r)$
⟨*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*⟩

instantiation *nat* :: *semiring-numeral-div*
begin

definition *divmod-nat* :: $num \Rightarrow num \Rightarrow nat \times nat$
where
divmod′-nat-def: $divmod\text{-}nat\ m\ n = (numeral\ m\ div\ numeral\ n,\ numeral\ m\ mod\ numeral\ n)$

definition *divmod-step-nat* :: $num \Rightarrow nat \times nat \Rightarrow nat \times nat$
where
$divmod\text{-}step\text{-}nat\ l\ qr = (let\ (q,\ r) = qr$
$in\ if\ r \geq numeral\ l\ then\ (2 * q + 1,\ r - numeral\ l)$
$else\ (2 * q,\ r))$

instance
⟨*proof*⟩

end

declare *divmod-algorithm-code* [**where** $?'a = nat$, *code*]

56.5.4 Further Facts about Quotient and Remainder

lemma *div-le-mono*:
fixes $m\ n\ k :: nat$
assumes $m \leq n$
shows $m\ div\ k \leq n\ div\ k$
⟨*proof*⟩

lemma *div-le-mono2*: $!!m{::}nat.\ [|\ 0{<}m;\ m{\leq}n\ |] ==> (k\ div\ n) \leq (k\ div\ m)$
⟨*proof*⟩

lemma *div-le-dividend* [*simp*]: $m\ div\ n \leq (m{::}nat)$
⟨*proof*⟩

lemma *div-less-dividend* [*simp*]:
$[\![(1{::}nat) < n;\ 0 < m]\!] \Longrightarrow m\ div\ n < m$
⟨*proof*⟩

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*: $(m\ mod\ d = 0) = (\exists\, q{::}nat.\ m = d{*}q)$
$\langle proof \rangle$

lemmas *mod-eq-0D* [*dest*!] = *mod-eq-0-iff* [*THEN iffD1*]

lemma *mod-eqD*:
fixes $m\ d\ r\ q :: nat$
assumes $m\ mod\ d = r$
shows $\exists\, q.\ m = r + q * d$
$\langle proof \rangle$

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** $\text{-} = (\text{-} \wedge (\text{-} \longrightarrow ?R))$)
$\langle proof \rangle$

lemma *split-div-lemma*:
assumes $0 < n$
shows $n * q \leq m \wedge m < n * Suc\ q \longleftrightarrow q = ((m{::}nat)\ div\ n)$ (**is** $?lhs \longleftrightarrow ?rhs$)
$\langle proof \rangle$

theorem *split-div'*:
$P\ ((m{::}nat)\ div\ n) = ((n = 0 \wedge P\ 0) \vee$
$(\exists\, q.\ (n * q \leq m \wedge m < n * (Suc\ q)) \wedge P\ q))$
$\langle proof \rangle$

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** $\text{-} = (\text{-} \wedge (\text{-} \longrightarrow ?R))$)
$\langle proof \rangle$

lemma *div-eq-dividend-iff*: $a \neq 0 \Longrightarrow (a :: nat)\ div\ b = a \longleftrightarrow b = 1$
$\langle proof \rangle$

lemma (**in** *field-char-0*) *of-nat-div*:
$of\text{-}nat\ (m\ div\ n) = ((of\text{-}nat\ m - of\text{-}nat\ (m\ mod\ n))\ /\ of\text{-}nat\ n)$
$\langle proof \rangle$

56.5.5 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 $P\ 0$
$\langle proof \rangle$

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 *mod2-Suc-Suc* [*simp*]: $Suc\ (Suc\ m)\ mod\ 2 = m\ mod\ 2$
  ⟨*proof*⟩

lemma *add-self-div-2* [*simp*]: $(m + m)\ div\ 2 = (m::nat)$
⟨*proof*⟩

lemma *mod2-gr-0* [*simp*]: $0 < (m::nat)\ mod\ 2 \longleftrightarrow m\ mod\ 2 = 1$
⟨*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-numeral* [*simp*] = *Suc-div-eq-add3-div* [*of - numeral v*] **for** *v*
lemmas *Suc-mod-eq-add3-mod-numeral* [*simp*] = *Suc-mod-eq-add3-mod* [*of - numeral v*] **for** *v*

lemma *Suc-times-mod-eq*: $1<k \Longrightarrow Suc\ (k * m)\ mod\ k = 1$
⟨*proof*⟩

declare *Suc-times-mod-eq* [*of numeral w*, *simp*] **for** *w*

lemma *Suc-div-le-mono* [*simp*]: $n\ div\ k \le (Suc\ n)\ div\ k$
⟨*proof*⟩

lemma *Suc-n-div-2-gt-zero* [*simp*]: $(0::nat) < n \Longrightarrow 0 < (n + 1)\ div\ 2$
⟨*proof*⟩

lemma *div-2-gt-zero* [*simp*]: **assumes** *A*: (*1*::*nat*) < *n* **shows** *0* < *n div 2*
⟨*proof*⟩

lemma *mod-mult-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*⟩

lemma *mod-2-not-eq-zero-eq-one-nat*:
  fixes *n* :: *nat*
  shows *n mod 2* ≠ *0* ⟷ *n mod 2* = *1*
  ⟨*proof*⟩

lemma *even-Suc-div-two* [*simp*]:
  even n ⟹ *Suc n div 2* = *n div 2*
  ⟨*proof*⟩

lemma *odd-Suc-div-two* [*simp*]:
  odd n ⟹ *Suc n div 2* = *Suc* (*n div 2*)
  ⟨*proof*⟩

lemma *odd-two-times-div-two-nat* [*simp*]:
  assumes *odd n*
  shows *2* ∗ (*n div 2*) = *n* − (*1* :: *nat*)
⟨*proof*⟩

lemma *parity-induct* [*case-names zero even odd*]:
  assumes *zero*: *P 0*
  assumes *even*: ⋀*n*. *P n* ⟹ *P* (*2* ∗ *n*)
  assumes *odd*: ⋀*n*. *P n* ⟹ *P* (*Suc* (*2* ∗ *n*))
  shows *P n*
⟨*proof*⟩

lemma *Suc-0-div-numeral* [*simp*]:
  fixes *k l* :: *num*
  shows *Suc 0 div numeral k* = *fst* (*divmod Num.One k*)
  ⟨*proof*⟩

lemma *Suc-0-mod-numeral* [*simp*]:
  fixes *k l* :: *num*
  shows *Suc 0 mod numeral k* = *snd* (*divmod Num.One k*)
  ⟨*proof*⟩

56.6 Division on *int*

context
begin

inductive *eucl-rel-int* :: *int* ⇒ *int* ⇒ *int* × *int* ⇒ *bool*
  where *eucl-rel-int-by0*: *eucl-rel-int k 0* (*0*, *k*)
  | *eucl-rel-int-dividesI*: $l \neq 0 \Longrightarrow k = q * l \Longrightarrow$ *eucl-rel-int k l* (*q*, *0*)
  | *eucl-rel-int-remainderI*: *sgn r* = *sgn l* $\Longrightarrow |r| < |l|$
    $\Longrightarrow k = q * l + r \Longrightarrow$ *eucl-rel-int k l* (*q*, *r*)

lemma *eucl-rel-int-iff*:
  eucl-rel-int k l (*q*, *r*) ⟷
    $k = l * q + r \wedge$
     (*if* $0 < l$ *then* $0 \leq r \wedge r < l$ *else if* $l < 0$ *then* $l < r \wedge r \leq 0$ *else* $q = 0$)
  ⟨*proof*⟩

lemma *unique-quotient-lemma*:
  $b * q' + r' \leq b * q + r \Longrightarrow 0 \leq r' \Longrightarrow r' < b \Longrightarrow r < b \Longrightarrow q' \leq (q{::}int)$
⟨*proof*⟩

lemma *unique-quotient-lemma-neg*:
  $b * q' + r' \leq b*q + r \Longrightarrow r \leq 0 \Longrightarrow b < r \Longrightarrow b < r' \Longrightarrow q \leq (q'{::}int)$
  ⟨*proof*⟩

lemma *unique-quotient*:
  eucl-rel-int a b (*q*, *r*) ⟹ *eucl-rel-int a b* (*q'*, *r'*) ⟹ $q = q'$
  ⟨*proof*⟩

lemma *unique-remainder*:
  eucl-rel-int a b (*q*, *r*) ⟹ *eucl-rel-int a b* (*q'*, *r'*) ⟹ $r = r'$
⟨*proof*⟩

end

instantiation *int* :: {*idom-modulo*, *normalization-semidom*}
begin

definition *normalize-int* :: *int* ⇒ *int*
  where [*simp*]: *normalize* = (*abs* :: *int* ⇒ *int*)

definition *unit-factor-int* :: *int* ⇒ *int*
  where [*simp*]: *unit-factor* = (*sgn* :: *int* ⇒ *int*)

definition *divide-int* :: *int* ⇒ *int* ⇒ *int*
  where *k div l* = (*if* $l = 0 \vee k = 0$ *then 0*
    else if $k > 0 \wedge l > 0 \vee k < 0 \wedge l < 0$
      then int (*nat* |*k*| *div nat* |*l*|)
      else
        if l dvd k then − *int* (*nat* |*k*| *div nat* |*l*|)
        else − *int* (*Suc* (*nat* |*k*| *div nat* |*l*|)))

definition *modulo-int* :: *int* ⇒ *int* ⇒ *int*

where $k\ mod\ l = (if\ l = 0\ then\ k\ else\ if\ l\ dvd\ k\ then\ 0$
$else\ if\ k > 0 \wedge l > 0 \vee k < 0 \wedge l < 0$
$then\ sgn\ l * int\ (nat\ |k|\ mod\ nat\ |l|)$
$else\ sgn\ l * (|l| - int\ (nat\ |k|\ mod\ nat\ |l|)))$

lemma *eucl-rel-int*:
$eucl\text{-}rel\text{-}int\ k\ l\ (k\ div\ l,\ k\ mod\ l)$
$\langle proof \rangle$

lemma *divmod-int-unique*:
assumes $eucl\text{-}rel\text{-}int\ k\ l\ (q,\ r)$
shows *div-int-unique*: $k\ div\ l = q$ **and** *mod-int-unique*: $k\ mod\ l = r$
$\langle proof \rangle$

instance $\langle proof \rangle$

end

lemma *is-unit-int*:
$is\text{-}unit\ (k::int) \longleftrightarrow k = 1 \vee k = -\ 1$
$\langle proof \rangle$

lemma *zdiv-int*:
$int\ (a\ div\ b) = int\ a\ div\ int\ b$
$\langle proof \rangle$

lemma *zmod-int*:
$int\ (a\ mod\ b) = int\ a\ mod\ int\ b$
$\langle proof \rangle$

lemma *div-abs-eq-div-nat*:
$|k|\ div\ |l| = int\ (nat\ |k|\ div\ nat\ |l|)$
$\langle proof \rangle$

lemma *mod-abs-eq-div-nat*:
$|k|\ mod\ |l| = int\ (nat\ |k|\ mod\ nat\ |l|)$
$\langle proof \rangle$

lemma *div-sgn-abs-cancel*:
fixes $k\ l\ v :: int$
assumes $v \neq 0$
shows $(sgn\ v * |k|)\ div\ (sgn\ v * |l|) = |k|\ div\ |l|$
$\langle proof \rangle$

lemma *div-eq-sgn-abs*:
fixes $k\ l\ v :: int$
assumes $sgn\ k = sgn\ l$
shows $k\ div\ l = |k|\ div\ |l|$
$\langle proof \rangle$

lemma *div-dvd-sgn-abs*:
fixes $k\ l :: int$
assumes $l\ dvd\ k$
shows $k\ div\ l = (sgn\ k * sgn\ l) * (|k|\ div\ |l|)$
⟨*proof*⟩

lemma *div-noneq-sgn-abs*:
fixes $k\ l :: int$
assumes $l \neq 0$
assumes $sgn\ k \neq sgn\ l$
shows $k\ div\ l = -\ (|k|\ div\ |l|) - of\text{-}bool\ (\neg\ l\ dvd\ k)$
⟨*proof*⟩

lemma *sgn-mod*:
fixes $k\ l :: int$
assumes $l \neq 0\ \neg\ l\ dvd\ k$
shows $sgn\ (k\ mod\ l) = sgn\ l$
⟨*proof*⟩

lemma *abs-mod-less*:
fixes $k\ l :: int$
assumes $l \neq 0$
shows $|k\ mod\ l| < |l|$
⟨*proof*⟩

instance *int* :: *ring-div*
⟨*proof*⟩

⟨*ML*⟩

Basic laws about division and remainder

lemma *pos-mod-conj*: $(0::int) < b \Longrightarrow 0 \leq a\ mod\ b \wedge a\ mod\ b < b$
⟨*proof*⟩

lemmas *pos-mod-sign* [*simp*] = *pos-mod-conj* [*THEN conjunct1*]
and *pos-mod-bound* [*simp*] = *pos-mod-conj* [*THEN conjunct2*]

lemma *neg-mod-conj*: $b < (0::int) \Longrightarrow a\ mod\ b \leq 0 \wedge b < a\ mod\ b$
⟨*proof*⟩

lemmas *neg-mod-sign* [*simp*] = *neg-mod-conj* [*THEN conjunct1*]
and *neg-mod-bound* [*simp*] = *neg-mod-conj* [*THEN conjunct2*]

56.6.1 General Properties of div and mod

lemma *div-pos-pos-trivial*: $[|\ (0::int) \leq a;\ \ a < b\ |] ==> a\ div\ b = 0$
⟨*proof*⟩

lemma *div-neg-neg-trivial*: $[| \ a \leq (0{::}int); \ \ b < a \ |] ==> a \ div \ b = 0$
⟨*proof*⟩

lemma *div-pos-neg-trivial*: $[| \ (0{::}int) < a; \ \ a{+}b \leq 0 \ |] ==> a \ div \ b = -1$
⟨*proof*⟩

lemma *mod-pos-pos-trivial*: $[| \ (0{::}int) \leq a; \ \ a < b \ |] ==> a \ mod \ b = a$
⟨*proof*⟩

lemma *mod-neg-neg-trivial*: $[| \ a \leq (0{::}int); \ \ b < a \ |] ==> a \ mod \ b = a$
⟨*proof*⟩

lemma *mod-pos-neg-trivial*: $[| \ (0{::}int) < a; \ \ a{+}b \leq 0 \ |] ==> a \ mod \ b = a{+}b$
⟨*proof*⟩

There is no *mod-neg-pos-trivial*.

56.6.2 Laws for div and mod with Unary Minus

lemma *zminus1-lemma*:
$eucl\text{-}rel\text{-}int \ a \ b \ (q, \ r) ==> b \neq 0$
$==> eucl\text{-}rel\text{-}int \ (-a) \ b \ (if \ r{=}0 \ then \ -q \ else \ -q - 1,$
$if \ r{=}0 \ then \ 0 \ else \ b{-}r)$
⟨*proof*⟩

lemma *zdiv-zminus1-eq-if*:
$b \neq (0{::}int)$
$==> (-a) \ div \ b =$
$(if \ a \ mod \ b = 0 \ then \ - (a \ div \ b) \ else \ - (a \ div \ b) - 1)$
⟨*proof*⟩

lemma *zmod-zminus1-eq-if*:
$(-a{::}int) \ mod \ b = (if \ a \ mod \ b = 0 \ then \ 0 \ else \ b - (a \ mod \ b))$
⟨*proof*⟩

lemma *zmod-zminus1-not-zero*:
fixes $k \ l :: int$
shows $- \ k \ mod \ l \neq 0 \Longrightarrow k \ mod \ l \neq 0$
⟨*proof*⟩

lemma *zmod-zminus2-not-zero*:
fixes $k \ l :: int$
shows $k \ mod - \ l \neq 0 \Longrightarrow k \ mod \ l \neq 0$
⟨*proof*⟩

lemma *zdiv-zminus2-eq-if*:

$b \neq (0::int)$
$==> a\ div\ (-b) =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ \ -\ (a\ div\ b) - 1)$
⟨*proof*⟩

lemma *zmod-zminus2-eq-if*:
$a\ mod\ (-b::int) = (if\ a\ mod\ b = 0\ then\ 0\ else\ \ (a\ mod\ b) - b)$
⟨*proof*⟩

56.6.3 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*: $[|\ a \leq a';\ \ 0 < (b::int)\ |] ==> a\ div\ b \leq a'\ div\ b$
⟨*proof*⟩

lemma *zdiv-mono1-neg*: $[|\ a \leq a';\ \ (b::int) < 0\ |] ==> a'\ div\ b \leq a\ div\ b$
⟨*proof*⟩

56.6.4 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
$[|\ 0 \leq b'*q' + r';\ r' < b';\ \ 0 < b'\ |] ==> 0 \leq (q'::int)$
⟨*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*⟩

56.6.5 More Algebraic Laws for div and mod

proving (a*b) div c = a * (b div c) + a * (b mod c)

lemma *zmult1-lemma*:
  [| *eucl-rel-int b c (q, r)* |]
  ==> *eucl-rel-int (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*⟩

proving (a+b) div c = a div c + b div c + ((a mod c + b mod c) div c)

lemma *zadd1-lemma*:
  [| *eucl-rel-int a c (aq, ar)*; *eucl-rel-int b c (bq, br)* |]
  ==> *eucl-rel-int (a+b) c (aq + bq + (ar+br) div c, (ar+br) mod c)*
⟨*proof*⟩

lemma *zdiv-zadd1-eq*:
  (a+b) div (c::int) = a div c + b div c + ((a mod c + b mod c) div c)
⟨*proof*⟩

lemma *zmod-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]*

56.6.6 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 ≤ 0* |] ==> *b ∗ c < b ∗ (q mod c) + r*
⟨*proof*⟩

lemma *zmult2-lemma-aux2*:
  [| *(0::int) < c*; *b < r*; *r ≤ 0* |] ==> *b ∗ (q mod c) + r ≤ 0*
⟨*proof*⟩

lemma *zmult2-lemma-aux3*: [| *(0::int) < c*; *0 ≤ r*; *r < b* |] ==> *0 ≤ b ∗ (q mod c) + r*
⟨*proof*⟩

lemma *zmult2-lemma-aux4*: [| *(0::int) < c*; *0 ≤ r*; *r < b* |] ==> *b ∗ (q mod c) + r < b ∗ c*
⟨*proof*⟩

lemma *zmult2-lemma*: [| *eucl-rel-int a b (q, r)*; *0 < c* |]
  ==> *eucl-rel-int a (b ∗ c) (q div c, b∗(q mod c) + r)*
⟨*proof*⟩

lemma *zdiv-zmult2-eq*:

fixes $a\ b\ c :: int$
shows $0 \leq c \Longrightarrow a\ div\ (b * c) = (a\ div\ b)\ div\ c$
⟨*proof*⟩

lemma *zmod-zmult2-eq*:
fixes $a\ b\ c :: int$
shows $0 \leq c \Longrightarrow a\ mod\ (b * c) = b * (a\ div\ b\ mod\ c) + a\ mod\ b$
⟨*proof*⟩

lemma *div-pos-geq*:
fixes $k\ l :: int$
assumes $0 < l$ **and** $l \leq k$
shows $k\ div\ l = (k - l)\ div\ l + 1$
⟨*proof*⟩

lemma *mod-pos-geq*:
fixes $k\ l :: int$
assumes $0 < l$ **and** $l \leq k$
shows $k\ mod\ l = (k - l)\ mod\ l$
⟨*proof*⟩

56.6.7 Splitting Rules for div and mod

The proofs of the two lemmas below are essentially identical

lemma *split-pos-lemma*:
$0<k ==>$
$P(n\ div\ k :: int)(n\ mod\ k) = (\forall i\ j.\ 0 \leq j\ \&\ j<k\ \&\ n = k*i + j\ --> P\ i\ j)$
⟨*proof*⟩

lemma *split-neg-lemma*:
$k<0 ==>$
$P(n\ div\ k :: int)(n\ mod\ k) = (\forall i\ j.\ k<j\ \&\ j \leq 0\ \&\ n = k*i + j\ --> P\ i\ j)$
⟨*proof*⟩

lemma *split-zdiv*:
$P(n\ div\ k :: int) =$
$((k = 0\ --> P\ 0)\ \&$
$(0<k\ --> (\forall i\ j.\ 0 \leq j\ \&\ j<k\ \&\ n = k*i + j\ --> P\ i))\ \&$
$(k<0\ --> (\forall i\ j.\ k<j\ \&\ j \leq 0\ \&\ n = k*i + j\ --> P\ i)))$
⟨*proof*⟩

lemma *split-zmod*:
$P(n\ mod\ k :: int) =$
$((k = 0\ --> P\ n)\ \&$
$(0<k\ --> (\forall i\ j.\ 0 \leq j\ \&\ j<k\ \&\ n = k*i + j\ --> P\ j))\ \&$
$(k<0\ --> (\forall i\ j.\ k<j\ \&\ j \leq 0\ \&\ n = k*i + j\ --> P\ j)))$
⟨*proof*⟩

Enable (lin)arith to deal with *op div* and *op mod* when these are applied to some constant that is of the form *numeral k*:

declare *split-zdiv* [*of - - numeral k*, *arith-split*] **for** *k*
declare *split-zmod* [*of - - numeral k*, *arith-split*] **for** *k*

56.6.8 Computing *div* and *mod* with shifting

lemma *pos-eucl-rel-int-mult-2*:
assumes $0 \leq b$
assumes *eucl-rel-int a b* (q, r)
shows *eucl-rel-int* $(1 + 2{*}a)$ $(2{*}b)$ $(q, 1 + 2{*}r)$
⟨*proof*⟩

lemma *neg-eucl-rel-int-mult-2*:
assumes $b \leq 0$
assumes *eucl-rel-int* $(a + 1)$ b (q, r)
shows *eucl-rel-int* $(1 + 2{*}a)$ $(2{*}b)$ $(q, 2{*}r - 1)$
⟨*proof*⟩

computing div by shifting

lemma *pos-zdiv-mult-2*: $(0{::}int) \leq a$ ==> $(1 + 2{*}b)$ *div* $(2{*}a) = b$ *div a*
⟨*proof*⟩

lemma *neg-zdiv-mult-2*:
assumes *A*: $a \leq (0{::}int)$ **shows** $(1 + 2{*}b)$ *div* $(2{*}a) = (b{+}1)$ *div a*
⟨*proof*⟩

lemma *zdiv-numeral-Bit0* [*simp*]:
numeral (*Num.Bit0 v*) *div numeral* (*Num.Bit0 w*) =
numeral v div (*numeral w* :: *int*)
⟨*proof*⟩

lemma *zdiv-numeral-Bit1* [*simp*]:
numeral (*Num.Bit1 v*) *div numeral* (*Num.Bit0 w*) =
(*numeral v div* (*numeral w* :: *int*))
⟨*proof*⟩

lemma *pos-zmod-mult-2*:
fixes *a b* :: *int*
assumes $0 \leq a$
shows $(1 + 2 * b)$ *mod* $(2 * a) = 1 + 2 * (b$ *mod* $a)$
⟨*proof*⟩

lemma *neg-zmod-mult-2*:
fixes *a b* :: *int*
assumes $a \leq 0$
shows $(1 + 2 * b)$ *mod* $(2 * a) = 2 * ((b + 1)$ *mod* $a) - 1$
⟨*proof*⟩

lemma *zmod-numeral-Bit0* [*simp*]:
  numeral (*Num.Bit0 v*) *mod numeral* (*Num.Bit0 w*) =
    (*2*::*int*) ∗ (*numeral v mod numeral w*)
  ⟨*proof*⟩

lemma *zmod-numeral-Bit1* [*simp*]:
  numeral (*Num.Bit1 v*) *mod numeral* (*Num.Bit0 w*) =
    2 ∗ (*numeral v mod numeral w*) + (*1*::*int*)
  ⟨*proof*⟩

lemma *zdiv-eq-0-iff*:
 (*i*::*int*) *div k* = *0* ⟷ *k*=*0* ∨ *0*≤*i* ∧ *i*<*k* ∨ *i*≤*0* ∧ *k*<*i* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *zmod-trival-iff*:
  fixes *i k* :: *int*
  shows *i mod k* = *i* ⟷ *k* = *0* ∨ *0* ≤ *i* ∧ *i* < *k* ∨ *i* ≤ *0* ∧ *k* < *i*
⟨*proof*⟩

instantiation *int* :: *unique-euclidean-ring*
begin

definition [*simp*]:
  euclidean-size-int = (*nat* ∘ *abs* :: *int* ⇒ *nat*)

definition [*simp*]:
  uniqueness-constraint-int (*k* :: *int*) *l* ⟷ *unit-factor k* = *unit-factor l*

instance
  ⟨*proof*⟩

end

56.6.9 Quotients of Signs

lemma *div-eq-minus1*: (*0*::*int*) < *b* ==> −*1 div b* = −*1*
⟨*proof*⟩

lemma *zmod-minus1*: (*0*::*int*) < *b* ==> −*1 mod b* = *b* − *1*
⟨*proof*⟩

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*: $(0{::}int) < b ==> (0 \leq a\ div\ b) = (0 \leq a)$
⟨*proof*⟩

lemma *pos-imp-zdiv-pos-iff*:
$0{<}k \Longrightarrow 0 < (i{::}int)\ div\ k \longleftrightarrow k \leq i$
⟨*proof*⟩

lemma *neg-imp-zdiv-nonneg-iff*:
$b < (0{::}int) ==> (0 \leq a\ div\ b) = (a \leq (0{::}int))$
⟨*proof*⟩

lemma *pos-imp-zdiv-neg-iff*: $(0{::}int) < b ==> (a\ div\ b < 0) = (a < 0)$
⟨*proof*⟩

lemma *neg-imp-zdiv-neg-iff*: $b < (0{::}int) ==> (a\ div\ b < 0) = (0 < a)$
⟨*proof*⟩

lemma *nonneg1-imp-zdiv-pos-iff*:
$(0{::}int) <= a \Longrightarrow (a\ div\ b > 0) = (a >= b\ \&\ b{>}0)$
⟨*proof*⟩

lemma *zmod-le-nonneg-dividend*: $(m{::}int) \geq 0 ==> m\ mod\ k \leq m$
⟨*proof*⟩

56.6.10 Computation of Division and Remainder

instantiation *int* :: *semiring-numeral-div*
begin

definition *divmod-int* :: $num \Rightarrow num \Rightarrow int \times int$
where
divmod-int m n = (*numeral m div numeral n*, *numeral m mod numeral n*)

definition *divmod-step-int* :: $num \Rightarrow int \times int \Rightarrow int \times int$
where
divmod-step-int l qr = (*let* (*q*, *r*) = *qr*
in if $r \geq$ *numeral l then* $(2 * q + 1,\ r -$ *numeral l*)
else $(2 * q,\ r))$

instance
⟨*proof*⟩

end

declare *divmod-algorithm-code* [**where** *?'a = int*, *code*]

context
begin

qualified definition *adjust-div* :: *int* × *int* ⇒ *int*
where
adjust-div qr = (*let* (*q*, *r*) = *qr in q* + *of-bool* (*r* ≠ *0*))

qualified lemma *adjust-div-eq* [*simp*, *code*]:
adjust-div (*q*, *r*) = *q* + *of-bool* (*r* ≠ *0*)
⟨*proof*⟩ **definition** *adjust-mod* :: *int* ⇒ *int* ⇒ *int*
where
[*simp*]: *adjust-mod l r* = (*if r* = *0 then 0 else l* − *r*)

lemma *minus-numeral-div-numeral* [*simp*]:
− *numeral m div numeral n* = − (*adjust-div* (*divmod m n*) :: *int*)
⟨*proof*⟩

lemma *minus-numeral-mod-numeral* [*simp*]:
− *numeral m mod numeral n* = *adjust-mod* (*numeral n*) (*snd* (*divmod m n*) :: *int*)
⟨*proof*⟩

lemma *numeral-div-minus-numeral* [*simp*]:
numeral m div − *numeral n* = − (*adjust-div* (*divmod m n*) :: *int*)
⟨*proof*⟩

lemma *numeral-mod-minus-numeral* [*simp*]:
numeral m mod − *numeral n* = − *adjust-mod* (*numeral n*) (*snd* (*divmod m n*) :: *int*)
⟨*proof*⟩

lemma *minus-one-div-numeral* [*simp*]:
− *1 div numeral n* = − (*adjust-div* (*divmod Num.One n*) :: *int*)
⟨*proof*⟩

lemma *minus-one-mod-numeral* [*simp*]:
− *1 mod numeral n* = *adjust-mod* (*numeral n*) (*snd* (*divmod Num.One n*) :: *int*)
⟨*proof*⟩

lemma *one-div-minus-numeral* [*simp*]:
1 div − *numeral n* = − (*adjust-div* (*divmod Num.One n*) :: *int*)
⟨*proof*⟩

lemma *one-mod-minus-numeral* [*simp*]:
1 mod − *numeral n* = − *adjust-mod* (*numeral n*) (*snd* (*divmod Num.One n*) ::

int)
  ⟨*proof*⟩

end

56.6.11 Further properties

Simplify expresions in which div and mod combine numerical constants

lemma *int-div-pos-eq*: $[\![(a::int) = b * q + r;\ 0 \le r;\ r < b]\!] \Longrightarrow a\ div\ b = q$
  ⟨*proof*⟩

lemma *int-div-neg-eq*: $[\![(a::int) = b * q + r;\ r \le 0;\ b < r]\!] \Longrightarrow a\ div\ b = q$
  ⟨*proof*⟩

lemma *int-mod-pos-eq*: $[\![(a::int) = b * q + r;\ 0 \le r;\ r < b]\!] \Longrightarrow a\ mod\ b = r$
  ⟨*proof*⟩

lemma *int-mod-neg-eq*: $[\![(a::int) = b * q + r;\ r \le 0;\ b < r]\!] \Longrightarrow a\ mod\ b = r$
  ⟨*proof*⟩

lemma *abs-div*: $(y::int)\ dvd\ x \Longrightarrow |x\ div\ y| = |x|\ div\ |y|$
⟨*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*⟩

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 (**in** *ring-div*) *mod-eq-dvd-iff*:
$a\ mod\ c = b\ mod\ c \longleftrightarrow c\ dvd\ a - b$ (**is** $?P \longleftrightarrow ?Q$)
⟨*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*⟩

56.6.12 Dedicated simproc for calculation

There is space for improvement here: the calculation itself could be carried outside the logic, and a generic simproc (simplifier setup) for generic calculation would be helpful.

⟨*ML*⟩

56.6.13 Code generation

lemma [*code*]:

fixes *k* :: *int*
shows
k div 0 = 0
k mod 0 = k
0 div k = 0
0 mod k = 0
k div Int.Pos Num.One = k
k mod Int.Pos Num.One = 0
k div Int.Neg Num.One = − k
k mod Int.Neg Num.One = 0
Int.Pos m div Int.Pos n = (fst (divmod m n) :: int)
Int.Pos m mod Int.Pos n = (snd (divmod m n) :: int)
Int.Neg m div Int.Pos n = − (Divides.adjust-div (divmod m n) :: int)
Int.Neg m mod Int.Pos n = Divides.adjust-mod (Int.Pos n) (snd (divmod m n) :: int)
Int.Pos m div Int.Neg n = − (Divides.adjust-div (divmod m n) :: int)
Int.Pos m mod Int.Neg n = − Divides.adjust-mod (Int.Pos n) (snd (divmod m n) :: int)
Int.Neg m div Int.Neg n = (fst (divmod m n) :: int)
Int.Neg m mod Int.Neg n = − (snd (divmod m n) :: int)
⟨*proof*⟩

code-identifier
code-module *Divides* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

lemma *dvd-eq-mod-eq-0-numeral*:
numeral x dvd (numeral y :: 'a) ⟷ numeral y mod numeral x = (0 :: 'a::semiring-div)
⟨*proof*⟩

declare *minus-div-mult-eq-mod* [*symmetric*, *nitpick-unfold*]

end

57 Combination and Cancellation Simprocs for Numeral Expressions

theory *Numeral-Simprocs*
imports *Divides*
begin

⟨*ML*⟩

lemmas *semiring-norm* =
Let-def arith-simps diff-nat-numeral rel-simps
if-False if-True
add-0 add-Suc add-numeral-left
add-neg-numeral-left mult-numeral-left
numeral-One [*symmetric*] *uminus-numeral-One* [*symmetric*] *Suc-eq-plus1*

eq-numeral-iff-iszero not-iszero-Numeral1

declare *split-div* [*of - - numeral k*, *arith-split*] **for** *k*
declare *split-mod* [*of - - numeral k*, *arith-split*] **for** *k*

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*

lemmas *nat-mult-le-cancel-disj* = *mult-le-cancel1*

lemmas *nat-mult-less-cancel-disj* = *mult-less-cancel1*

lemma *nat-mult-eq-cancel-disj*:
fixes k m n :: *nat*
shows $k * m = k * n \longleftrightarrow k = 0 \vee m = n$
⟨*proof*⟩

lemma *nat-mult-div-cancel-disj* [*simp*]:
fixes k m n :: *nat*
shows $(k * m)$ *div* $(k * n) = ($*if* $k = 0$ *then* 0 *else* m *div* $n)$
⟨*proof*⟩

lemma *numeral-times-minus-swap*:
fixes x:: *'a::comm-ring-1* **shows** *numeral* $w * -x = x * -$ *numeral* w
⟨*proof*⟩

⟨*ML*⟩

end

58 Semiring normalization

theory *Semiring-Normalization*
imports *Numeral-Simprocs Nat-Transfer*
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

subclass (**in** *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 *semiring-normalization-rules*:
$(a * m) + (b * m) = (a + b) * m$
$(a * m) + m = (a + 1) * m$
$m + (a * m) = (a + 1) * m$
$m + m = (1 + 1) * m$
$0 + a = a$
$a + 0 = a$
$a * b = b * a$
$(a + b) * c = (a * c) + (b * c)$
$0 * a = 0$
$a * 0 = 0$
$1 * a = a$
$a * 1 = a$
$(lx * ly) * (rx * ry) = (lx * rx) * (ly * ry)$
$(lx * ly) * (rx * ry) = lx * (ly * (rx * ry))$
$(lx * ly) * (rx * ry) = rx * ((lx * ly) * ry)$
$(lx * ly) * rx = (lx * rx) * ly$
$(lx * ly) * rx = lx * (ly * rx)$
$lx * (rx * ry) = (lx * rx) * ry$
$lx * (rx * ry) = rx * (lx * ry)$
$(a + b) + (c + d) = (a + c) + (b + d)$
$(a + b) + c = a + (b + c)$
$a + (c + d) = c + (a + d)$
$(a + b) + c = (a + c) + b$

$a + c = c + a$
$a + (c + d) = (a + c) + d$
$(x \hat{} p) * (x \hat{} q) = x \hat{} (p + q)$
$x * (x \hat{} q) = x \hat{} (Suc\ q)$
$(x \hat{} q) * x = x \hat{} (Suc\ q)$
$x * x = x^2$
$(x * y) \hat{} q = (x \hat{} q) * (y \hat{} q)$
$(x \hat{} p) \hat{} q = x \hat{} (p * q)$
$x \hat{} 0 = 1$
$x \hat{} 1 = x$
$x * (y + z) = (x * y) + (x * z)$
$x \hat{} (Suc\ q) = x * (x \hat{} q)$
$x \hat{} (2*n) = (x \hat{} n) * (x \hat{} n)$
⟨*proof*⟩

⟨*ML*⟩

end

context *comm-ring-1*
begin

lemma *ring-normalization-rules*:
$- x = (- 1) * x$
$x - y = x + (- y)$
⟨*proof*⟩

⟨*ML*⟩

end

context *comm-semiring-1-cancel-crossproduct*
begin

⟨*ML*⟩

end

context *idom*
begin

⟨*ML*⟩

end

context *field*
begin

⟨*ML*⟩

end

code-identifier
code-module *Semiring-Normalization* $\rightharpoonup$ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

59 Groebner bases

theory *Groebner-Basis*
imports *Semiring-Normalization Parity*
begin

59.1 Groebner Bases

lemmas *bool-simps* = *simp-thms*(*1*−*34*) — FIXME move to *HOL*

lemma *nnf-simps*: — FIXME shadows fact binding in *HOL*
$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$ $(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$
$(P \longrightarrow Q) = (\neg P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg\, Q))$ $(\neg\, \neg(P)) = P$
⟨*proof*⟩

lemma *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 *PFalse*:
$P \equiv False \Longrightarrow \neg\, P$
$\neg\, P \Longrightarrow (P \equiv False)$
⟨*proof*⟩

named-theorems *algebra pre*−*simplification rules for algebraic methods*
⟨*ML*⟩

declare *dvd-def*[*algebra*]
declare *mod-eq-0-iff-dvd*[*algebra*]
declare *mod-div-trivial*[*algebra*]
declare *mod-mod-trivial*[*algebra*]
declare *div-by-0*[*algebra*]
declare *mod-by-0*[*algebra*]
declare *mult-div-mod-eq*[*algebra*]

declare *div-minus-minus*[*algebra*]
declare *mod-minus-minus*[*algebra*]
declare *div-minus-right*[*algebra*]
declare *mod-minus-right*[*algebra*]
declare *div-0*[*algebra*]
declare *mod-0*[*algebra*]
declare *mod-by-1*[*algebra*]
declare *div-by-1*[*algebra*]
declare *mod-minus1-right*[*algebra*]
declare *div-minus1-right*[*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 *mod-eq-dvd-iff*[*algebra*]
declare *nat-mod-eq-iff*[*algebra*]

context *semiring-parity*
begin

declare *even-times-iff* [*algebra*]
declare *even-power* [*algebra*]

end

context *ring-parity*
begin

declare *even-minus* [*algebra*]

end

declare *even-Suc* [*algebra*]
declare *even-diff-nat* [*algebra*]

end

60 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets

theory *Lattices-Big*
imports *Option*
begin

60.1 Generic lattice operations over a set

60.1.1 Without neutral element

locale *semilattice-set* = *semilattice*
begin

interpretation *comp-fun-idem f*
⟨*proof*⟩

definition F :: $'a$ *set* $\Rightarrow$ $'a$
where
eq-fold': $F\ A = $ *the* (*Finite-Set.fold* ($\lambda x\ y.$ *Some* (*case* y *of None* $\Rightarrow x \mid$ *Some* z $\Rightarrow f\ x\ z$)) *None* A)

lemma *eq-fold*:
assumes *finite* A
shows F (*insert* x A) = *Finite-Set.fold* $f\ x\ A$
⟨*proof*⟩

lemma *singleton* [*simp*]:
$F\ \{x\} = x$
⟨*proof*⟩

lemma *insert-not-elem*:
assumes *finite* A **and** $x \notin A$ **and** $A \neq \{\}$
shows F (*insert* x A) = $x * F\ A$
⟨*proof*⟩

lemma *in-idem*:
assumes *finite* A **and** $x \in A$
shows $x * F\ A = F\ A$
⟨*proof*⟩

lemma *insert* [*simp*]:
assumes *finite* A **and** $A \neq \{\}$
shows F (*insert* x A) = $x * F\ A$
⟨*proof*⟩

lemma *union*:
assumes *finite* A $A \neq \{\}$ **and** *finite* B $B \neq \{\}$
shows $F\ (A \cup B) = F\ A * F\ B$
⟨*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 *subset*:
assumes *finite* A $B \neq \{\}$ **and** $B \subseteq A$
shows $F\ B * F\ A = F\ A$
⟨*proof*⟩

lemma *closed*:
assumes *finite* A $A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x * y \in \{x, y\}$
shows $F\ A \in A$
⟨*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*⟩

lemma *infinite*: ¬ *finite* $A \Longrightarrow F\ A =$ *the None*
⟨*proof*⟩

end

locale *semilattice-order-set* = *binary?*: *semilattice-order* + *semilattice-set*
begin

lemma *bounded-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $x \leq F\ A \longleftrightarrow (\forall\, a \in A.\ x \leq a)$
⟨*proof*⟩

lemma *boundedI*:
assumes *finite* A
assumes $A \neq \{\}$
assumes $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
shows $x \leq F\ A$
⟨*proof*⟩

lemma *boundedE*:
assumes *finite* A **and** $A \neq \{\}$ **and** $x \leq F\ A$
obtains $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
⟨*proof*⟩

lemma *coboundedI*:
assumes *finite* A
and $a \in A$
shows $F\ A \leq a$

⟨*proof*⟩

lemma *antimono*:
  assumes $A \subseteq B$ **and** $A \neq \{\}$ **and** *finite* B
  shows $F\ B \leq F\ A$
⟨*proof*⟩

end

60.1.2 With neutral element

locale *semilattice-neutr-set* = *semilattice-neutr*
begin

interpretation *comp-fun-idem f*
  ⟨*proof*⟩

definition $F :: 'a\ set \Rightarrow 'a$
where
  eq-fold: $F\ A = Finite\text{-}Set.fold\ f\ \mathbf{1}\ A$

lemma *infinite* [*simp*]:
  $\neg$ *finite* $A \Longrightarrow F\ A = \mathbf{1}$
  ⟨*proof*⟩

lemma *empty* [*simp*]:
  $F\ \{\} = \mathbf{1}$
  ⟨*proof*⟩

lemma *insert* [*simp*]:
  assumes *finite* A
  shows $F\ (insert\ x\ A) = x * F\ A$
  ⟨*proof*⟩

lemma *in-idem*:
  assumes *finite* A **and** $x \in A$
  shows $x * F\ A = F\ A$
⟨*proof*⟩

lemma *union*:
  assumes *finite* A **and** *finite* B
  shows $F\ (A \cup B) = F\ A * F\ B$
  ⟨*proof*⟩

lemma *remove*:
  assumes *finite* A **and** $x \in A$
  shows $F\ A = x * F\ (A - \{x\})$
⟨*proof*⟩

lemma *insert-remove*:
  assumes *finite A*
  shows $F\ (insert\ x\ A) = x * F\ (A - \{x\})$
  ⟨*proof*⟩

lemma *subset*:
  assumes *finite A* **and** $B \subseteq A$
  shows $F\ B * F\ A = F\ A$
⟨*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

locale *semilattice-order-neutr-set* = *binary?*: *semilattice-neutr-order* + *semilattice-neutr-set*
begin

lemma *bounded-iff*:
  assumes *finite A*
  shows $x \leq F\ A \longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
  ⟨*proof*⟩

lemma *boundedI*:
  assumes *finite A*
  assumes $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
  shows $x \leq F\ A$
  ⟨*proof*⟩

lemma *boundedE*:
  assumes *finite A* **and** $x \leq F\ A$
  obtains $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
  ⟨*proof*⟩

lemma *coboundedI*:
  assumes *finite A*
    and $a \in A$
  shows $F\ A \leq a$
⟨*proof*⟩

lemma *antimono*:
  assumes $A \subseteq B$ **and** *finite B*
  shows $F\ B \leq F\ A$
⟨*proof*⟩

end

60.2 Lattice operations on finite sets

context *semilattice-inf*
begin

sublocale *Inf-fin*: *semilattice-order-set inf less-eq less*
defines
Inf-fin ($\bigsqcap_{fin}$- [900] 900) = *Inf-fin.F* ⟨*proof*⟩

end

context *semilattice-sup*
begin

sublocale *Sup-fin*: *semilattice-order-set sup greater-eq greater*
defines
Sup-fin ($\bigsqcup_{fin}$- [900] 900) = *Sup-fin.F* ⟨*proof*⟩

end

60.3 Infimum and Supremum over non-empty sets

context *lattice*
begin

lemma *Inf-fin-le-Sup-fin* [*simp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
⟨*proof*⟩

lemma *sup-Inf-absorb* [*simp*]:
finite $A \Longrightarrow a \in A \Longrightarrow \bigsqcap_{fin} A \sqcup a = a$
⟨*proof*⟩

lemma *inf-Sup-absorb* [*simp*]:
finite $A \Longrightarrow a \in A \Longrightarrow a \sqcap \bigsqcup_{fin} A = a$
⟨*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 $\bigsqcap_{fin} A = \bigsqcap A$
⟨*proof*⟩

lemma *Sup-fin-Sup*:
assumes *finite* A **and** $A \neq \{\}$
shows $\bigsqcup_{fin} A = \bigsqcup A$
⟨*proof*⟩

end

60.4 Minimum and Maximum over non-empty sets

context *linorder*
begin

sublocale *Min*: *semilattice-order-set min less-eq less*
\+ *Max*: *semilattice-order-set max greater-eq greater*
defines
Min = *Min.F* **and** *Max* = *Max.F* ⟨*proof*⟩

end

An aside: *Min*/*Max* on linear orders as special case of *Inf-fin*/*Sup-fin*

lemma *Inf-fin-Min*:
Inf-fin = (*Min* :: $'a$::{*semilattice-inf*, *linorder*} *set* ⇒ $'a$)
⟨*proof*⟩

lemma *Sup-fin-Max*:

$Sup\text{-}fin = (Max :: 'a::\{semilattice\text{-}sup,\ linorder\}\ set \Rightarrow 'a)$
⟨proof⟩

context *linorder*
begin

lemma *dual-min*:
$ord.min\ greater\text{-}eq = max$
⟨proof⟩

lemma *dual-max*:
$ord.max\ greater\text{-}eq = min$
⟨proof⟩

lemma *dual-Min*:
$linorder.Min\ greater\text{-}eq = Max$
⟨proof⟩

lemma *dual-Max*:
$linorder.Max\ greater\text{-}eq = Min$
⟨proof⟩

lemmas *Min-singleton* = *Min.singleton*
lemmas *Max-singleton* = *Max.singleton*
lemmas *Min-insert* = *Min.insert*
lemmas *Max-insert* = *Max.insert*
lemmas *Min-Un* = *Min.union*
lemmas *Max-Un* = *Max.union*
lemmas *hom-Min-commute* = *Min.hom-commute*
lemmas *hom-Max-commute* = *Max.hom-commute*

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-insert2*:
assumes *finite A* **and** *min*: $\bigwedge b.\ b \in A \Longrightarrow a \leq b$
shows $Min\ (insert\ a\ A) = a$
⟨proof⟩

lemma *Max-insert2*:
assumes *finite A* **and** *max*: $\bigwedge b.\ b \in A \Longrightarrow b \leq a$
shows $Max\ (insert\ a\ A) = a$

⟨*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-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 *eq-Min-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow m = Min\ A \longleftrightarrow m \in A \wedge (\forall a \in A.\ m \leq a)$
⟨*proof*⟩

lemma *Min-eq-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Min\ A = m \longleftrightarrow m \in A \wedge (\forall a \in A.\ m \leq a)$
⟨*proof*⟩

lemma *eq-Max-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow m = Max\ A \longleftrightarrow m \in A \wedge (\forall a \in A.\ a \leq m)$
⟨*proof*⟩

lemma *Max-eq-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Max\ A = m \longleftrightarrow m \in A \wedge (\forall a \in A.\ a \leq m)$
⟨*proof*⟩

context
fixes $A :: 'a\ set$
assumes *fin-nonempty*: *finite* A $A \neq \{\}$
begin

lemma *Min-ge-iff* [*simp*]:
$x \leq Min\ A \longleftrightarrow (\forall a{\in}A.\ x \leq a)$

⟨*proof*⟩

lemma *Max-le-iff* [*simp*]:
$Max\ A \leq x \longleftrightarrow (\forall\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *Min-gr-iff* [*simp*]:
$x < Min\ A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
⟨*proof*⟩

lemma *Max-less-iff* [*simp*]:
$Max\ A < x \longleftrightarrow (\forall\, a{\in}A.\ a < x)$
⟨*proof*⟩

lemma *Min-le-iff*:
$Min\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
⟨*proof*⟩

lemma *Max-ge-iff*:
$x \leq Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$
⟨*proof*⟩

lemma *Min-less-iff*:
$Min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
⟨*proof*⟩

lemma *Max-gr-iff*:
$x < Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$
⟨*proof*⟩

end

lemma *Max-eq-if*:
assumes *finite A* *finite B* $\forall\, a{\in}A.\ \exists\, b{\in}B.\ a \leq b$ $\forall\, b{\in}B.\ \exists\, a{\in}A.\ b \leq a$
shows $Max\ A = Max\ B$
⟨*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*⟩

end

context *linorder*
begin

lemma *mono-Min-commute*:
  assumes *mono f*
  assumes *finite A* **and** $A \neq \{\}$
  shows $f\ (Min\ A) = Min\ (f\ `\ A)$
⟨*proof*⟩

lemma *mono-Max-commute*:
  assumes *mono f*
  assumes *finite A* **and** $A \neq \{\}$
  shows $f\ (Max\ A) = Max\ (f\ `\ A)$
⟨*proof*⟩

lemma *finite-linorder-max-induct* [*consumes 1*, *case-names empty insert*]:
  assumes *fin*: *finite A*
  and *empty*: $P\ \{\}$
  and *insert*: $\bigwedge b\ A.\ finite\ A \Longrightarrow \forall a{\in}A.\ a < b \Longrightarrow P\ A \Longrightarrow P\ (insert\ b\ A)$
  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*⟩

lemma *Least-Min*:
  assumes *finite* $\{a.\ P\ a\}$ **and** $\exists a.\ P\ a$
  shows $(LEAST\ a.\ P\ a) = Min\ \{a.\ P\ a\}$
⟨*proof*⟩

lemma *infinite-growing*:
  assumes $X \neq \{\}$
  assumes $*$: $\bigwedge x.\ x \in X \Longrightarrow \exists y{\in}X.\ y > x$
  shows $\neg$ *finite X*
⟨*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 $k + Max\ N = Max\ \{k + m \mid m.\ m \in N\}$
$\langle proof \rangle$

end

context *linordered-ab-group-add*
begin

lemma *minus-Max-eq-Min* [*simp*]:
  $finite\ S \Longrightarrow S \neq \{\} \Longrightarrow -\ Max\ S = Min\ (uminus\ `\ S)$
  $\langle proof \rangle$

lemma *minus-Min-eq-Max* [*simp*]:
  $finite\ S \Longrightarrow S \neq \{\} \Longrightarrow -\ Min\ S = Max\ (uminus\ `\ S)$
  $\langle proof \rangle$

end

context *complete-linorder*
begin

lemma *Min-Inf*:
  assumes *finite* A **and** $A \neq \{\}$
  shows $Min\ A = Inf\ A$
$\langle proof \rangle$

lemma *Max-Sup*:
  assumes *finite* A **and** $A \neq \{\}$
  shows $Max\ A = Sup\ A$
$\langle proof \rangle$

end

60.5 Arg Min

definition *is-arg-min* :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
$is\text{-}arg\text{-}min\ f\ P\ x = (P\ x \wedge \neg(\exists\, y.\ P\ y \wedge f\ y < f\ x))$

definition *arg-min* :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
$arg\text{-}min\ f\ P = (SOME\ x.\ is\text{-}arg\text{-}min\ f\ P\ x)$

abbreviation *arg-min-on* :: $('a \Rightarrow 'b::ord) \Rightarrow 'a\ set \Rightarrow 'a$ **where**
$arg\text{-}min\text{-}on\ f\ S \equiv arg\text{-}min\ f\ (\lambda x.\ x \in S)$

syntax
  $\text{-}arg\text{-}min :: ('a \Rightarrow 'b) \Rightarrow pttrn \Rightarrow bool \Rightarrow 'a$

$((3ARG'\text{-}MIN \text{ - -./ -}) [0, 0, 10] 10)$
translations
$ARG\text{-}MIN\ f\ x.\ P \rightleftharpoons CONST\ arg\text{-}min\ f\ (\lambda x.\ P)$

lemma *is-arg-min-linorder*: **fixes** $f :: 'a \Rightarrow 'b :: linorder$
shows $is\text{-}arg\text{-}min\ f\ P\ x = (P\ x \wedge (\forall y.\ P\ y \longrightarrow f\ x \leq f\ y))$
$\langle proof \rangle$

lemma *arg-minI*:
$[\![P\ x;$
$\bigwedge y.\ P\ y \Longrightarrow \neg\ f\ y < f\ x;$
$\bigwedge x.\ [\![P\ x; \forall y.\ P\ y \longrightarrow \neg\ f\ y < f\ x]\!] \Longrightarrow Q\ x]\!]$
$\Longrightarrow Q\ (arg\text{-}min\ f\ P)$
$\langle proof \rangle$

lemma *arg-min-equality*:
$[\![P\ k; \bigwedge x.\ P\ x \Longrightarrow f\ k \leq f\ x]\!] \Longrightarrow f\ (arg\text{-}min\ f\ P) = f\ k$
for $f :: \text{-} \Rightarrow 'a::order$
$\langle proof \rangle$

lemma *wf-linord-ex-has-least*:
$[\![wf\ r; \forall x\ y.\ (x, y) \in r^+ \longleftrightarrow (y, x) \notin r^*; P\ k]\!]$
$\Longrightarrow \exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow (m\ x, m\ y) \in r^*)$
$\langle proof \rangle$

lemma *ex-has-least-nat*: $P\ k \Longrightarrow \exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow m\ x \leq m\ y)$
for $m :: 'a \Rightarrow nat$
$\langle proof \rangle$

lemma *arg-min-nat-lemma*:
$P\ k \Longrightarrow P(arg\text{-}min\ m\ P) \wedge (\forall y.\ P\ y \longrightarrow m\ (arg\text{-}min\ m\ P) \leq m\ y)$
for $m :: 'a \Rightarrow nat$
$\langle proof \rangle$

lemmas $arg\text{-}min\text{-}natI = arg\text{-}min\text{-}nat\text{-}lemma\ [THEN\ conjunct1]$

lemma *is-arg-min-arg-min-nat*: **fixes** $m :: 'a \Rightarrow nat$
shows $P\ x \Longrightarrow is\text{-}arg\text{-}min\ m\ P\ (arg\text{-}min\ m\ P)$
$\langle proof \rangle$

lemma *arg-min-nat-le*: $P\ x \Longrightarrow m\ (arg\text{-}min\ m\ P) \leq m\ x$
for $m :: 'a \Rightarrow nat$
$\langle proof \rangle$

lemma *ex-min-if-finite*:
$[\![finite\ S; S \neq \{\}]\!] \Longrightarrow \exists m{\in}S.\ \neg(\exists x{\in}S.\ x < (m::'a::order))$
$\langle proof \rangle$

lemma *ex-is-arg-min-if-finite*: **fixes** $f :: 'a \Rightarrow 'b :: order$

shows $\llbracket$ *finite* S; $S \neq \{\}$ $\rrbracket \Longrightarrow \exists\, x.$ *is-arg-min* f $(\lambda x.\ x : S)$ x
⟨*proof*⟩

lemma *arg-min-SOME-Min*:
finite $S \Longrightarrow$ *arg-min-on* f S = (*SOME* $y.$ $y \in S \wedge f\ y = Min(f$ ‘ $S))$
⟨*proof*⟩

lemma *arg-min-if-finite*: **fixes** f :: $'a \Rightarrow 'b$ :: *order*
assumes *finite* S $S \neq \{\}$
shows *arg-min-on* f $S \in S$ **and** $\neg(\exists\, x{\in}S.\ f\ x < f$ (*arg-min-on* f S))
⟨*proof*⟩

lemma *arg-min-least*: **fixes** f :: $'a \Rightarrow 'b$ :: *linorder*
shows $\llbracket$ *finite* S; $S \neq \{\}$; $y \in S$ $\rrbracket \Longrightarrow f($*arg-min-on* f $S) \leq f\ y$
⟨*proof*⟩

lemma *arg-min-inj-eq*: **fixes** f :: $'a \Rightarrow 'b$ :: *order*
shows $\llbracket$ *inj-on* f $\{x.\ P\ x\}$; $P\ a$; $\forall\, y.\ P\ y \longrightarrow f\ a \leq f\ y$ $\rrbracket \Longrightarrow$ *arg-min* f $P = a$
⟨*proof*⟩

60.6 Arg Max

definition *is-arg-max* :: $('a \Rightarrow 'b{::}ord) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
is-arg-max f P x = $(P\ x \wedge \neg(\exists\, y.\ P\ y \wedge f\ y > f\ x))$

definition *arg-max* :: $('a \Rightarrow 'b{::}ord) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
arg-max f P = (*SOME* $x.$ *is-arg-max* f P x)

abbreviation *arg-max-on* :: $('a \Rightarrow 'b{::}ord) \Rightarrow 'a\ set \Rightarrow 'a$ **where**
arg-max-on f $S \equiv$ *arg-max* f $(\lambda x.\ x \in S)$

syntax
-*arg-max* :: $('a \Rightarrow 'b) \Rightarrow pttrn \Rightarrow bool \Rightarrow 'a$
((*3ARG'-MAX* - -./ -) [*0*, *0*, *10*] *10*)
translations
ARG-MAX f $x.$ $P \rightleftharpoons$ *CONST arg-max* f $(\lambda x.\ P)$

lemma *is-arg-max-linorder*: **fixes** f :: $'a \Rightarrow 'b$ :: *linorder*
shows *is-arg-max* f P x = $(P\ x \wedge (\forall\, y.\ P\ y \longrightarrow f\ x \geq f\ y))$
⟨*proof*⟩

lemma *arg-maxI*:
$P\ x \Longrightarrow$
$(\bigwedge y.\ P\ y \Longrightarrow \neg\ f\ y > f\ x) \Longrightarrow$
$(\bigwedge x.\ P\ x \Longrightarrow \forall\, y.\ P\ y \longrightarrow \neg\ f\ y > f\ x \Longrightarrow Q\ x) \Longrightarrow$
Q (*arg-max* f P)
⟨*proof*⟩

lemma *arg-max-equality*:

$\llbracket\ P\ k;\ \bigwedge x.\ P\ x \Longrightarrow f\,x \leq f\,k\ \rrbracket \Longrightarrow f\ (\textit{arg-max}\ f\ P) = f\,k$
for f :: - $\Rightarrow$ $'a::order$
⟨*proof*⟩

lemma *ex-has-greatest-nat-lemma*:
$P\ k \Longrightarrow \forall x.\ P\ x \longrightarrow (\exists y.\ P\ y \wedge \neg\ f\ y \leq f\ x) \Longrightarrow \exists y.\ P\ y \wedge \neg\ f\ y < f\ k + n$
for f :: $'a \Rightarrow nat$
⟨*proof*⟩

lemma *ex-has-greatest-nat*:
$P\ k \Longrightarrow \forall y.\ P\ y \longrightarrow f\ y < b \Longrightarrow \exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow f\ y \leq f\ x)$
for f :: $'a \Rightarrow nat$
⟨*proof*⟩

lemma *arg-max-nat-lemma*:
$\llbracket\ P\ k;\ \ \forall y.\ P\ y \longrightarrow f\ y < b\ \rrbracket$
$\Longrightarrow P\ (\textit{arg-max}\ f\ P) \wedge (\forall y.\ P\ y \longrightarrow f\ y \leq f\ (\textit{arg-max}\ f\ P))$
for f :: $'a \Rightarrow nat$
⟨*proof*⟩

lemmas *arg-max-natI* = *arg-max-nat-lemma* [*THEN conjunct1*]

lemma *arg-max-nat-le*: $P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow f\ y < b \Longrightarrow f\ x \leq f\ (\textit{arg-max}\ f\ P)$
for f :: $'a \Rightarrow nat$
⟨*proof*⟩

end

61 Set intervals

theory *Set-Interval*
imports *Lattices-Big Divides 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 (*ASCII*)
-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 (*latex* **output**)
-UNION-le :: $'a \Rightarrow 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcup$(‹*unbreakable*›- ≤ -)/ -) [*0*, *0*, *10*] *10*)
-UNION-less :: $'a \Rightarrow 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcup$(‹*unbreakable*›- < -)/ -) [*0*, *0*, *10*] *10*)
-INTER-le :: $'a \Rightarrow 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcap$(‹*unbreakable*›- ≤ -)/ -) [*0*, *0*, *10*] *10*)
-INTER-less :: $'a \Rightarrow 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcap$(‹*unbreakable*›- < -)/ -) [*0*, *0*, *10*] *10*)

syntax
-UNION-le :: $'a => 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcup$ -≤-./ -) [*0*, *0*, *10*] *10*)
-UNION-less :: $'a => 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcup$ -<-./ -) [*0*, *0*, *10*] *10*)
-INTER-le :: $'a => 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcap$ -≤-./ -) [*0*, *0*, *10*] *10*)
-INTER-less :: $'a => 'a => 'b$ *set* => $'b$ *set* ((*3*$\bigcap$ -<-./ -) [*0*, *0*, *10*] *10*)

translations

$\bigcup i \leq n.\ A \rightleftharpoons \bigcup i \in \{..n\}.\ A$
$\bigcup i < n.\ A \rightleftharpoons \bigcup i \in \{..<n\}.\ A$
$\bigcap i \leq n.\ A \rightleftharpoons \bigcap i \in \{..n\}.\ A$
$\bigcap i < n.\ A \rightleftharpoons \bigcap i \in \{..<n\}.\ A$

61.1 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*⟩

lemma (**in** *linorder*) *lessThan-Int-lessThan*: $\{\ a <..\} \cap \{\ b <..\} = \{\ max\ a\ b <..\}$
⟨*proof*⟩

lemma (**in** *linorder*) *greaterThan-Int-greaterThan*: $\{..<\ a\} \cap \{..<\ b\} = \{..<\ min\ a\ b\}$
⟨*proof*⟩

61.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-empty-triv* [*simp*]: $\{\{\}..\} = UNIV$

⟨*proof*⟩

lemma *atMost-UNIV-triv* [*simp*]: $\{..UNIV\} = UNIV$
⟨*proof*⟩

lemma *atLeast-subset-iff* [*iff*]:
$(atLeast\ x \subseteq atLeast\ y) = (y \leq (x::'a::order))$
⟨*proof*⟩

lemma *atLeast-eq-iff* [*iff*]:
$(atLeast\ x = atLeast\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *greaterThan-subset-iff* [*iff*]:
$(greaterThan\ x \subseteq greaterThan\ y) = (y \leq (x::'a::linorder))$
⟨*proof*⟩

lemma *greaterThan-eq-iff* [*iff*]:
$(greaterThan\ x = greaterThan\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *atMost-subset-iff* [*iff*]: $(atMost\ x \subseteq atMost\ y) = (x \leq (y::'a::order))$
⟨*proof*⟩

lemma *atMost-eq-iff* [*iff*]: $(atMost\ x = atMost\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *lessThan-subset-iff* [*iff*]:
$(lessThan\ x \subseteq lessThan\ y) = (x \leq (y::'a::linorder))$
⟨*proof*⟩

lemma *lessThan-eq-iff* [*iff*]:
$(lessThan\ x = lessThan\ y) = (x = (y::'a::linorder))$
⟨*proof*⟩

lemma *lessThan-strict-subset-iff*:
fixes $m\ n :: 'a::linorder$
shows $\{..<m\} < \{..<n\} \longleftrightarrow m < n$
⟨*proof*⟩

lemma (**in** *linorder*) *Ici-subset-Ioi-iff*: $\{a\ ..\} \subseteq \{b\ <..\} \longleftrightarrow b < a$
⟨*proof*⟩

lemma (**in** *linorder*) *Iic-subset-Iio-iff*: $\{..\ a\} \subseteq \{..<\ b\} \longleftrightarrow a < b$
⟨*proof*⟩

lemma (**in** *preorder*) *Ioi-le-Ico*: $\{a\ <..\} \subseteq \{a\ ..\}$
⟨*proof*⟩

61.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [*simp*]:
$(i : \{l<..<u\}) = (l < i \ \& \ i < u)$
⟨*proof*⟩

lemma *atLeastLessThan-iff* [*simp*]:
$(i : \{l..<u\}) = (l <= i \ \& \ i < u)$
⟨*proof*⟩

lemma *greaterThanAtMost-iff* [*simp*]:
$(i : \{l<..u\}) = (l < i \ \& \ i <= u)$
⟨*proof*⟩

lemma *atLeastAtMost-iff* [*simp*]:
$(i : \{l..u\}) = (l <= i \ \& \ i <= u)$
⟨*proof*⟩

The above four lemmas could be declared as iffs. Unfortunately this breaks many proofs. Since it only helps blast, it is better to leave them alone.

lemma *greaterThanLessThan-eq*: $\{ a <..< b\} = \{ a <..\} \cap \{..< b \}$
⟨*proof*⟩

end

61.3.1 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 *Icc-eq-Icc*[*simp*]:
$\{l..h\} = \{l'..h'\} = (l=l' \wedge h=h' \vee \neg\ l \leq h \wedge \neg\ l' \leq h')$
⟨proof⟩

lemma *atLeastAtMost-singleton-iff* [*simp*]:
$\{a\ ..\ b\} = \{c\} \longleftrightarrow a = b \wedge b = c$
⟨proof⟩

lemma *Icc-subset-Ici-iff* [*simp*]:
$\{l..h\} \subseteq \{l'..\} = (\sim\ l \leq h \vee l \geq l')$
⟨proof⟩

lemma *Icc-subset-Iic-iff* [*simp*]:
$\{l..h\} \subseteq \{..h'\} = (\sim\ l \leq h \vee h \leq h')$
⟨proof⟩

lemma *not-Ici-eq-empty*[*simp*]: $\{l..\} \neq \{\}$
⟨*proof*⟩

lemma *not-Iic-eq-empty*[*simp*]: $\{..h\} \neq \{\}$
⟨*proof*⟩

lemmas *not-empty-eq-Ici-eq-empty*[*simp*] = *not-Ici-eq-empty*[*symmetric*]
lemmas *not-empty-eq-Iic-eq-empty*[*simp*] = *not-Iic-eq-empty*[*symmetric*]

end

context *no-top*
begin

lemma *not-UNIV-le-Icc*[*simp*]: $\neg$ *UNIV* $\subseteq \{l..h\}$
⟨*proof*⟩

lemma *not-UNIV-le-Iic*[*simp*]: $\neg$ *UNIV* $\subseteq \{..h\}$
⟨*proof*⟩

lemma *not-Ici-le-Icc*[*simp*]: $\neg \{l..\} \subseteq \{l'..h'\}$
⟨*proof*⟩

lemma *not-Ici-le-Iic*[*simp*]: $\neg \{l..\} \subseteq \{..h'\}$
⟨*proof*⟩

end

context *no-bot*
begin

lemma *not-UNIV-le-Ici*[*simp*]: $\neg$ *UNIV* $\subseteq \{l..\}$
⟨*proof*⟩

lemma *not-Iic-le-Icc*[*simp*]: $\neg \{..h\} \subseteq \{l'..h'\}$
⟨*proof*⟩

lemma *not-Iic-le-Ici*[*simp*]: $\neg \{..h\} \subseteq \{l'..\}$
⟨*proof*⟩

end

context *no-top*
begin

lemma *not-UNIV-eq-Icc*[*simp*]: $\neg$ *UNIV* $= \{l'..h'\}$

⟨*proof*⟩

lemmas *not-Icc-eq-UNIV*[*simp*] = *not-UNIV-eq-Icc*[*symmetric*]

lemma *not-UNIV-eq-Iic*[*simp*]: $\neg$ *UNIV* = $\{..h'\}$
⟨*proof*⟩

lemmas *not-Iic-eq-UNIV*[*simp*] = *not-UNIV-eq-Iic*[*symmetric*]

lemma *not-Icc-eq-Ici*[*simp*]: $\neg \{l..h\} = \{l'..\}$
⟨*proof*⟩

lemmas *not-Ici-eq-Icc*[*simp*] = *not-Icc-eq-Ici*[*symmetric*]

lemma *not-Iic-eq-Ici*[*simp*]: $\neg \{..h\} = \{l'..\}$
⟨*proof*⟩

lemmas *not-Ici-eq-Iic*[*simp*] = *not-Iic-eq-Ici*[*symmetric*]

end

context *no-bot*
begin

lemma *not-UNIV-eq-Ici*[*simp*]: $\neg$ *UNIV* = $\{l'..\}$
⟨*proof*⟩

lemmas *not-Ici-eq-UNIV*[*simp*] = *not-UNIV-eq-Ici*[*symmetric*]

lemma *not-Icc-eq-Iic*[*simp*]: $\neg \{l..h\} = \{..h'\}$
⟨*proof*⟩

lemmas *not-Iic-eq-Icc*[*simp*] = *not-Icc-eq-Iic*[*symmetric*]

end

context *dense-linorder*
begin

lemma *greaterThanLessThan-empty-iff*[*simp*]:
  $\{ a <..< b \} = \{\} \longleftrightarrow b \leq a$
  ⟨*proof*⟩

lemma *greaterThanLessThan-empty-iff2*[*simp*]:
  $\{\} = \{ a <..< b \} \longleftrightarrow b \leq a$
  ⟨*proof*⟩

lemma *atLeastLessThan-subseteq-atLeastAtMost-iff*:
$\{a ..< b\} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
⟨*proof*⟩

lemma *greaterThanAtMost-subseteq-atLeastAtMost-iff*:
$\{a <.. b\} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
⟨*proof*⟩

lemma *greaterThanLessThan-subseteq-atLeastAtMost-iff*:
$\{a <..< b\} \subseteq \{ c .. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
⟨*proof*⟩

lemma *atLeastAtMost-subseteq-atLeastLessThan-iff*:
$\{a .. b\} \subseteq \{ c ..< d \} \longleftrightarrow (a \leq b \longrightarrow c \leq a \wedge b < d)$
⟨*proof*⟩

lemma *greaterThanLessThan-subseteq-greaterThanLessThan*:
$\{a <..< b\} \subseteq \{c <..< d\} \longleftrightarrow (a < b \longrightarrow a \geq c \wedge b \leq d)$
⟨*proof*⟩

lemma *greaterThanAtMost-subseteq-atLeastLessThan-iff*:
$\{a <.. b\} \subseteq \{ c ..< d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b < d)$
⟨*proof*⟩

lemma *greaterThanLessThan-subseteq-atLeastLessThan-iff*:
$\{a <..< b\} \subseteq \{ c ..< d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
⟨*proof*⟩

lemma *greaterThanLessThan-subseteq-greaterThanAtMost-iff*:
$\{a <..< b\} \subseteq \{ c <.. d \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
⟨*proof*⟩

end

context *no-top*
begin

lemma *greaterThan-non-empty*[*simp*]: $\{x <..\} \neq \{\}$
⟨*proof*⟩

end

context *no-bot*
begin

lemma *lessThan-non-empty*[*simp*]: $\{..< x\} \neq \{\}$
⟨*proof*⟩

end

lemma (**in** *linorder*) *atLeastLessThan-subset-iff*:
$\{a..<b\} <= \{c..<d\} \Longrightarrow b <= a \mid c<=a \ \& \ b<=d$
⟨*proof*⟩

lemma *atLeastLessThan-inj*:
fixes $a\ b\ c\ d$ :: $'a$::*linorder*
assumes *eq*: $\{a\ ..<\ b\} = \{c\ ..<\ d\}$ **and** $a < b$ $c < d$
shows $a = c$ $b = d$
⟨*proof*⟩

lemma *atLeastLessThan-eq-iff*:
fixes $a\ b\ c\ d$ :: $'a$::*linorder*
assumes $a < b$ $c < d$
shows $\{a\ ..<\ b\} = \{c\ ..<\ d\} \longleftrightarrow a = c \wedge b = d$
⟨*proof*⟩

lemma (**in** *linorder*) *Ioc-inj*: $\{a <..\ b\} = \{c <..\ d\} \longleftrightarrow (b \leq a \wedge d \leq c) \vee a = c \wedge b = d$
⟨*proof*⟩

lemma (**in** *order*) *Iio-Int-singleton*: $\{..<k\} \cap \{x\} = (\mathit{if}\ x < k\ \mathit{then}\ \{x\}\ \mathit{else}\ \{\})$
⟨*proof*⟩

lemma (**in** *linorder*) *Ioc-subset-iff*: $\{a<..b\} \subseteq \{c<..d\} \longleftrightarrow (b \leq a \vee c \leq a \wedge b \leq d)$
⟨*proof*⟩

lemma (**in** *order-bot*) *atLeast-eq-UNIV-iff*: $\{x..\} = \mathit{UNIV} \longleftrightarrow x = \mathit{bot}$
⟨*proof*⟩

lemma (**in** *order-top*) *atMost-eq-UNIV-iff*: $\{..x\} = \mathit{UNIV} \longleftrightarrow x = \mathit{top}$
⟨*proof*⟩

lemma (**in** *bounded-lattice*) *atLeastAtMost-eq-UNIV-iff*:
$\{x..y\} = \mathit{UNIV} \longleftrightarrow (x = \mathit{bot} \wedge y = \mathit{top})$
⟨*proof*⟩

lemma *Iio-eq-empty-iff*: $\{..<\ n::'a::\{\mathit{linorder},\ \mathit{order\text{-}bot}\}\} = \{\} \longleftrightarrow n = \mathit{bot}$
⟨*proof*⟩

lemma *Iio-eq-empty-iff-nat*: $\{..<\ n::\mathit{nat}\} = \{\} \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *mono-image-least*:
assumes *f-mono*: *mono f* **and** *f-img*: f ‘ $\{m\ ..<\ n\} = \{m'\ ..<\ n'\}$ $m < n$
shows $f\ m = m'$
⟨*proof*⟩

61.4 Infinite intervals

context *dense-linorder*
begin

lemma *infinite-Ioo*:
  assumes $a < b$
  shows $\neg$ *finite* $\{a{<}..{<}b\}$
$\langle proof \rangle$

lemma *infinite-Icc*: $a < b \Longrightarrow \neg$ *finite* $\{a .. b\}$
  $\langle proof \rangle$

lemma *infinite-Ico*: $a < b \Longrightarrow \neg$ *finite* $\{a ..< b\}$
  $\langle proof \rangle$

lemma *infinite-Ioc*: $a < b \Longrightarrow \neg$ *finite* $\{a <.. b\}$
  $\langle proof \rangle$

lemma *infinite-Ioo-iff* [*simp*]: *infinite* $\{a{<}..{<}b\} \longleftrightarrow a < b$
  $\langle proof \rangle$

lemma *infinite-Icc-iff* [*simp*]: *infinite* $\{a .. b\} \longleftrightarrow a < b$
  $\langle proof \rangle$

lemma *infinite-Ico-iff* [*simp*]: *infinite* $\{a..{<}b\} \longleftrightarrow a < b$
  $\langle proof \rangle$

lemma *infinite-Ioc-iff* [*simp*]: *infinite* $\{a{<}..b\} \longleftrightarrow a < b$
  $\langle proof \rangle$

end

lemma *infinite-Iio*: $\neg$ *finite* $\{..< a :: 'a :: \{no\text{-}bot, linorder\}\}$
$\langle proof \rangle$

lemma *infinite-Iic*: $\neg$ *finite* $\{.. a :: 'a :: \{no\text{-}bot, linorder\}\}$
  $\langle proof \rangle$

lemma *infinite-Ioi*: $\neg$ *finite* $\{a :: 'a :: \{no\text{-}top, linorder\} <..\}$
$\langle proof \rangle$

lemma *infinite-Ici*: $\neg$ *finite* $\{a :: 'a :: \{no\text{-}top, linorder\} ..\}$
  $\langle proof \rangle$

61.4.1 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*⟩

lemma *Int-atMost*[*simp*]: $\{..a\} \cap \{..b\} = \{..\ min\ a\ b\}$
⟨*proof*⟩

lemma *Ioc-disjoint*: $\{a<..b\} \cap \{c<..d\} = \{\} \longleftrightarrow b \leq a \vee d \leq c \vee b \leq c \vee d \leq a$
⟨*proof*⟩

end

context *complete-lattice*
begin

lemma
shows *Sup-atLeast*[*simp*]: *Sup* $\{x\ ..\} = top$
and *Sup-greaterThanAtLeast*[*simp*]: $x < top \Longrightarrow$ *Sup* $\{x\ <..\} = top$
and *Sup-atMost*[*simp*]: *Sup* $\{..\ y\} = y$
and *Sup-atLeastAtMost*[*simp*]: $x \leq y \Longrightarrow$ *Sup* $\{\ x\ ..\ y\} = y$
and *Sup-greaterThanAtMost*[*simp*]: $x < y \Longrightarrow$ *Sup* $\{\ x\ <..\ y\} = y$
⟨*proof*⟩

lemma
shows *Inf-atMost*[*simp*]: *Inf* $\{..\ x\} = bot$
and *Inf-atMostLessThan*[*simp*]: $top < x \Longrightarrow$ *Inf* $\{..<\ x\} = bot$

and *Inf-atLeast*[*simp*]: $Inf\ \{x\ ..\} = x$
and *Inf-atLeastAtMost*[*simp*]: $x \leq y \Longrightarrow Inf\ \{\ x\ ..\ y\} = x$
and *Inf-atLeastLessThan*[*simp*]: $x < y \Longrightarrow Inf\ \{\ x\ ..< y\} = x$
⟨*proof*⟩

end

lemma
fixes $x\ y :: 'a :: \{complete\text{-}lattice,\ dense\text{-}linorder\}$
shows *Sup-lessThan*[*simp*]: $Sup\ \{..< y\} = y$
and *Sup-atLeastLessThan*[*simp*]: $x < y \Longrightarrow Sup\ \{\ x\ ..< y\} = y$
and *Sup-greaterThanLessThan*[*simp*]: $x < y \Longrightarrow Sup\ \{\ x <..< y\} = y$
and *Inf-greaterThan*[*simp*]: $Inf\ \{x <..\} = x$
and *Inf-greaterThanAtMost*[*simp*]: $x < y \Longrightarrow Inf\ \{\ x <..\ y\} = x$
and *Inf-greaterThanLessThan*[*simp*]: $x < y \Longrightarrow Inf\ \{\ x <..< y\} = x$
⟨*proof*⟩

61.5 Intervals of natural numbers

61.5.1 The Constant *lessThan*

lemma *lessThan-0* [*simp*]: $lessThan\ (0::nat) = \{\}$
⟨*proof*⟩

lemma *lessThan-Suc*: $lessThan\ (Suc\ k) = insert\ k\ (lessThan\ k)$
⟨*proof*⟩

The following proof is convenient in induction proofs where new elements get indices at the beginning. So it is used to transform $\{..<Suc\ n\}$ to 0 and $\{..<n\}$.

lemma *zero-notin-Suc-image*: $0 \notin Suc\ `\ A$
⟨*proof*⟩

lemma *lessThan-Suc-eq-insert-0*: $\{..<Suc\ n\} = insert\ 0\ (Suc\ `\ \{..<n\})$
⟨*proof*⟩

lemma *lessThan-Suc-atMost*: $lessThan\ (Suc\ k) = atMost\ k$
⟨*proof*⟩

lemma *Iic-Suc-eq-insert-0*: $\{..\ Suc\ n\} = insert\ 0\ (Suc\ `\ \{..\ n\})$
⟨*proof*⟩

lemma *UN-lessThan-UNIV*: $(UN\ m::nat.\ lessThan\ m) = UNIV$
⟨*proof*⟩

61.5.2 The Constant *greaterThan*

lemma *greaterThan-0*: $greaterThan\ 0 = range\ Suc$
⟨*proof*⟩

lemma *greaterThan-Suc*: $greaterThan\ (Suc\ k) = greaterThan\ k - \{Suc\ k\}$
⟨*proof*⟩

lemma *INT-greaterThan-UNIV*: $(INT\ m::nat.\ greaterThan\ m) = \{\}$
⟨*proof*⟩

61.5.3 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*⟩

61.5.4 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*⟩

61.5.5 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* [*code-abbrev*]: $\{0::nat..<n\} = \{..<n\}$
⟨*proof*⟩

lemma *atLeast0AtMost* [*code-abbrev*]: $\{0..n::nat\} = \{..n\}$
⟨*proof*⟩

lemma *lessThan-atLeast0*:
$\{..<n\} = \{0::nat..<n\}$
⟨*proof*⟩

lemma *atMost-atLeast0*:

$\{..n\} = \{0::nat..n\}$
⟨*proof*⟩

lemma *atLeastLessThan0*: $\{m..<0::nat\} = \{\}$
⟨*proof*⟩

lemma *atLeast0-lessThan-Suc*:
$\{0..<Suc\ n\} = insert\ n\ \{0..<n\}$
⟨*proof*⟩

lemma *atLeast0-lessThan-Suc-eq-insert-0*:
$\{0..<Suc\ n\} = insert\ 0\ (Suc\ `\ \{0..<n\})$
⟨*proof*⟩

61.5.6 The Constant *atLeastAtMost*

lemma *atLeast0-atMost-Suc*:
$\{0..Suc\ n\} = insert\ (Suc\ n)\ \{0..n\}$
⟨*proof*⟩

lemma *atLeast0-atMost-Suc-eq-insert-0*:
$\{0..Suc\ n\} = insert\ 0\ (Suc\ `\ \{0..n\})$
⟨*proof*⟩

61.5.7 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 *atLeastAtMost-insertL*: $m \le n \Longrightarrow insert\ m\ \{Suc\ m..n\} = \{m\ ..n\}$
⟨*proof*⟩

The analogous result is useful on *int*:

lemma *atLeastAtMostPlus1-int-conv*:
$m <= 1+n \Longrightarrow \{m..1+n\} = insert\ (1+n)\ \{m..n::int\}$
⟨*proof*⟩

lemma *atLeastLessThan-add-Un*: $i \leq j \Longrightarrow \{i..<j+k\} = \{i..<j\} \cup \{j..<j+k::nat\}$
⟨*proof*⟩

61.5.8 Intervals and numerals

lemma *lessThan-nat-numeral*: — Evaluation for specific numerals
$lessThan\ (numeral\ k :: nat) = insert\ (pred\text{-}numeral\ k)\ (lessThan\ (pred\text{-}numeral\ k))$
⟨*proof*⟩

lemma *atMost-nat-numeral*: — Evaluation for specific numerals
$atMost\ (numeral\ k :: nat) = insert\ (numeral\ k)\ (atMost\ (pred\text{-}numeral\ k))$
⟨*proof*⟩

lemma *atLeastLessThan-nat-numeral*: — Evaluation for specific numerals
$atLeastLessThan\ m\ (numeral\ k :: nat) =$
$(if\ m \leq (pred\text{-}numeral\ k)\ then\ insert\ (pred\text{-}numeral\ k)\ (atLeastLessThan\ m\ (pred\text{-}numeral\ k))$
$else\ \{\})$
⟨*proof*⟩

61.5.9 Image

lemma *image-add-atLeastAtMost* [*simp*]:
fixes $k :: 'a::linordered\text{-}semidom$
shows $(\lambda n.\ n+k)$ ‘ $\{i..j\} = \{i+k..j+k\}$ (**is** $?A = ?B$)
⟨*proof*⟩

lemma *image-diff-atLeastAtMost* [*simp*]:
fixes $d::'a::linordered\text{-}idom$ **shows** $(op - d$ ‘ $\{a..b\}) = \{d-b..d-a\}$
⟨*proof*⟩

lemma *image-mult-atLeastAtMost* [*simp*]:
fixes $d::'a::linordered\text{-}field$
assumes $d>0$ **shows** $(op * d$ ‘ $\{a..b\}) = \{d*a..d*b\}$
⟨*proof*⟩

lemma *image-affinity-atLeastAtMost*:
fixes $c :: 'a::linordered\text{-}field$
shows $((\lambda x.\ m*x + c)$ ‘ $\{a..b\}) = (if\ \{a..b\}=\{\}\ then\ \{\}$
$else\ if\ 0 \leq m\ then\ \{m*a + c\ ..\ m *b + c\}$
$else\ \{m*b + c\ ..\ m*a + c\})$
⟨*proof*⟩

lemma *image-affinity-atLeastAtMost-diff*:
  fixes c :: $'a$::*linordered-field*
  shows $((\lambda x.\ m*x - c)$ ‘ $\{a..b\}) = (if\ \{a..b\}=\{\}\ then\ \{\}$
    $else\ if\ 0 \leq m\ then\ \{m*a - c\ ..\ m*b - c\}$
    $else\ \{m*b - c\ ..\ m*a - c\})$
  ⟨*proof*⟩

lemma *image-affinity-atLeastAtMost-div*:
  fixes c :: $'a$::*linordered-field*
  shows $((\lambda x.\ x/m + c)$ ‘ $\{a..b\}) = (if\ \{a..b\}=\{\}\ then\ \{\}$
    $else\ if\ 0 \leq m\ then\ \{a/m + c\ ..\ b/m + c\}$
    $else\ \{b/m + c\ ..\ a/m + c\})$
  ⟨*proof*⟩

lemma *image-affinity-atLeastAtMost-div-diff*:
  fixes c :: $'a$::*linordered-field*
  shows $((\lambda x.\ x/m - c)$ ‘ $\{a..b\}) = (if\ \{a..b\}=\{\}\ then\ \{\}$
    $else\ if\ 0 \leq m\ then\ \{a/m - c\ ..\ b/m - c\}$
    $else\ \{b/m - c\ ..\ a/m - c\})$
  ⟨*proof*⟩

lemma *image-add-atLeastLessThan*:
  $(\%n::nat.\ n+k)$ ‘ $\{i..<j\} = \{i+k..<j+k\}$ (**is** $?A = ?B$)
⟨*proof*⟩

corollary *image-Suc-lessThan*:
  Suc ‘ $\{..<n\} = \{1..n\}$
  ⟨*proof*⟩

corollary *image-Suc-atMost*:
  Suc ‘ $\{..n\} = \{1..Suc\ n\}$
  ⟨*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 *atLeast1-lessThan-eq-remove0*:
  $\{Suc\ 0..<n\} = \{..<n\} - \{0\}$
  ⟨*proof*⟩

lemma *atLeast1-atMost-eq-remove0*:
  $\{Suc\ 0..n\} = \{..n\} - \{0\}$
  ⟨*proof*⟩

lemma *image-add-int-atLeastLessThan*:
  $(\%x.\ x + (l{::}int))$ ‘ $\{0..{<}u{-}l\} = \{l..{<}u\}$
  ⟨*proof*⟩

lemma *image-minus-const-atLeastLessThan-nat*:
  fixes $c :: nat$
  shows $(\lambda i.\ i - c)$ ‘ $\{x\ ..{<}\ y\}$ =
    $(\mathit{if}\ c < y\ \mathit{then}\ \{x - c\ ..{<}\ y - c\}\ \mathit{else\ if}\ x < y\ \mathit{then}\ \{0\}\ \mathit{else}\ \{\})$
  (**is** - = *?right*)
⟨*proof*⟩

lemma *image-int-atLeastLessThan*: *int* ‘ $\{a..{<}b\} = \{int\ a..{<}int\ b\}$
  ⟨*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

61.5.10 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*⟩

lemma *bounded-Max-nat*:
fixes P :: $nat \Rightarrow bool$
assumes x: $P\ x$ **and** M: $\bigwedge x.\ P\ x \Longrightarrow x \le M$
obtains m **where** $P\ m$ $\bigwedge x.\ P\ x \Longrightarrow x \le m$
⟨*proof*⟩

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
assumes $A \subseteq \{k..<k + card\ A\}$
shows $A = \{k..<k + card\ A\}$
⟨*proof*⟩

61.5.11 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*:
$(\bigcup i \le n{::}nat.\ M(i+k)) = (\bigcup i \in \{k..n+k\}.\ M\ i)$ (**is** $?A = ?B$)
$\langle proof \rangle$

lemma *UN-UN-finite-eq*: $(\bigcup n{::}nat.\ \bigcup i \in \{0..<n\}.\ A\ i) = (\bigcup n.\ A\ n)$
$\langle proof \rangle$

lemma *UN-finite-subset*:
$(\bigwedge n{::}nat.\ (\bigcup i \in \{0..<n\}.\ A\ i) \subseteq C) \Longrightarrow (\bigcup n.\ A\ n) \subseteq C$
$\langle proof \rangle$

lemma *UN-finite2-subset*:
assumes $\bigwedge n{::}nat.\ (\bigcup i \in \{0..<n\}.\ A\ i) \subseteq (\bigcup i \in \{0..<n + k\}.\ B\ i)$
shows $(\bigcup n.\ A\ n) \subseteq (\bigcup n.\ B\ n)$
$\langle proof \rangle$

lemma *UN-finite2-eq*:
$(\bigwedge n{::}nat.\ (\bigcup i \in \{0..<n\}.\ A\ i) = (\bigcup i \in \{0..<n + k\}.\ B\ i)) \Longrightarrow$
$(\bigcup n.\ A\ n) = (\bigcup n.\ B\ n)$
$\langle proof \rangle$

61.5.12 Cardinality

lemma *card-lessThan* [*simp*]: $card\ \{..<u\} = u$
$\langle proof \rangle$

lemma *card-atMost* [*simp*]: $card\ \{..u\} = Suc\ u$
$\langle proof \rangle$

lemma *card-atLeastLessThan* [*simp*]: $card\ \{l..<u\} = u - l$
$\langle proof \rangle$

lemma *card-atLeastAtMost* [*simp*]: $card\ \{l..u\} = Suc\ u - l$
$\langle proof \rangle$

lemma *card-greaterThanAtMost* [*simp*]: $card\ \{l<..u\} = u - l$
$\langle proof \rangle$

lemma *card-greaterThanLessThan* [*simp*]: $card\ \{l<..<u\} = u - Suc\ l$
$\langle proof \rangle$

lemma *subset-eq-atLeast0-lessThan-finite*:
fixes $n :: nat$
assumes $N \subseteq \{0..<n\}$
shows *finite N*
$\langle proof \rangle$

lemma *subset-eq-atLeast0-atMost-finite*:

fixes $n :: nat$
assumes $N \subseteq \{0..n\}$
shows *finite N*
⟨*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*:
$finite\ A \Longrightarrow finite\ B \Longrightarrow card\ A = card\ B \Longrightarrow EX\ h.\ bij\text{-}betw\ h\ A\ B$
⟨*proof*⟩

lemma *ex-bij-betw-nat-finite-1*:
$finite\ M \Longrightarrow \exists h.\ bij\text{-}betw\ h\ \{1\ ..\ card\ M\}\ M$
⟨*proof*⟩

lemma *bij-betw-iff-card*:
assumes *finite A finite B*
shows $(\exists f.\ bij\text{-}betw\ f\ A\ B) \longleftrightarrow (card\ A = card\ B)$
⟨*proof*⟩

lemma *inj-on-iff-card-le*:
assumes *FIN*: *finite A* **and** *FIN'*: *finite B*
shows $(\exists f.\ inj\text{-}on\ f\ A \wedge f\ `\ A \leq B) = (card\ A \leq card\ B)$
⟨*proof*⟩

lemma *subset-eq-atLeast0-lessThan-card*:
fixes $n :: nat$
assumes $N \subseteq \{0..<n\}$
shows $card\ N \leq n$
⟨*proof*⟩

61.6 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*⟩

61.6.1 Finiteness

lemma *image-atLeastZeroLessThan-int*: $0 \leq u ==>$
$\{(0::int)..<u\} = int\ \text{`}\ \{..<nat\ u\}$
⟨*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*⟩

61.6.2 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* $\{k.\ P\ k \wedge k < (i::nat)\}$
⟨*proof*⟩

lemma *card-less*:
assumes *zero-in-M*: $0 \in M$
shows *card* $\{k \in M.\ k < Suc\ i\} \neq 0$
⟨*proof*⟩

lemma *card-less-Suc2*: $0 \notin M \Longrightarrow card\ \{k.\ Suc\ k \in M \wedge k < i\} = card\ \{k \in M.\ k < Suc\ i\}$
⟨*proof*⟩

lemma *card-less-Suc*:
  assumes *zero-in-M*: $0 \in M$
    shows $Suc\ (card\ \{k.\ Suc\ k \in M \wedge k < i\}) = card\ \{k \in M.\ k < Suc\ i\}$
$\langle proof \rangle$

61.7 Lemmas useful with the summation operator sum

For examples, see Algebra/poly/UnivPoly2.thy

61.7.1 Disjoint Unions

Singletons and open intervals

lemma *ivl-disj-un-singleton*:
  $\{l::'a::linorder\}\ Un\ \{l<..\} = \{l..\}$
  $\{..<u\}\ Un\ \{u::'a::linorder\} = \{..u\}$
  $(l::'a::linorder) < u ==> \{l\}\ Un\ \{l<..<u\} = \{l..<u\}$
  $(l::'a::linorder) < u ==> \{l<..<u\}\ Un\ \{u\} = \{l<..u\}$
  $(l::'a::linorder) <= u ==> \{l\}\ Un\ \{l<..u\} = \{l..u\}$
  $(l::'a::linorder) <= u ==> \{l..<u\}\ Un\ \{u\} = \{l..u\}$
$\langle proof \rangle$

One- and two-sided intervals

lemma *ivl-disj-un-one*:
  $(l::'a::linorder) < u ==> \{..l\}\ Un\ \{l<..<u\} = \{..<u\}$
  $(l::'a::linorder) <= u ==> \{..<l\}\ Un\ \{l..<u\} = \{..<u\}$
  $(l::'a::linorder) <= u ==> \{..l\}\ Un\ \{l<..u\} = \{..u\}$
  $(l::'a::linorder) <= u ==> \{..<l\}\ Un\ \{l..u\} = \{..u\}$
  $(l::'a::linorder) <= u ==> \{l<..u\}\ Un\ \{u<..\} = \{l<..\}$
  $(l::'a::linorder) < u ==> \{l<..<u\}\ Un\ \{u..\} = \{l<..\}$
  $(l::'a::linorder) <= u ==> \{l..u\}\ Un\ \{u<..\} = \{l..\}$
  $(l::'a::linorder) <= u ==> \{l..<u\}\ Un\ \{u..\} = \{l..\}$
$\langle proof \rangle$

Two- and two-sided intervals

lemma *ivl-disj-un-two*:
  $[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..<u\} = \{l<..<u\}$
  $[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l<..m\}\ Un\ \{m<..<u\} = \{l<..<u\}$
  $[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..<u\} = \{l..<u\}$
  $[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l..m\}\ Un\ \{m<..<u\} = \{l..<u\}$
  $[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..u\} = \{l<..u\}$
  $[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l<..m\}\ Un\ \{m<..u\} = \{l<..u\}$
  $[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..u\} = \{l..u\}$
  $[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..m\}\ Un\ \{m<..u\} = \{l..u\}$
$\langle proof \rangle$

lemma *ivl-disj-un-two-touch*:
  $[|\ (l::'a::linorder) < m;\ m < u\ |] ==> \{l<..m\}\ Un\ \{m..<u\} = \{l<..<u\}$
  $[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l..m\}\ Un\ \{m..<u\} = \{l..<u\}$

$[| (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..m\}\ Un\ \{m..u\} = \{l<..u\}$
$[| (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..m\}\ Un\ \{m..u\} = \{l..u\}$
⟨*proof*⟩

lemmas *ivl-disj-un* = *ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two ivl-disj-un-two-touch*

61.7.2 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*

61.7.3 Some Differences

lemma *ivl-diff* [*simp*]:
$i \le n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
⟨*proof*⟩

lemma (**in** *linorder*) *lessThan-minus-lessThan* [*simp*]:
$\{..< n\} - \{..< m\} = \{m\ ..< n\}$
⟨*proof*⟩

lemma (**in** *linorder*) *atLeastAtMost-diff-ends*:
$\{a..b\} - \{a,\ b\} = \{a<..<b\}$
⟨*proof*⟩

61.7.4 Some Subset Conditions

lemma *ivl-subset* [*simp*]:
$(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \mid m \leq i \ \& \ j \leq (n::'a::linorder))$
⟨*proof*⟩

61.8 Generic big monoid operation over intervals

lemma *inj-on-add-nat'* [*simp*]:
inj-on (*plus k*) *N* **for** *k* :: *nat*
⟨*proof*⟩

context *comm-monoid-set*
begin

lemma *atLeast-lessThan-shift-bounds*:
fixes *m n k* :: *nat*
shows $F\ g\ \{m + k..<n + k\} = F\ (g \circ plus\ k)\ \{m..<n\}$
⟨*proof*⟩

lemma *atLeast-atMost-shift-bounds*:
fixes *m n k* :: *nat*
shows $F\ g\ \{m + k..n + k\} = F\ (g \circ plus\ k)\ \{m..n\}$
⟨*proof*⟩

lemma *atLeast-Suc-lessThan-Suc-shift*:
$F\ g\ \{Suc\ m..<Suc\ n\} = F\ (g \circ Suc)\ \{m..<n\}$
⟨*proof*⟩

lemma *atLeast-Suc-atMost-Suc-shift*:
$F\ g\ \{Suc\ m..Suc\ n\} = F\ (g \circ Suc)\ \{m..n\}$
⟨*proof*⟩

lemma *atLeast0-lessThan-Suc*:
$F\ g\ \{0..<Suc\ n\} = F\ g\ \{0..<n\} * g\ n$
⟨*proof*⟩

lemma *atLeast0-atMost-Suc*:
$F\ g\ \{0..Suc\ n\} = F\ g\ \{0..n\} * g\ (Suc\ n)$
⟨*proof*⟩

lemma *atLeast0-lessThan-Suc-shift*:
$F\ g\ \{0..<Suc\ n\} = g\ 0 * F\ (g \circ Suc)\ \{0..<n\}$
⟨*proof*⟩

lemma *atLeast0-atMost-Suc-shift*:
$F\ g\ \{0..Suc\ n\} = g\ 0 * F\ (g \circ Suc)\ \{0..n\}$
⟨*proof*⟩

lemma *ivl-cong*:

$a = c \Longrightarrow b = d \Longrightarrow (\bigwedge x.\ c \le x \Longrightarrow x < d \Longrightarrow g\ x = h\ x)$
$\Longrightarrow F\ g\ \{a..<b\} = F\ h\ \{c..<d\}$
⟨*proof*⟩

lemma *atLeast-lessThan-shift-0*:
fixes $m\ n\ p :: nat$
shows $F\ g\ \{m..<n\} = F\ (g \circ plus\ m)\ \{0..<n - m\}$
⟨*proof*⟩

lemma *atLeast-atMost-shift-0*:
fixes $m\ n\ p :: nat$
assumes $m \le n$
shows $F\ g\ \{m..n\} = F\ (g \circ plus\ m)\ \{0..n - m\}$
⟨*proof*⟩

lemma *atLeast-lessThan-concat*:
fixes $m\ n\ p :: nat$
shows $m \le n \Longrightarrow n \le p \Longrightarrow F\ g\ \{m..<n\} * F\ g\ \{n..<p\} = F\ g\ \{m..<p\}$
⟨*proof*⟩

lemma *atLeast-lessThan-rev*:
$F\ g\ \{n..<m\} = F\ (\lambda i.\ g\ (m + n - Suc\ i))\ \{n..<m\}$
⟨*proof*⟩

lemma *atLeast-atMost-rev*:
fixes $n\ m :: nat$
shows $F\ g\ \{n..m\} = F\ (\lambda i.\ g\ (m + n - i))\ \{n..m\}$
⟨*proof*⟩

lemma *atLeast-lessThan-rev-at-least-Suc-atMost*:
$F\ g\ \{n..<m\} = F\ (\lambda i.\ g\ (m + n - i))\ \{Suc\ n..m\}$
⟨*proof*⟩

end

61.9 Summation indexed over intervals

syntax (*ASCII*)
-from-to-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM - = -..-./ -*) [0,0,0,10] 10)
-from-upto-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM - = -..<-./ -*) [0,0,0,10] 10)
-upt-sum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM -<-./ -*) [0,0,10] 10)
-upto-sum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*SUM -<=-./ -*) [0,0,10] 10)

syntax (*latex-sum* **output**)
-from-to-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
$((3\sum_{- = -}^{-}\ -)\ [0,0,0,10]\ 10)$
-from-upto-sum :: $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
$((3\sum_{- = -}^{<-}\ -)\ [0,0,0,10]\ 10)$
-upt-sum :: $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$

((3$\sum$- < - -) [0,0,10] 10)
-upto-sum :: idt ⇒ 'a ⇒ 'b ⇒ 'b
((3$\sum$- ≤ - -) [0,0,10] 10)

syntax
-from-to-sum :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((3$\sum$- = -..-./ -) [0,0,0,10] 10)
-from-upto-sum :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((3$\sum$- = -..<-./ -) [0,0,0,10] 10)
-upt-sum :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((3$\sum$-<-./ -) [0,0,10] 10)
-upto-sum :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((3$\sum$-≤-./ -) [0,0,10] 10)

translations
$\sum x{=}a..b.\ t$ == *CONST sum* $(\lambda x.\ t)\ \{a..b\}$
$\sum x{=}a..{<}b.\ t$ == *CONST sum* $(\lambda x.\ t)\ \{a..{<}b\}$
$\sum i{\le}n.\ t$ == *CONST sum* $(\lambda i.\ t)\ \{..n\}$
$\sum i{<}n.\ t$ == *CONST sum* $(\lambda i.\ t)\ \{..{<}n\}$

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	LaTeX
$\sum x{\in}\{a..b\}.\ e$	$\sum x = a..b.\ e$	$\sum_{x = a}^{b} e$
$\sum x{\in}\{a..{<}b\}.\ e$	$\sum x = a..{<}b.\ e$	$\sum_{x = a}^{<b} e$
$\sum x{\in}\{..b\}.\ e$	$\sum x{\le}b.\ e$	$\sum_{x \le b} e$
$\sum x{\in}\{..{<}b\}.\ e$	$\sum x{<}b.\ e$	$\sum_{x < b} e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode* = *latex-sum* in antiquotations). It is not the default LaTeX output because it only works well with italic-style formulae, not tt-style.
Note that for uniformity on *nat* it is better to use $\sum x = 0..{<}n.\ e$ rather than $\sum x{<}n.\ e$: *sum* 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 *sum.cong* does not work well with the simplifier who adds the unsimplified premise $x \in B$ to the context.

lemmas *sum-ivl-cong* = *sum.ivl-cong*

lemma *sum-atMost-Suc* [*simp*]:
$(\sum i \le Suc\ n.\ f\ i) = (\sum i \le n.\ f\ i) + f\ (Suc\ n)$
⟨*proof*⟩

lemma *sum-lessThan-Suc* [*simp*]:

$(\sum i < Suc\ n.\ f\ i) = (\sum i < n.\ f\ i) + f\ n$
⟨*proof*⟩

lemma *sum-cl-ivl-Suc* [*simp*]:
$sum\ f\ \{m..Suc\ n\} = (if\ Suc\ n < m\ then\ 0\ else\ sum\ f\ \{m..n\} + f(Suc\ n))$
⟨*proof*⟩

lemma *sum-op-ivl-Suc* [*simp*]:
$sum\ f\ \{m..{<}Suc\ n\} = (if\ n < m\ then\ 0\ else\ sum\ f\ \{m..{<}n\} + f(n))$
⟨*proof*⟩

lemma *sum-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 *sum-head-Suc*:
$m \le n \Longrightarrow sum\ f\ \{m..n\} = f\ m + sum\ f\ \{Suc\ m..n\}$
⟨*proof*⟩

lemma *sum-head-upt-Suc*:
$m < n \Longrightarrow sum\ f\ \{m..{<}n\} = f\ m + sum\ f\ \{Suc\ m..{<}n\}$
⟨*proof*⟩

lemma *sum-ub-add-nat*: **assumes** $(m::nat) \le n + 1$
shows $sum\ f\ \{m..n + p\} = sum\ f\ \{m..n\} + sum\ f\ \{n + 1..n + p\}$
⟨*proof*⟩

lemmas *sum-add-nat-ivl* = *sum.atLeast-lessThan-concat*

lemma *sum-diff-nat-ivl*:
fixes $f :: nat \Rightarrow 'a::ab\text{-}group\text{-}add$
shows $[\![\ m \le n;\ n \le p\ ]\!] \Longrightarrow$
$sum\ f\ \{m..{<}p\} - sum\ f\ \{m..{<}n\} = sum\ f\ \{n..{<}p\}$
⟨*proof*⟩

lemma *sum-natinterval-difff*:
fixes $f :: nat \Rightarrow ('a::ab\text{-}group\text{-}add)$
shows $sum\ (\lambda k.\ f\ k - f(k + 1))\ \{(m::nat)\ ..\ n\} =$
$(if\ m <= n\ then\ f\ m - f(n + 1)\ else\ 0)$
⟨*proof*⟩

lemma *sum-nat-group*: $(\sum m{<}n::nat.\ sum\ f\ \{m * k\ ..{<}\ m*k + k\}) = sum\ f\ \{..{<}\ n * k\}$
⟨*proof*⟩

lemma *sum-triangle-reindex*:

fixes n :: *nat*
shows $(\sum (i,j)\in\{(i,j).\ i+j < n\}.\ f\ i\ j) = (\sum k<n.\ \sum i\leq k.\ f\ i\ (k - i))$
⟨*proof*⟩

lemma *sum-triangle-reindex-eq*:
fixes n :: *nat*
shows $(\sum (i,j)\in\{(i,j).\ i+j \leq n\}.\ f\ i\ j) = (\sum k\leq n.\ \sum i\leq k.\ f\ i\ (k - i))$
⟨*proof*⟩

lemma *nat-diff-sum-reindex*: $(\sum i<n.\ f\ (n - Suc\ i)) = (\sum i<n.\ f\ i)$
⟨*proof*⟩

61.9.1 Shifting bounds

lemma *sum-shift-bounds-nat-ivl*:
sum f $\{m+k..<n+k\}$ = *sum* (%i. $f(i + k)$)$\{m..<n::nat\}$
⟨*proof*⟩

lemma *sum-shift-bounds-cl-nat-ivl*:
sum f $\{m+k..n+k\}$ = *sum* (%i. $f(i + k)$)$\{m..n::nat\}$
⟨*proof*⟩

corollary *sum-shift-bounds-cl-Suc-ivl*:
sum f $\{Suc\ m..Suc\ n\}$ = *sum* (%i. $f(Suc\ i)$)$\{m..n\}$
⟨*proof*⟩

corollary *sum-shift-bounds-Suc-ivl*:
sum f $\{Suc\ m..<Suc\ n\}$ = *sum* (%i. $f(Suc\ i)$)$\{m..<n\}$
⟨*proof*⟩

lemma *sum-shift-lb-Suc0-0*:
$f(0::nat) = (0::nat) \Longrightarrow$ *sum f* $\{Suc\ 0..k\}$ = *sum f* $\{0..k\}$
⟨*proof*⟩

lemma *sum-shift-lb-Suc0-0-upt*:
$f(0::nat) = 0 \Longrightarrow$ *sum f* $\{Suc\ 0..<k\}$ = *sum f* $\{0..<k\}$
⟨*proof*⟩

lemma *sum-atMost-Suc-shift*:
fixes f :: *nat* ⇒ *'a::comm-monoid-add*
shows $(\sum i\leq Suc\ n.\ f\ i) = f\ 0 + (\sum i\leq n.\ f\ (Suc\ i))$
⟨*proof*⟩

lemma *sum-lessThan-Suc-shift*:
$(\sum i<Suc\ n.\ f\ i) = f\ 0 + (\sum i<n.\ f\ (Suc\ i))$
⟨*proof*⟩

lemma *sum-atMost-shift*:
fixes f :: *nat* ⇒ *'a::comm-monoid-add*

shows $(\sum i\le n.\ f\ i) = f\ 0 + (\sum i<n.\ f\ (Suc\ i))$
⟨*proof*⟩

lemma *sum-last-plus*: **fixes** $n::nat$ **shows** $m <= n \Longrightarrow (\sum i = m..n.\ f\ i) = f\ n + (\sum i = m..<n.\ f\ i)$
⟨*proof*⟩

lemma *sum-Suc-diff*:
fixes $f :: nat \Rightarrow 'a::ab\text{-}group\text{-}add$
assumes $m \le Suc\ n$
shows $(\sum i = m..n.\ f(Suc\ i) - f\ i) = f\ (Suc\ n) - f\ m$
⟨*proof*⟩

lemma *sum-Suc-diff'*:
fixes $f :: nat \Rightarrow 'a::ab\text{-}group\text{-}add$
assumes $m \le n$
shows $(\sum i = m..<n.\ f\ (Suc\ i) - f\ i) = f\ n - f\ m$
⟨*proof*⟩

lemma *nested-sum-swap*:
$(\sum i = 0..n.\ (\sum j = 0..<i.\ a\ i\ j)) = (\sum j = 0..<n.\ \sum i = Suc\ j..n.\ a\ i\ j)$
⟨*proof*⟩

lemma *nested-sum-swap'*:
$(\sum i\le n.\ (\sum j<i.\ a\ i\ j)) = (\sum j<n.\ \sum i = Suc\ j..n.\ a\ i\ j)$
⟨*proof*⟩

lemma *sum-atLeast1-atMost-eq*:
$sum\ f\ \{Suc\ 0..n\} = (\sum k<n.\ f\ (Suc\ k))$
⟨*proof*⟩

61.9.2 Telescoping

lemma *sum-telescope*:
fixes $f::nat \Rightarrow 'a::ab\text{-}group\text{-}add$
shows $sum\ (\lambda i.\ f\ i - f\ (Suc\ i))\ \{..\ i\} = f\ 0 - f\ (Suc\ i)$
⟨*proof*⟩

lemma *sum-telescope''*:
assumes $m \le n$
shows $(\sum k\in\{Suc\ m..n\}.\ f\ k - f\ (k - 1)) = f\ n - (f\ m :: 'a :: ab\text{-}group\text{-}add)$
⟨*proof*⟩

lemma *sum-lessThan-telescope*:
$(\sum n<m.\ f\ (Suc\ n) - f\ n :: 'a :: ab\text{-}group\text{-}add) = f\ m - f\ 0$
⟨*proof*⟩

lemma *sum-lessThan-telescope'*:
$(\sum n<m.\ f\ n - f\ (Suc\ n) :: 'a :: ab\text{-}group\text{-}add) = f\ 0 - f\ m$

⟨*proof*⟩

61.10 The formula for geometric sums

lemma *sum-power2*: $(\sum i{=}0..{<}k.\ (2{::}nat)\hat{\ }i) = 2\hat{\ }k{-}1$
⟨*proof*⟩

lemma *geometric-sum*:
assumes $x \neq 1$
shows $(\sum i{<}n.\ x\ \hat{\ }\ i) = (x\ \hat{\ }\ n - 1) / (x - 1{::}'a{::}field)$
⟨*proof*⟩

lemma *diff-power-eq-sum*:
fixes $y :: 'a{::}\{comm\text{-}ring,monoid\text{-}mult\}$
shows
$x\ \hat{\ }\ (Suc\ n) - y\ \hat{\ }\ (Suc\ n) =$
$(x - y) * (\sum p{<}Suc\ n.\ (x\ \hat{\ }\ p) * y\ \hat{\ }\ (n - p))$
⟨*proof*⟩

corollary *power-diff-sumr2*: — *COMPLEX-POLYFUN* in HOL Light
fixes $x :: 'a{::}\{comm\text{-}ring,monoid\text{-}mult\}$
shows $x\hat{\ }n - y\hat{\ }n = (x - y) * (\sum i{<}n.\ y\hat{\ }(n - Suc\ i) * x\hat{\ }i)$
⟨*proof*⟩

lemma *power-diff-1-eq*:
fixes $x :: 'a{::}\{comm\text{-}ring,monoid\text{-}mult\}$
shows $n \neq 0 \Longrightarrow x\hat{\ }n - 1 = (x - 1) * (\sum i{<}n.\ (x\hat{\ }i))$
⟨*proof*⟩

lemma *one-diff-power-eq'*:
fixes $x :: 'a{::}\{comm\text{-}ring,monoid\text{-}mult\}$
shows $n \neq 0 \Longrightarrow 1 - x\hat{\ }n = (1 - x) * (\sum i{<}n.\ x\hat{\ }(n - Suc\ i))$
⟨*proof*⟩

lemma *one-diff-power-eq*:
fixes $x :: 'a{::}\{comm\text{-}ring,monoid\text{-}mult\}$
shows $n \neq 0 \Longrightarrow 1 - x\hat{\ }n = (1 - x) * (\sum i{<}n.\ x\hat{\ }i)$
⟨*proof*⟩

lemma *sum-gp-basic*:
fixes $x :: 'a{::}\{comm\text{-}ring,monoid\text{-}mult\}$
shows $(1 - x) * (\sum i{\le}n.\ x\hat{\ }i) = 1 - x\hat{\ }Suc\ n$
⟨*proof*⟩

lemma *sum-power-shift*:
fixes $x :: 'a{::}\{comm\text{-}ring,monoid\text{-}mult\}$
assumes $m \le n$
shows $(\sum i{=}m..n.\ x\hat{\ }i) = x\hat{\ }m * (\sum i{\le}n{-}m.\ x\hat{\ }i)$
⟨*proof*⟩

lemma *sum-gp-multiplied*:
fixes $x :: 'a::\{comm\text{-}ring,monoid\text{-}mult\}$
assumes $m \leq n$
shows $(1 - x) * (\sum i{=}m..n.\ x\hat{\ }i) = x\hat{\ }m - x\hat{\ }Suc\ n$
⟨*proof*⟩

lemma *sum-gp*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{=}m..n.\ x\hat{\ }i) =$
$(if\ n < m\ then\ 0$
$else\ if\ x = 1\ then\ of\text{-}nat((n + 1) - m)$
$else\ (x\hat{\ }m - x\hat{\ }Suc\ n)\ /\ (1 - x))$
⟨*proof*⟩

61.11 Geometric progressions

lemma *sum-gp0*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{\leq}n.\ x\hat{\ }i) = (if\ x = 1\ then\ of\text{-}nat(n + 1)\ else\ (1 - x\hat{\ }Suc\ n)\ /\ (1 - x))$
⟨*proof*⟩

lemma *sum-power-add*:
fixes $x :: 'a::\{comm\text{-}ring,monoid\text{-}mult\}$
shows $(\sum i{\in}I.\ x\hat{\ }(m{+}i)) = x\hat{\ }m * (\sum i{\in}I.\ x\hat{\ }i)$
⟨*proof*⟩

lemma *sum-gp-offset*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{=}m..m{+}n.\ x\hat{\ }i) =$
$(if\ x = 1\ then\ of\text{-}nat\ n + 1\ else\ x\hat{\ }m * (1 - x\hat{\ }Suc\ n)\ /\ (1 - x))$
⟨*proof*⟩

lemma *sum-gp-strict*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{<}n.\ x\hat{\ }i) = (if\ x = 1\ then\ of\text{-}nat\ n\ else\ (1 - x\hat{\ }n)\ /\ (1 - x))$
⟨*proof*⟩

61.11.1 The formula for arithmetic sums

lemma *gauss-sum*:
$(2::'a::comm\text{-}semiring\text{-}1)*(\sum i{\in}\{1..n\}.\ of\text{-}nat\ i) = of\text{-}nat\ n*((of\text{-}nat\ n){+}1)$
⟨*proof*⟩

theorem *arith-series-general*:
$(2::'a::comm\text{-}semiring\text{-}1) * (\sum i{\in}\{..{<}n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1)*d))$
⟨*proof*⟩

lemma *arith-series-nat*:
$(2{::}nat) * (\sum i{\in}\{..{<}n\}.\ a{+}i{*}d) = n * (a + (a{+}(n - 1){*}d))$
⟨*proof*⟩

lemma *arith-series-int*:
$2 * (\sum i{\in}\{..{<}n\}.\ a + int\ i * d) = int\ n * (a + (a + int(n - 1){*}d))$
⟨*proof*⟩

lemma *sum-diff-distrib*: $\forall x.\ Q\ x \le P\ x \Longrightarrow (\sum x{<}n.\ P\ x) - (\sum x{<}n.\ Q\ x) = (\sum x{<}n.\ P\ x - Q\ x :: nat)$
⟨*proof*⟩

61.11.2 Division remainder

lemma *range-mod*:
fixes $n :: nat$
assumes $n > 0$
shows $range\ (\lambda m.\ m\ mod\ n) = \{0..{<}n\}$ (**is** $?A = ?B$)
⟨*proof*⟩

61.12 Products indexed over intervals

syntax (*ASCII*)
$\text{-}from\text{-}to\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*PROD* - = -..-./ -) [0,0,0,10] 10)
$\text{-}from\text{-}upto\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*PROD* - = -..<-./ -) [0,0,0,10] 10)
$\text{-}upt\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*PROD* -<-./ -) [0,0,10] 10)
$\text{-}upto\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ ((*PROD* -<=-./ -) [0,0,10] 10)

syntax (*latex-prod* **output**)
$\text{-}from\text{-}to\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
$((3\prod_{\text{-} = \text{-}}^{\text{-}}\ \text{-})\ [0,0,0,10]\ 10)$
$\text{-}from\text{-}upto\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
$((3\prod_{\text{-} = \text{-}}^{<\text{-}}\ \text{-})\ [0,0,0,10]\ 10)$
$\text{-}upt\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
$((3\prod_{\text{-} < \text{-}}\ \text{-})\ [0,0,10]\ 10)$
$\text{-}upto\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
$((3\prod_{\text{-} \le \text{-}}\ \text{-})\ [0,0,10]\ 10)$

syntax
$\text{-}from\text{-}to\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ $((3\prod$ - = -..-./ -) [0,0,0,10] 10)
$\text{-}from\text{-}upto\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ $((3\prod$ - = -..<-./ -) [0,0,0,10] 10)
$\text{-}upt\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ $((3\prod$ -<-./ -) [0,0,10] 10)
$\text{-}upto\text{-}prod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ $((3\prod$ -≤-./ -) [0,0,10] 10)

translations
$\prod x{=}a..b.\ t \rightleftharpoons CONST\ prod\ (\lambda x.\ t)\ \{a..b\}$
$\prod x{=}a..{<}b.\ t \rightleftharpoons CONST\ prod\ (\lambda x.\ t)\ \{a..{<}b\}$
$\prod i{\le}n.\ t \rightleftharpoons CONST\ prod\ (\lambda i.\ t)\ \{..n\}$
$\prod i{<}n.\ t \rightleftharpoons CONST\ prod\ (\lambda i.\ t)\ \{..{<}n\}$

lemma *prod-int-plus-eq*: $prod\ int\ \{i..i+j\} = \prod\{int\ i..int\ (i+j)\}$
⟨*proof*⟩

lemma *prod-int-eq*: $prod\ int\ \{i..j\} = \prod\{int\ i..int\ j\}$
⟨*proof*⟩

61.12.1 Shifting bounds

lemma *prod-shift-bounds-nat-ivl*:
$prod\ f\ \{m+k..<n+k\} = prod\ (\%i.\ f(i + k))\{m..<n::nat\}$
⟨*proof*⟩

lemma *prod-shift-bounds-cl-nat-ivl*:
$prod\ f\ \{m+k..n+k\} = prod\ (\%i.\ f(i + k))\{m..n::nat\}$
⟨*proof*⟩

corollary *prod-shift-bounds-cl-Suc-ivl*:
$prod\ f\ \{Suc\ m..Suc\ n\} = prod\ (\%i.\ f(Suc\ i))\{m..n\}$
⟨*proof*⟩

corollary *prod-shift-bounds-Suc-ivl*:
$prod\ f\ \{Suc\ m..<Suc\ n\} = prod\ (\%i.\ f(Suc\ i))\{m..<n\}$
⟨*proof*⟩

lemma *prod-lessThan-Suc*: $prod\ f\ \{..<Suc\ n\} = prod\ f\ \{..<n\} * f\ n$
⟨*proof*⟩

lemma *prod-lessThan-Suc-shift*:$(\prod i<Suc\ n.\ f\ i) = f\ 0 * (\prod i<n.\ f\ (Suc\ i))$
⟨*proof*⟩

lemma *prod-atLeastLessThan-Suc*: $a \le b \Longrightarrow prod\ f\ \{a..<Suc\ b\} = prod\ f\ \{a..<b\} * f\ b$
⟨*proof*⟩

lemma *prod-nat-ivl-Suc'*:
assumes $m \le Suc\ n$
shows $prod\ f\ \{m..Suc\ n\} = f\ (Suc\ n) * prod\ f\ \{m..n\}$
⟨*proof*⟩

61.13 Efficient folding over intervals

function *fold-atLeastAtMost-nat* **where**
[*simp del*]: $fold\text{-}atLeastAtMost\text{-}nat\ f\ a\ (b::nat)\ acc =$
$(if\ a > b\ then\ acc\ else\ fold\text{-}atLeastAtMost\text{-}nat\ f\ (a+1)\ b\ (f\ a\ acc))$
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *fold-atLeastAtMost-nat*:
assumes *comp-fun-commute f*

shows *fold-atLeastAtMost-nat f a b acc = Finite-Set.fold f acc {a..b}*
⟨*proof*⟩

lemma *sum-atLeastAtMost-code*:
sum f {a..b} = fold-atLeastAtMost-nat (λa acc. f a + acc) a b 0
⟨*proof*⟩

lemma *prod-atLeastAtMost-code*:
*prod f {a..b} = fold-atLeastAtMost-nat (λa acc. f a * acc) a b 1*
⟨*proof*⟩

61.14 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 ⟹ nat-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-nat m ⟹ is-nat n ⟹ {m..n} = int ‘ {nat m..nat n}
⟨*proof*⟩

lemma *transfer-int-nat-set-function-closures*:
is-nat x ⟹ nat-set {x..y}
⟨*proof*⟩

declare *transfer-morphism-int-nat*[*transfer add*
return: *transfer-int-nat-set-functions*
transfer-int-nat-set-function-closures
]

end

62 Decision Procedure for Presburger Arithmetic

theory *Presburger*
imports *Groebner-Basis Set-Interval*
keywords *try0* :: *diag*
begin

⟨*ML*⟩

62.1 The $-\infty$ and $+\infty$ Properties

lemma *minf*:

$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x; \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x; \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x < t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \leq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x > t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \geq t) = False$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})<z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})<z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x<z.\ F = F$
⟨*proof*⟩

lemma *pinf*:

$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x; \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x; \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x < t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \leq t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x > t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \geq t) = True$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})>z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})>z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x>z.\ F = F$
⟨*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*⟩

62.2 The A and B sets

lemma *bset*:
$[\![\forall x.(\forall j \in \{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow P x \longrightarrow P(x - D) ;$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow Q x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x - D) \wedge Q (x - D))$
$[\![\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow P x \longrightarrow P(x - D) ;$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow Q x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (P x \vee Q x) \longrightarrow (P(x - D) \vee Q (x - D))$
$[\![D>0; t - 1\in B]\!] \Longrightarrow (\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (x = t) \longrightarrow (x - D = t))$
$[\![D>0 ; t \in B]\!] \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
$D>0 \Longrightarrow (\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (x < t) \longrightarrow (x - D < t))$
$D>0 \Longrightarrow (\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (x \leq t) \longrightarrow (x - D \leq t))$
$[\![D>0 ; t \in B]\!] \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (x > t) \longrightarrow (x - D > t))$
$[\![D>0 ; t - 1 \in B]\!] \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (x \geq t) \longrightarrow (x - D \geq t))$
$d\ dvd\ D \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (d\ dvd\ x+t) \longrightarrow (d\ dvd\ (x - D) + t))$
$d\ dvd\ D \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j)\longrightarrow (\neg d\ dvd\ x+t) \longrightarrow (\neg\ d\ dvd\ (x - D) + t))$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow F \longrightarrow F$
$\langle proof \rangle$

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*⟩

62.3 Cooper's Theorem $-\infty$ and $+\infty$ Version

62.3.1 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*⟩

62.3.2 The $-\infty$ Version

lemma *decr-lemma*: $0 < (d::int) \Longrightarrow x - (|x - z| + 1) * d < z$
⟨*proof*⟩

lemma *incr-lemma*: $0 < (d::int) \Longrightarrow z < x + (|x - z| + 1) * d$
⟨*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** $?L = (?R1 \vee ?R2)$)
⟨*proof*⟩

62.3.3 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 $(\exists\ x.\ P'\ x) \longrightarrow (\exists\ x.\ P\ x)$
$\langle proof \rangle$

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 $ALL\ x.\ P\ x \longrightarrow P(x + k*d)$
$\langle proof \rangle$

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** $?L = (?R1 \vee ?R2)$)
$\langle proof \rangle$

lemma *simp-from-to*: $\{i..j::int\} = (if\ j < i\ then\ \{\}\ else\ insert\ i\ \{i+1..j\})$
$\langle proof \rangle$

theorem *unity-coeff-ex*: $(\exists(x::'a::\{semiring\text{-}0,Rings.dvd\}).\ P\ (l * x)) \equiv (\exists x.\ l\ dvd\ (x + 0) \wedge P\ x)$
  $\langle proof \rangle$

lemma *zdvd-mono*:
  fixes $k\ m\ t :: int$
  assumes $k \neq 0$
  shows $m\ dvd\ t \equiv k * m\ dvd\ k * t$
  $\langle proof \rangle$

lemma *uminus-dvd-conv*:
  fixes $d\ t :: int$
  shows $d\ dvd\ t \equiv -\ d\ dvd\ t$ **and** $d\ dvd\ t \equiv d\ dvd\ -\ t$
  $\langle proof \rangle$

Theorems for transforming predicates on nat to predicates on *int*

lemma *zdiff-int-split*: $P\ (int\ (x - y)) =$
  $((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
  $\langle proof \rangle$

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*:

$[\![x = x'; 0 \leq x' \Longrightarrow P = P']\!] \Longrightarrow (0 \leq (x::int) \longrightarrow P) = (0 \leq x' \longrightarrow P')$
⟨proof⟩

theorem *conj-le-cong*:
$[\![x = x'; 0 \leq x' \Longrightarrow P = P']\!] \Longrightarrow (0 \leq (x::int) \wedge P) = (0 \leq x' \wedge P')$
⟨proof⟩

⟨ML⟩

declare *mod-eq-0-iff-dvd* [*presburger*]
declare *mod-by-Suc-0* [*presburger*]
declare *mod-0* [*presburger*]
declare *mod-by-1* [*presburger*]
declare *mod-self* [*presburger*]
declare *div-by-0* [*presburger*]
declare *mod-by-0* [*presburger*]
declare *mod-div-trivial* [*presburger*]
declare *mult-div-mod-eq* [*presburger*]
declare *div-mult-mod-eq* [*presburger*]
declare *mod-mult-self1* [*presburger*]
declare *mod-mult-self2* [*presburger*]
declare *mod2-Suc-Suc* [*presburger*]
declare *not-mod-2-eq-0-eq-1* [*presburger*]
declare *nat-zero-less-power-iff* [*presburger*]

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⟩

context *semiring-parity*
begin

declare *even-times-iff* [*presburger*]

declare *even-power* [*presburger*]

lemma [*presburger*]:
$even\ (a + b) \longleftrightarrow even\ a \wedge even\ b \vee odd\ a \wedge odd\ b$
⟨proof⟩

end

context *ring-parity*
begin

declare *even-minus* [*presburger*]

end

context *linordered-idom*
begin

declare *zero-le-power-eq* [*presburger*]

declare *zero-less-power-eq* [*presburger*]

declare *power-less-zero-eq* [*presburger*]

declare *power-le-zero-eq* [*presburger*]

end

declare *even-Suc* [*presburger*]

lemma [*presburger*]:
$Suc\ n\ div\ Suc\ (Suc\ 0) = n\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ n$
⟨*proof*⟩

declare *even-diff-nat* [*presburger*]

lemma [*presburger*]:
fixes $k :: int$
shows $(k + 1)\ div\ 2 = k\ div\ 2 \longleftrightarrow even\ k$
⟨*proof*⟩

lemma [*presburger*]:
fixes $k :: int$
shows $(k + 1)\ div\ 2 = k\ div\ 2 + 1 \longleftrightarrow odd\ k$
⟨*proof*⟩

lemma [*presburger*]:
$even\ n \longleftrightarrow even\ (int\ n)$
⟨*proof*⟩

62.4 Nice facts about division by $4::'a$

lemma *even-even-mod-4-iff*:
$even\ (n::nat) \longleftrightarrow even\ (n\ mod\ 4)$
⟨*proof*⟩

lemma *odd-mod-4-div-2*:
$n\ mod\ 4 = (3::nat) \Longrightarrow odd\ ((n - 1)\ div\ 2)$
⟨*proof*⟩

lemma *even-mod-4-div-2*:
$n\ mod\ 4 = (1::nat) \Longrightarrow even\ ((n - 1)\ div\ 2)$
⟨*proof*⟩

62.5 Try0

⟨*ML*⟩

end

63 Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2

theory *SMT*
imports *Divides*
keywords *smt-status* :: *diag*
begin

63.1 A skolemization tactic and proof method

lemma *choices*:
$\bigwedge Q.\ \forall x.\ \exists y\ ya.\ Q\ x\ y\ ya \Longrightarrow \exists f\ fa.\ \forall x.\ Q\ x\ (f\ x)\ (fa\ x)$
$\bigwedge Q.\ \forall x.\ \exists y\ ya\ yb.\ Q\ x\ y\ ya\ yb \Longrightarrow \exists f\ fa\ fb.\ \forall x.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)$
$\bigwedge Q.\ \forall x.\ \exists y\ ya\ yb\ yc.\ Q\ x\ y\ ya\ yb\ yc \Longrightarrow \exists f\ fa\ fb\ fc.\ \forall x.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)$
$(fc\ x)$
$\bigwedge Q.\ \forall x.\ \exists y\ ya\ yb\ yc\ yd.\ Q\ x\ y\ ya\ yb\ yc\ yd \Longrightarrow$
$\exists f\ fa\ fb\ fc\ fd.\ \forall x.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)$
$\bigwedge Q.\ \forall x.\ \exists y\ ya\ yb\ yc\ yd\ ye.\ Q\ x\ y\ ya\ yb\ yc\ yd\ ye \Longrightarrow$
$\exists f\ fa\ fb\ fc\ fd\ fe.\ \forall x.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)\ (fe\ x)$
$\bigwedge Q.\ \forall x.\ \exists y\ ya\ yb\ yc\ yd\ ye\ yf.\ Q\ x\ y\ ya\ yb\ yc\ yd\ ye\ yf \Longrightarrow$
$\exists f\ fa\ fb\ fc\ fd\ fe\ ff.\ \forall x.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)\ (fe\ x)\ (ff\ x)$
$\bigwedge Q.\ \forall x.\ \exists y\ ya\ yb\ yc\ yd\ ye\ yf\ yg.\ Q\ x\ y\ ya\ yb\ yc\ yd\ ye\ yf\ yg \Longrightarrow$
$\exists f\ fa\ fb\ fc\ fd\ fe\ ff\ fg.\ \forall x.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)\ (fe\ x)\ (ff\ x)\ (fg$
$x)$
⟨*proof*⟩

lemma *bchoices*:
$\bigwedge Q.\ \forall x \in S.\ \exists y\ ya.\ Q\ x\ y\ ya \Longrightarrow \exists f\ fa.\ \forall x \in S.\ Q\ x\ (f\ x)\ (fa\ x)$
$\bigwedge Q.\ \forall x \in S.\ \exists y\ ya\ yb.\ Q\ x\ y\ ya\ yb \Longrightarrow \exists f\ fa\ fb.\ \forall x \in S.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)$
$\bigwedge Q.\ \forall x \in S.\ \exists y\ ya\ yb\ yc.\ Q\ x\ y\ ya\ yb\ yc \Longrightarrow \exists f\ fa\ fb\ fc.\ \forall x \in S.\ Q\ x\ (f\ x)\ (fa$
$x)\ (fb\ x)\ (fc\ x)$
$\bigwedge Q.\ \forall x \in S.\ \exists y\ ya\ yb\ yc\ yd.\ Q\ x\ y\ ya\ yb\ yc\ yd \Longrightarrow$
$\exists f\ fa\ fb\ fc\ fd.\ \forall x \in S.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)$
$\bigwedge Q.\ \forall x \in S.\ \exists y\ ya\ yb\ yc\ yd\ ye.\ Q\ x\ y\ ya\ yb\ yc\ yd\ ye \Longrightarrow$
$\exists f\ fa\ fb\ fc\ fd\ fe.\ \forall x \in S.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)\ (fe\ x)$
$\bigwedge Q.\ \forall x \in S.\ \exists y\ ya\ yb\ yc\ yd\ ye\ yf.\ Q\ x\ y\ ya\ yb\ yc\ yd\ ye\ yf \Longrightarrow$
$\exists f\ fa\ fb\ fc\ fd\ fe\ ff.\ \forall x \in S.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)\ (fe\ x)\ (ff\ x)$
$\bigwedge Q.\ \forall x \in S.\ \exists y\ ya\ yb\ yc\ yd\ ye\ yf\ yg.\ Q\ x\ y\ ya\ yb\ yc\ yd\ ye\ yf\ yg \Longrightarrow$

$\exists f\ fa\ fb\ fc\ fd\ fe\ ff\ fg.\ \forall x \in S.\ Q\ x\ (f\ x)\ (fa\ x)\ (fb\ x)\ (fc\ x)\ (fd\ x)\ (fe\ x)\ (ff\ x)\ (fg\ x)$
⟨*proof*⟩

⟨*ML*⟩

hide-fact (**open**) *choices bchoices*

63.2 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by "pat") triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by "nopat") inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.

typedecl *'a symb-list*

consts
Symb-Nil :: *'a symb-list*
Symb-Cons :: *'a* ⇒ *'a symb-list* ⇒ *'a symb-list*

typedecl *pattern*

consts
pat :: *'a* ⇒ *pattern*
nopat :: *'a* ⇒ *pattern*

definition *trigger* :: *pattern symb-list symb-list* ⇒ *bool* ⇒ *bool* **where**
trigger - P = *P*

63.3 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* :: *'a* ⇒ *'a* **where** *fun-app f* = *f*

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*

63.4 Normalization

lemma *case-bool-if* [*abs-def*]: *case-bool x y P* = (*if P then x else y*)
⟨*proof*⟩

lemmas *Ex1-def-raw* = *Ex1-def* [*abs-def*]
lemmas *Ball-def-raw* = *Ball-def* [*abs-def*]
lemmas *Bex-def-raw* = *Bex-def* [*abs-def*]
lemmas *abs-if-raw* = *abs-if* [*abs-def*]
lemmas *min-def-raw* = *min-def* [*abs-def*]
lemmas *max-def-raw* = *max-def* [*abs-def*]

lemma *nat-int′*: $\forall n.\ nat\ (int\ n) = n$ ⟨*proof*⟩
lemma *int-nat-nneg*: $\forall i.\ i \geq 0 \longrightarrow int\ (nat\ i) = i$ ⟨*proof*⟩
lemma *int-nat-neg*: $\forall i.\ i < 0 \longrightarrow int\ (nat\ i) = 0$ ⟨*proof*⟩

lemmas *nat-zero-as-int* = *transfer-nat-int-numerals*(*1*)
lemmas *nat-one-as-int* = *transfer-nat-int-numerals*(*2*)
lemma *nat-numeral-as-int*: $numeral = (\lambda i.\ nat\ (numeral\ i))$ ⟨*proof*⟩
lemma *nat-less-as-int*: $op < = (\lambda a\ b.\ int\ a < int\ b)$ ⟨*proof*⟩
lemma *nat-leq-as-int*: $op \leq = (\lambda a\ b.\ int\ a \leq int\ b)$ ⟨*proof*⟩
lemma *Suc-as-int*: $Suc = (\lambda a.\ nat\ (int\ a + 1))$ ⟨*proof*⟩
lemma *nat-plus-as-int*: $op + = (\lambda a\ b.\ nat\ (int\ a + int\ b))$ ⟨*proof*⟩
lemma *nat-minus-as-int*: $op - = (\lambda a\ b.\ nat\ (int\ a - int\ b))$ ⟨*proof*⟩
lemma *nat-times-as-int*: $op * = (\lambda a\ b.\ nat\ (int\ a * int\ b))$ ⟨*proof*⟩
lemma *nat-div-as-int*: $op\ div = (\lambda a\ b.\ nat\ (int\ a\ div\ int\ b))$ ⟨*proof*⟩
lemma *nat-mod-as-int*: $op\ mod = (\lambda a\ b.\ nat\ (int\ a\ mod\ int\ b))$ ⟨*proof*⟩

lemma *int-Suc*: $int\ (Suc\ n) = int\ n + 1$ ⟨*proof*⟩
lemma *int-plus*: $int\ (n + m) = int\ n + int\ m$ ⟨*proof*⟩
lemma *int-minus*: $int\ (n - m) = int\ (nat\ (int\ n - int\ m))$ ⟨*proof*⟩

63.5 Integer division and modulo for Z3

The following Z3-inspired definitions are overspecified for the case where $l = 0$. This Schönheitsfehler is corrected in the *div-as-z3div* and *mod-as-z3mod* theorems.

definition *z3div* :: *int* ⇒ *int* ⇒ *int* **where**
$z3div\ k\ l = (if\ l \geq 0\ then\ k\ div\ l\ else\ -\ (k\ div\ -\ l))$

definition *z3mod* :: *int* ⇒ *int* ⇒ *int* **where**
$z3mod\ k\ l = k\ mod\ (if\ l \geq 0\ then\ l\ else\ -\ l)$

lemma *div-as-z3div*:
$\forall k\ l.\ k\ div\ l = (if\ l = 0\ then\ 0\ else\ if\ l > 0\ then\ z3div\ k\ l\ else\ z3div\ (-\ k)\ (-\ l))$
⟨*proof*⟩

lemma *mod-as-z3mod*:
$\forall k\ l.\ k\ mod\ l = (if\ l = 0\ then\ k\ else\ if\ l > 0\ then\ z3mod\ k\ l\ else\ -\ z3mod\ (-\ k)$

(− *l*))
⟨*proof*⟩

63.6 Setup

⟨*ML*⟩

63.7 Configuration

The current configuration can be printed by the command *smt-status*, which shows the values of most options.

63.8 General configuration options

The option *smt-solver* can be used to change the target SMT solver. The possible values can be obtained from the *smt-status* command.

declare [[*smt-solver* = *z3*]]

Since SMT solvers are potentially nonterminating, there is a timeout (given in seconds) to restrict their runtime.

declare [[*smt-timeout* = *20*]]

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

declare [[*smt-random-seed* = *1*]]

In general, the binding to SMT solvers runs as an oracle, i.e, the SMT solvers are fully trusted without additional checks. The following option can cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently only implemented for Z3.

declare [[*smt-oracle* = *false*]]

Each SMT solver provides several commandline options to tweak its behaviour. They can be passed to the solver by setting the following options.

declare [[*cvc3-options* =]]
declare [[*cvc4-options* = *−−full−saturate−quant −−inst−when=full−last−call −−inst−no−entail −−term−db−mode=relevant −−multi−trigger−linear*]]
declare [[*verit-options* = *−−index−sorts −−index−fresh−sorts*]]
declare [[*z3-options* =]]

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

declare [[*smt-infer-triggers* = *false*]]

Enable the following option to use built-in support for datatypes, codatatypes, and records in CVC4. Currently, this is implemented only in oracle mode.

declare [[*cvc4-extensions* = *false*]]

Enable the following option to use built-in support for div/mod, datatypes, and records in Z3. Currently, this is implemented only in oracle mode.

declare [[*z3-extensions* = *false*]]

63.9 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. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

declare [[*smt-certificates* =]]

The option *smt-read-only-certificates* 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-read-only-certificates* = *false*]]

63.10 Tracing

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to *false*.

declare [[*smt-verbose* = *true*]]

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*]]

63.11 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
NO-MATCH-def

lemma [*z3-rule*]:
$(P \wedge Q) = (\neg (\neg P \vee \neg Q))$
$(P \wedge Q) = (\neg (\neg Q \vee \neg P))$
$(\neg P \wedge Q) = (\neg (P \vee \neg Q))$
$(\neg P \wedge Q) = (\neg (\neg Q \vee P))$
$(P \wedge \neg Q) = (\neg (\neg P \vee Q))$
$(P \wedge \neg Q) = (\neg (Q \vee \neg P))$
$(\neg P \wedge \neg Q) = (\neg (P \vee Q))$
$(\neg P \wedge \neg Q) = (\neg (Q \vee P))$
⟨*proof*⟩

lemma [*z3-rule*]:
$(P \longrightarrow Q) = (Q \vee \neg P)$
$(\neg P \longrightarrow Q) = (P \vee Q)$
$(\neg P \longrightarrow Q) = (Q \vee P)$
$(True \longrightarrow P) = P$
$(P \longrightarrow True) = True$
$(False \longrightarrow P) = True$
$(P \longrightarrow P) = True$
$(\neg (A \longleftrightarrow \neg B)) \longleftrightarrow (A \longleftrightarrow B)$
⟨*proof*⟩

lemma [*z3-rule*]:
$((P = Q) \longrightarrow R) = (R \mid (Q = (\neg P)))$
⟨*proof*⟩

lemma [*z3-rule*]:
$(\neg True) = False$
$(\neg False) = True$
$(x = x) = True$
$(P = True) = P$
$(True = P) = P$
$(P = False) = (\neg P)$
$(False = P) = (\neg P)$
$((\neg P) = P) = False$
$(P = (\neg P)) = False$
$((\neg P) = (\neg Q)) = (P = Q)$
$\neg (P = (\neg Q)) = (P = Q)$
$\neg ((\neg P) = Q) = (P = Q)$
$(P \neq Q) = (Q = (\neg P))$
$(P = Q) = ((\neg P \vee Q) \wedge (P \vee \neg Q))$

$(P \neq Q) = ((\neg\ P \vee \neg\ Q) \wedge (P \vee Q))$
$\langle proof \rangle$

lemma [*z3-rule*]:
$(\textit{if } P \textit{ then } P \textit{ else } \neg\ P) = \textit{True}$
$(\textit{if } \neg\ P \textit{ then } \neg\ P \textit{ else } P) = \textit{True}$
$(\textit{if } P \textit{ then True else False}) = P$
$(\textit{if } P \textit{ then False else True}) = (\neg\ P)$
$(\textit{if } P \textit{ then } Q \textit{ else True}) = ((\neg\ P) \vee Q)$
$(\textit{if } P \textit{ then } Q \textit{ else True}) = (Q \vee (\neg\ P))$
$(\textit{if } P \textit{ then } Q \textit{ else } \neg\ Q) = (P = Q)$
$(\textit{if } P \textit{ then } Q \textit{ else } \neg\ Q) = (Q = P)$
$(\textit{if } P \textit{ then } \neg\ Q \textit{ else } Q) = (P = (\neg\ Q))$
$(\textit{if } P \textit{ then } \neg\ Q \textit{ else } Q) = ((\neg\ Q) = P)$
$(\textit{if } \neg\ P \textit{ then } x \textit{ else } y) = (\textit{if } P \textit{ then } y \textit{ else } x)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } x) = (\textit{if } P \wedge (\neg\ Q) \textit{ then } y \textit{ else } x)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } x) = (\textit{if } (\neg\ Q) \wedge P \textit{ then } y \textit{ else } x)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } y) = (\textit{if } P \wedge Q \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } (\textit{if } Q \textit{ then } x \textit{ else } y) \textit{ else } y) = (\textit{if } Q \wedge P \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } x \textit{ else if } P \textit{ then } y \textit{ else } z) = (\textit{if } P \textit{ then } x \textit{ else } z)$
$(\textit{if } P \textit{ then } x \textit{ else if } Q \textit{ then } x \textit{ else } y) = (\textit{if } P \vee Q \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } x \textit{ else if } Q \textit{ then } x \textit{ else } y) = (\textit{if } Q \vee P \textit{ then } x \textit{ else } y)$
$(\textit{if } P \textit{ then } x = y \textit{ else } x = z) = (x = (\textit{if } P \textit{ then } y \textit{ else } z))$
$(\textit{if } P \textit{ then } x = y \textit{ else } y = z) = (y = (\textit{if } P \textit{ then } x \textit{ else } z))$
$(\textit{if } P \textit{ then } x = y \textit{ else } z = y) = (y = (\textit{if } P \textit{ then } x \textit{ else } z))$
$\langle proof \rangle$

lemma [*z3-rule*]:
$0 + (x::int) = x$
$x + 0 = x$
$x + x = 2 * x$
$0 * x = 0$
$1 * x = x$
$x + y = y + x$
$\langle proof \rangle$

lemma [*z3-rule*]:
$P = Q \vee P \vee Q$
$P = Q \vee \neg\ P \vee \neg\ Q$
$(\neg\ P) = Q \vee \neg\ P \vee Q$
$(\neg\ P) = Q \vee P \vee \neg\ Q$
$P = (\neg\ Q) \vee \neg\ P \vee Q$
$P = (\neg\ Q) \vee P \vee \neg\ Q$
$P \neq Q \vee P \vee \neg\ Q$
$P \neq Q \vee \neg\ P \vee Q$
$P \neq (\neg\ Q) \vee P \vee Q$
$(\neg\ P) \neq Q \vee P \vee Q$
$P \vee Q \vee P \neq (\neg\ Q)$
$P \vee Q \vee (\neg\ P) \neq Q$

$P \lor \neg\ Q \lor P \neq Q$
$\neg\ P \lor Q \lor P \neq Q$
$P \lor y = (\textit{if } P \textit{ then } x \textit{ else } y)$
$P \lor (\textit{if } P \textit{ then } x \textit{ else } y) = y$
$\neg\ P \lor x = (\textit{if } P \textit{ then } x \textit{ else } y)$
$\neg\ P \lor (\textit{if } P \textit{ then } x \textit{ else } y) = x$
$P \lor R \lor \neg\ (\textit{if } P \textit{ then } Q \textit{ else } R)$
$\neg\ P \lor Q \lor \neg\ (\textit{if } P \textit{ then } Q \textit{ else } R)$
$\neg\ (\textit{if } P \textit{ then } Q \textit{ else } R) \lor \neg\ P \lor Q$
$\neg\ (\textit{if } P \textit{ then } Q \textit{ else } R) \lor P \lor R$
$(\textit{if } P \textit{ then } Q \textit{ else } R) \lor \neg\ P \lor \neg\ Q$
$(\textit{if } P \textit{ then } Q \textit{ else } R) \lor P \lor \neg\ R$
$(\textit{if } P \textit{ then } \neg\ Q \textit{ else } R) \lor \neg\ P \lor Q$
$(\textit{if } P \textit{ then } Q \textit{ else } \neg\ R) \lor P \lor R$
⟨*proof*⟩

hide-type (**open**) *symb-list pattern*
hide-const (**open**) *Symb-Nil Symb-Cons trigger pat nopat fun-app z3div z3mod*

end

64 Sledgehammer: Isabelle–ATP Linkup

theory *Sledgehammer*
imports *Presburger SMT*
keywords
sledgehammer :: *diag* **and**
sledgehammer-params :: *thy-decl*
begin

lemma *size-ne-size-imp-ne*: $\textit{size } x \neq \textit{size } y \Longrightarrow x \neq y$
⟨*proof*⟩

⟨*ML*⟩

end

65 Numeric types for code generation onto target language numerals only

theory *Code-Numeral*
imports *Nat-Transfer Divides Lifting*
begin

65.1 Type of target language integers

typedef $\textit{integer} = \textit{UNIV} :: \textit{int set}$
morphisms *int-of-integer integer-of-int* ⟨*proof*⟩

setup-lifting *type-definition-integer*

lemma *integer-eq-iff*:
$k = l \longleftrightarrow$ *int-of-integer* $k =$ *int-of-integer* l
⟨*proof*⟩

lemma *integer-eqI*:
int-of-integer $k =$ *int-of-integer* $l \Longrightarrow k = l$
⟨*proof*⟩

lemma *int-of-integer-integer-of-int* [*simp*]:
int-of-integer (*integer-of-int* k) $= k$
⟨*proof*⟩

lemma *integer-of-int-int-of-integer* [*simp*]:
integer-of-int (*int-of-integer* k) $= k$
⟨*proof*⟩

instantiation *integer* :: *ring-1*
begin

lift-definition *zero-integer* :: *integer*
is *0* :: *int*
⟨*proof*⟩

declare *zero-integer.rep-eq* [*simp*]

lift-definition *one-integer* :: *integer*
is *1* :: *int*
⟨*proof*⟩

declare *one-integer.rep-eq* [*simp*]

lift-definition *plus-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *plus* :: *int* ⇒ *int* ⇒ *int*
⟨*proof*⟩

declare *plus-integer.rep-eq* [*simp*]

lift-definition *uminus-integer* :: *integer* ⇒ *integer*
is *uminus* :: *int* ⇒ *int*
⟨*proof*⟩

declare *uminus-integer.rep-eq* [*simp*]

lift-definition *minus-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *minus* :: *int* ⇒ *int* ⇒ *int*
⟨*proof*⟩

declare *minus-integer.rep-eq* [*simp*]

lift-definition *times-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *times* :: *int* ⇒ *int* ⇒ *int*
⟨*proof*⟩

declare *times-integer.rep-eq* [*simp*]

instance ⟨*proof*⟩

end

instance *integer* :: *Rings.dvd* ⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun pcr-integer (*rel-fun pcr-integer HOL.iff*) *Rings.dvd Rings.dvd*
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-integer (*of-nat* :: *nat* ⇒ *int*) (*of-nat* :: *nat* ⇒ *integer*)
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-integer (λk :: *int*. *k* :: *int*) (*of-int* :: *int* ⇒ *integer*)
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-integer (*numeral* :: *num* ⇒ *int*) (*numeral* :: *num* ⇒ *integer*)
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun HOL.eq (*rel-fun HOL.eq pcr-integer*) (*Num.sub* :: - ⇒ - ⇒ *int*) (*Num.sub* :: - ⇒ - ⇒ *integer*)
⟨*proof*⟩

lemma *int-of-integer-of-nat* [*simp*]:
int-of-integer (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lift-definition *integer-of-nat* :: *nat* ⇒ *integer*
is *of-nat* :: *nat* ⇒ *int*
⟨*proof*⟩

lemma *integer-of-nat-eq-of-nat* [*code*]:
integer-of-nat = *of-nat*
⟨*proof*⟩

lemma *int-of-integer-integer-of-nat* [*simp*]:

int-of-integer (*integer-of-nat n*) = *of-nat n*
⟨*proof*⟩

lift-definition *nat-of-integer* :: *integer* ⇒ *nat*
is *Int.nat*
⟨*proof*⟩

lemma *nat-of-integer-of-nat* [*simp*]:
nat-of-integer (*of-nat n*) = *n*
⟨*proof*⟩

lemma *int-of-integer-of-int* [*simp*]:
int-of-integer (*of-int k*) = *k*
⟨*proof*⟩

lemma *nat-of-integer-integer-of-nat* [*simp*]:
nat-of-integer (*integer-of-nat n*) = *n*
⟨*proof*⟩

lemma *integer-of-int-eq-of-int* [*simp*, *code-abbrev*]:
integer-of-int = *of-int*
⟨*proof*⟩

lemma *of-int-integer-of* [*simp*]:
of-int (*int-of-integer k*) = (*k* :: *integer*)
⟨*proof*⟩

lemma *int-of-integer-numeral* [*simp*]:
int-of-integer (*numeral k*) = *numeral k*
⟨*proof*⟩

lemma *int-of-integer-sub* [*simp*]:
int-of-integer (*Num.sub k l*) = *Num.sub k l*
⟨*proof*⟩

definition *integer-of-num* :: *num* ⇒ *integer*
where [*simp*]: *integer-of-num* = *numeral*

lemma *integer-of-num* [*code*]:
integer-of-num Num.One = *1*
integer-of-num (*Num.Bit0 n*) = (*let k* = *integer-of-num n in k* + *k*)
integer-of-num (*Num.Bit1 n*) = (*let k* = *integer-of-num n in k* + *k* + *1*)
⟨*proof*⟩

lemma *integer-of-num-triv*:
integer-of-num Num.One = *1*
integer-of-num (*Num.Bit0 Num.One*) = *2*
⟨*proof*⟩

instantiation *integer* :: {*linordered-idom*, *equal*}
begin

lift-definition *abs-integer* :: *integer* ⇒ *integer*
is *abs* :: *int* ⇒ *int*
⟨*proof*⟩

declare *abs-integer.rep-eq* [*simp*]

lift-definition *sgn-integer* :: *integer* ⇒ *integer*
is *sgn* :: *int* ⇒ *int*
⟨*proof*⟩

declare *sgn-integer.rep-eq* [*simp*]

lift-definition *less-eq-integer* :: *integer* ⇒ *integer* ⇒ *bool*
is *less-eq* :: *int* ⇒ *int* ⇒ *bool*
⟨*proof*⟩

lift-definition *less-integer* :: *integer* ⇒ *integer* ⇒ *bool*
is *less* :: *int* ⇒ *int* ⇒ *bool*
⟨*proof*⟩

lift-definition *equal-integer* :: *integer* ⇒ *integer* ⇒ *bool*
is *HOL.equal* :: *int* ⇒ *int* ⇒ *bool*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma [*transfer-rule*]:
rel-fun pcr-integer (*rel-fun pcr-integer pcr-integer*) (*min* :: - ⇒ - ⇒ *int*) (*min* :: - ⇒ - ⇒ *integer*)
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun pcr-integer (*rel-fun pcr-integer pcr-integer*) (*max* :: - ⇒ - ⇒ *int*) (*max* :: - ⇒ - ⇒ *integer*)
⟨*proof*⟩

lemma *int-of-integer-min* [*simp*]:
int-of-integer (*min k l*) = *min* (*int-of-integer k*) (*int-of-integer l*)
⟨*proof*⟩

lemma *int-of-integer-max* [*simp*]:
int-of-integer (*max k l*) = *max* (*int-of-integer k*) (*int-of-integer l*)

⟨*proof*⟩

lemma *nat-of-integer-non-positive* [*simp*]:
$k \leq 0 \Longrightarrow$ *nat-of-integer* $k = 0$
⟨*proof*⟩

lemma *of-nat-of-integer* [*simp*]:
of-nat (*nat-of-integer* k) = *max* 0 k
⟨*proof*⟩

instantiation *integer* :: *normalization-semidom*
begin

lift-definition *normalize-integer* :: *integer* ⇒ *integer*
is *normalize* :: *int* ⇒ *int*
⟨*proof*⟩

declare *normalize-integer.rep-eq* [*simp*]

lift-definition *unit-factor-integer* :: *integer* ⇒ *integer*
is *unit-factor* :: *int* ⇒ *int*
⟨*proof*⟩

declare *unit-factor-integer.rep-eq* [*simp*]

lift-definition *divide-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *divide* :: *int* ⇒ *int* ⇒ *int*
⟨*proof*⟩

declare *divide-integer.rep-eq* [*simp*]

instance
⟨*proof*⟩

end

instantiation *integer* :: *ring-div*
begin

lift-definition *modulo-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *modulo* :: *int* ⇒ *int* ⇒ *int*
⟨*proof*⟩

declare *modulo-integer.rep-eq* [*simp*]

instance
⟨*proof*⟩

end

instantiation *integer* :: *semiring-numeral-div*
begin

definition *divmod-integer* :: *num* ⇒ *num* ⇒ *integer* × *integer*
where
divmod-integer′-def: *divmod-integer m n* = (*numeral m div numeral n*, *numeral m mod numeral n*)

definition *divmod-step-integer* :: *num* ⇒ *integer* × *integer* ⇒ *integer* × *integer*
where
divmod-step-integer l qr = (*let* (*q*, *r*) = *qr*
in if r ≥ *numeral l then* (*2* ∗ *q* + *1*, *r* − *numeral l*)
else (*2* ∗ *q*, *r*))

instance ⟨*proof*⟩

end

declare *divmod-algorithm-code* [**where** *?′a* = *integer*,
folded integer-of-num-def, *unfolded integer-of-num-triv*,
code]

lemma *integer-of-nat-0*: *integer-of-nat 0* = *0*
⟨*proof*⟩

lemma *integer-of-nat-1*: *integer-of-nat 1* = *1*
⟨*proof*⟩

lemma *integer-of-nat-numeral*:
integer-of-nat (*numeral n*) = *numeral n*
⟨*proof*⟩

65.2 Code theorems for target language integers

Constructors

definition *Pos* :: *num* ⇒ *integer*
where
[*simp*, *code-post*]: *Pos* = *numeral*

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-integer numeral Pos
⟨*proof*⟩

lemma *Pos-fold* [*code-unfold*]:
numeral Num.One = *Pos Num.One*
numeral (*Num.Bit0 k*) = *Pos* (*Num.Bit0 k*)
numeral (*Num.Bit1 k*) = *Pos* (*Num.Bit1 k*)
⟨*proof*⟩

definition *Neg* :: *num* ⇒ *integer*
where
[*simp*, *code-abbrev*]: *Neg n* = − *Pos n*

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-integer (λ*n*. − *numeral n*) *Neg*
⟨*proof*⟩

code-datatype *0*::*integer Pos Neg*

A further pair of constructors for generated computations

context
begin

qualified definition *positive* :: *num* ⇒ *integer*
where [*simp*]: *positive* = *numeral*

qualified definition *negative* :: *num* ⇒ *integer*
where [*simp*]: *negative* = *uminus* ∘ *numeral*

lemma [*code-computation-unfold*]:
numeral = *positive*
Pos = *positive*
Neg = *negative*
⟨*proof*⟩

end

Auxiliary operations

lift-definition *dup* :: *integer* ⇒ *integer*
is λ*k*::*int*. *k* + *k*
⟨*proof*⟩

lemma *dup-code* [*code*]:
dup 0 = *0*
dup (*Pos n*) = *Pos* (*Num.Bit0 n*)
dup (*Neg n*) = *Neg* (*Num.Bit0 n*)
⟨*proof*⟩

lift-definition *sub* :: *num* ⇒ *num* ⇒ *integer*
is λ*m n*. *numeral m* − *numeral n* :: *int*
⟨*proof*⟩

lemma *sub-code* [*code*]:
sub Num.One Num.One = *0*
sub (*Num.Bit0 m*) *Num.One* = *Pos* (*Num.BitM m*)
sub (*Num.Bit1 m*) *Num.One* = *Pos* (*Num.Bit0 m*)
sub Num.One (*Num.Bit0 n*) = *Neg* (*Num.BitM n*)

sub Num.One (Num.Bit1 n) = Neg (Num.Bit0 n)
sub (Num.Bit0 m) (Num.Bit0 n) = dup (sub m n)
sub (Num.Bit1 m) (Num.Bit1 n) = dup (sub m n)
sub (Num.Bit1 m) (Num.Bit0 n) = dup (sub m n) + 1
sub (Num.Bit0 m) (Num.Bit1 n) = dup (sub m n) − 1
⟨*proof*⟩

Implementations

lemma *one-integer-code* [*code*, *code-unfold*]:
1 = Pos Num.One
⟨*proof*⟩

lemma *plus-integer-code* [*code*]:
k + 0 = (k::integer)
0 + l = (l::integer)
Pos m + Pos n = Pos (m + n)
Pos m + Neg n = sub m n
Neg m + Pos n = sub n m
Neg m + Neg n = Neg (m + n)
⟨*proof*⟩

lemma *uminus-integer-code* [*code*]:
uminus 0 = (0::integer)
uminus (Pos m) = Neg m
uminus (Neg m) = Pos m
⟨*proof*⟩

lemma *minus-integer-code* [*code*]:
k − 0 = (k::integer)
0 − l = uminus (l::integer)
Pos m − Pos n = sub m n
Pos m − Neg n = Pos (m + n)
Neg m − Pos n = Neg (m + n)
Neg m − Neg n = sub n m
⟨*proof*⟩

lemma *abs-integer-code* [*code*]:
|k| = (if (k::integer) < 0 then − k else k)
⟨*proof*⟩

lemma *sgn-integer-code* [*code*]:
sgn k = (if k = 0 then 0 else if (k::integer) < 0 then − 1 else 1)
⟨*proof*⟩

lemma *times-integer-code* [*code*]:
k ∗ 0 = (0::integer)
0 ∗ l = (0::integer)
Pos m ∗ Pos n = Pos (m ∗ n)
Pos m ∗ Neg n = Neg (m ∗ n)

$Neg\ m * Pos\ n = Neg\ (m * n)$
$Neg\ m * Neg\ n = Pos\ (m * n)$
⟨*proof*⟩

lemma *normalize-integer-code* [*code*]:
$normalize = (abs :: integer \Rightarrow integer)$
⟨*proof*⟩

lemma *unit-factor-integer-code* [*code*]:
$unit\text{-}factor = (sgn :: integer \Rightarrow integer)$
⟨*proof*⟩

definition $divmod\text{-}integer :: integer \Rightarrow integer \Rightarrow integer \times integer$
where
$divmod\text{-}integer\ k\ l = (k\ div\ l,\ k\ mod\ l)$

lemma *fst-divmod* [*simp*]:
$fst\ (divmod\text{-}integer\ k\ l) = k\ div\ l$
⟨*proof*⟩

lemma *snd-divmod* [*simp*]:
$snd\ (divmod\text{-}integer\ k\ l) = k\ mod\ l$
⟨*proof*⟩

definition $divmod\text{-}abs :: integer \Rightarrow integer \Rightarrow integer \times integer$
where
$divmod\text{-}abs\ k\ l = (|k|\ div\ |l|,\ |k|\ mod\ |l|)$

lemma *fst-divmod-abs* [*simp*]:
$fst\ (divmod\text{-}abs\ k\ l) = |k|\ div\ |l|$
⟨*proof*⟩

lemma *snd-divmod-abs* [*simp*]:
$snd\ (divmod\text{-}abs\ k\ l) = |k|\ mod\ |l|$
⟨*proof*⟩

lemma *divmod-abs-code* [*code*]:
$divmod\text{-}abs\ (Pos\ k)\ (Pos\ l) = divmod\ k\ l$
$divmod\text{-}abs\ (Neg\ k)\ (Neg\ l) = divmod\ k\ l$
$divmod\text{-}abs\ (Neg\ k)\ (Pos\ l) = divmod\ k\ l$
$divmod\text{-}abs\ (Pos\ k)\ (Neg\ l) = divmod\ k\ l$
$divmod\text{-}abs\ j\ 0 = (0,\ |j|)$
$divmod\text{-}abs\ 0\ j = (0,\ 0)$
⟨*proof*⟩

lemma *divmod-integer-code* [*code*]:
$divmod\text{-}integer\ k\ l =$
$(if\ k = 0\ then\ (0,\ 0)\ else\ if\ l = 0\ then\ (0,\ k)\ else$
$(apsnd \circ times \circ sgn)\ l\ (if\ sgn\ k = sgn\ l$

then divmod-abs k l
else (let (r, s) = divmod-abs k l in
if s = 0 then (− r, 0) else (− r − 1, |l| − s))))
⟨*proof*⟩

lemma *div-integer-code* [*code*]:
k div l = fst (divmod-integer k l)
⟨*proof*⟩

lemma *mod-integer-code* [*code*]:
k mod l = snd (divmod-integer k l)
⟨*proof*⟩

lemma *equal-integer-code* [*code*]:
HOL.equal 0 (0::integer) ⟷ True
HOL.equal 0 (Pos l) ⟷ False
HOL.equal 0 (Neg l) ⟷ False
HOL.equal (Pos k) 0 ⟷ False
HOL.equal (Pos k) (Pos l) ⟷ HOL.equal k l
HOL.equal (Pos k) (Neg l) ⟷ False
HOL.equal (Neg k) 0 ⟷ False
HOL.equal (Neg k) (Pos l) ⟷ False
HOL.equal (Neg k) (Neg l) ⟷ HOL.equal k l
⟨*proof*⟩

lemma *equal-integer-refl* [*code nbe*]:
HOL.equal (k::integer) k ⟷ True
⟨*proof*⟩

lemma *less-eq-integer-code* [*code*]:
0 ≤ (0::integer) ⟷ True
0 ≤ Pos l ⟷ True
0 ≤ Neg l ⟷ False
Pos k ≤ 0 ⟷ False
Pos k ≤ Pos l ⟷ k ≤ l
Pos k ≤ Neg l ⟷ False
Neg k ≤ 0 ⟷ True
Neg k ≤ Pos l ⟷ True
Neg k ≤ Neg l ⟷ l ≤ k
⟨*proof*⟩

lemma *less-integer-code* [*code*]:
0 < (0::integer) ⟷ False
0 < Pos l ⟷ True
0 < Neg l ⟷ False
Pos k < 0 ⟷ False
Pos k < Pos l ⟷ k < l
Pos k < Neg l ⟷ False
Neg k < 0 ⟷ True

$Neg\ k < Pos\ l \longleftrightarrow True$
$Neg\ k < Neg\ l \longleftrightarrow l < k$
⟨*proof*⟩

lift-definition *num-of-integer* :: *integer* ⇒ *num*
is *num-of-nat* ∘ *nat*
⟨*proof*⟩

lemma *num-of-integer-code* [*code*]:
num-of-integer k = (*if k* ≤ *1 then Num.One*
else let
(*l*, *j*) = *divmod-integer k 2*;
l' = *num-of-integer l*;
l'' = *l'* + *l'*
in if j = *0 then l'' else l''* + *Num.One*)
⟨*proof*⟩

lemma *nat-of-integer-code* [*code*]:
nat-of-integer k = (*if k* ≤ *0 then 0*
else let
(*l*, *j*) = *divmod-integer k 2*;
l' = *nat-of-integer l*;
l'' = *l'* + *l'*
in if j = *0 then l'' else l''* + *1*)
⟨*proof*⟩

lemma *int-of-integer-code* [*code*]:
int-of-integer k = (*if k* < *0 then* − (*int-of-integer* (− *k*))
else if k = *0 then 0*
else let
(*l*, *j*) = *divmod-integer k 2*;
l' = *2* ∗ *int-of-integer l*
in if j = *0 then l' else l'* + *1*)
⟨*proof*⟩

lemma *integer-of-int-code* [*code*]:
integer-of-int k = (*if k* < *0 then* − (*integer-of-int* (− *k*))
else if k = *0 then 0*
else let
l = *2* ∗ *integer-of-int* (*k div 2*);
j = *k mod 2*
in if j = *0 then l else l* + *1*)
⟨*proof*⟩

hide-const (**open**) *Pos Neg sub dup divmod-abs*

65.3 Serializer setup for target language integers

code-reserved *Eval int Integer abs*

code-printing
  type-constructor *integer* ⇀
    (*SML*) *IntInf.int*
    and (*OCaml*) *Big′-int.big′-int*
    and (*Haskell*) *Integer*
    and (*Scala*) *BigInt*
    and (*Eval*) *int*
| **class-instance** *integer* :: *equal* ⇀
    (*Haskell*) −

code-printing
  constant *0*::*integer* ⇀
    (*SML*) !(*0*/ :/ *IntInf.int*)
    and (*OCaml*) *Big′-int.zero′-big′-int*
    and (*Haskell*) !(*0*/ ::/ *Integer*)
    and (*Scala*) *BigInt*(*0*)

⟨*ML*⟩

code-printing
  constant *plus* :: *integer* ⇒ - ⇒ - ⇀
    (*SML*) *IntInf.*+ ((-), (-))
    and (*OCaml*) *Big′-int.add′-big′-int*
    and (*Haskell*) **infixl** *6* +
    and (*Scala*) **infixl** *7* +
    and (*Eval*) **infixl** *8* +
| **constant** *uminus* :: *integer* ⇒ - ⇀
    (*SML*) *IntInf.*~
    and (*OCaml*) *Big′-int.minus′-big′-int*
    and (*Haskell*) *negate*
    and (*Scala*) !(− -)
    and (*Eval*) ~/ -
| **constant** *minus* :: *integer* ⇒ - ⇀
    (*SML*) *IntInf.*− ((-), (-))
    and (*OCaml*) *Big′-int.sub′-big′-int*
    and (*Haskell*) **infixl** *6* −
    and (*Scala*) **infixl** *7* −
    and (*Eval*) **infixl** *8* −
| **constant** *Code-Numeral.dup* ⇀
    (*SML*) *IntInf.*∗/ (*2*,/ (-))
    and (*OCaml*) *Big′-int.mult′-big′-int*/ (*Big′-int.big′-int′-of′-int*/ *2*)
    and (*Haskell*) !(*2* ∗ -)
    and (*Scala*) !(*2* ∗ -)
    and (*Eval*) !(*2* ∗ -)
| **constant** *Code-Numeral.sub* ⇀
    (*SML*) !(*raise*/ *Fail*/ *sub*)
    and (*OCaml*) *failwith*/ *sub*
    and (*Haskell*) *error*/ *sub*

and (*Scala*) !*sys.error*(*sub*)
| **constant** *times* :: *integer* ⇒ - ⇒ - ⇀
(*SML*) *IntInf*.∗ ((-), (-))
and (*OCaml*) *Big′-int.mult′-big′-int*
and (*Haskell*) **infixl** *7* ∗
and (*Scala*) **infixl** *8* ∗
and (*Eval*) **infixl** *9* ∗
| **constant** *Code-Numeral.divmod-abs* ⇀
(*SML*) *IntInf*.*divMod*/ (*IntInf*.*abs* -,/ *IntInf*.*abs* -)
and (*OCaml*) *Big′-int.quomod′-big′-int*/ (*Big′-int.abs′-big′-int* -)/ (*Big′-int.abs′-big′-int* -)
and (*Haskell*) *divMod*/ (*abs* -)/ (*abs* -)
and (*Scala*) !((*k*: *BigInt*) => (*l*: *BigInt*) =>/ *if* (*l* == *0*)/ (*BigInt*(*0*), *k*) *else*/ (*k*.*abs* ′/% *l*.*abs*))
and (*Eval*) *Integer.div′-mod*/ (*abs* -)/ (*abs* -)
| **constant** *HOL.equal* :: *integer* ⇒ - ⇒ *bool* ⇀
(*SML*) !((- : *IntInf*.*int*) = -)
and (*OCaml*) *Big′-int.eq′-big′-int*
and (*Haskell*) **infix** *4* ==
and (*Scala*) **infixl** *5* ==
and (*Eval*) **infixl** *6* =
| **constant** *less-eq* :: *integer* ⇒ - ⇒ *bool* ⇀
(*SML*) *IntInf*.<= ((-), (-))
and (*OCaml*) *Big′-int.le′-big′-int*
and (*Haskell*) **infix** *4* <=
and (*Scala*) **infixl** *4* <=
and (*Eval*) **infixl** *6* <=
| **constant** *less* :: *integer* ⇒ - ⇒ *bool* ⇀
(*SML*) *IntInf*.< ((-), (-))
and (*OCaml*) *Big′-int.lt′-big′-int*
and (*Haskell*) **infix** *4* <
and (*Scala*) **infixl** *4* <
and (*Eval*) **infixl** *6* <
| **constant** *abs* :: *integer* ⇒ - ⇀
(*SML*) *IntInf*.*abs*
and (*OCaml*) *Big′-int.abs′-big′-int*
and (*Haskell*) *Prelude.abs*
and (*Scala*) -.*abs*
and (*Eval*) *abs*

code-identifier
code-module *Code-Numeral* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

65.4 Type of target language naturals

typedef *natural* = *UNIV* :: *nat set*
morphisms *nat-of-natural natural-of-nat* ⟨*proof*⟩

setup-lifting *type-definition-natural*

lemma *natural-eq-iff* [*termination-simp*]:
m = *n* ⟷ *nat-of-natural m* = *nat-of-natural n*
⟨*proof*⟩

lemma *natural-eqI*:
nat-of-natural m = *nat-of-natural n* ⟹ *m* = *n*
⟨*proof*⟩

lemma *nat-of-natural-of-nat-inverse* [*simp*]:
nat-of-natural (*natural-of-nat n*) = *n*
⟨*proof*⟩

lemma *natural-of-nat-of-natural-inverse* [*simp*]:
natural-of-nat (*nat-of-natural n*) = *n*
⟨*proof*⟩

instantiation *natural* :: {*comm-monoid-diff*, *semiring-1*}
begin

lift-definition *zero-natural* :: *natural*
is *0* :: *nat*
⟨*proof*⟩

declare *zero-natural.rep-eq* [*simp*]

lift-definition *one-natural* :: *natural*
is *1* :: *nat*
⟨*proof*⟩

declare *one-natural.rep-eq* [*simp*]

lift-definition *plus-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *plus* :: *nat* ⇒ *nat* ⇒ *nat*
⟨*proof*⟩

declare *plus-natural.rep-eq* [*simp*]

lift-definition *minus-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *minus* :: *nat* ⇒ *nat* ⇒ *nat*
⟨*proof*⟩

declare *minus-natural.rep-eq* [*simp*]

lift-definition *times-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *times* :: *nat* ⇒ *nat* ⇒ *nat*
⟨*proof*⟩

declare *times-natural.rep-eq* [*simp*]

instance ⟨*proof*⟩

end

instance *natural* :: *Rings.dvd* ⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun pcr-natural (*rel-fun pcr-natural HOL.iff*) *Rings.dvd Rings.dvd*
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-natural (λn::*nat*. n) (*of-nat* :: *nat* ⇒ *natural*)
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-natural (*numeral* :: *num* ⇒ *nat*) (*numeral* :: *num* ⇒ *natural*)
⟨*proof*⟩

lemma *nat-of-natural-of-nat* [*simp*]:
nat-of-natural (*of-nat n*) = *n*
⟨*proof*⟩

lemma *natural-of-nat-of-nat* [*simp*, *code-abbrev*]:
natural-of-nat = *of-nat*
⟨*proof*⟩

lemma *of-nat-of-natural* [*simp*]:
of-nat (*nat-of-natural n*) = *n*
⟨*proof*⟩

lemma *nat-of-natural-numeral* [*simp*]:
nat-of-natural (*numeral k*) = *numeral k*
⟨*proof*⟩

instantiation *natural* :: {*linordered-semiring*, *equal*}
begin

lift-definition *less-eq-natural* :: *natural* ⇒ *natural* ⇒ *bool*
is *less-eq* :: *nat* ⇒ *nat* ⇒ *bool*
⟨*proof*⟩

declare *less-eq-natural.rep-eq* [*termination-simp*]

lift-definition *less-natural* :: *natural* ⇒ *natural* ⇒ *bool*
is *less* :: *nat* ⇒ *nat* ⇒ *bool*
⟨*proof*⟩

declare *less-natural.rep-eq* [*termination-simp*]

lift-definition *equal-natural* :: *natural* ⇒ *natural* ⇒ *bool*
is *HOL.equal* :: *nat* ⇒ *nat* ⇒ *bool*
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma [*transfer-rule*]:
rel-fun pcr-natural (*rel-fun pcr-natural pcr-natural*) (*min* :: - ⇒ - ⇒ *nat*) (*min* :: - ⇒ - ⇒ *natural*)
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun pcr-natural (*rel-fun pcr-natural pcr-natural*) (*max* :: - ⇒ - ⇒ *nat*) (*max* :: - ⇒ - ⇒ *natural*)
⟨*proof*⟩

lemma *nat-of-natural-min* [*simp*]:
nat-of-natural (*min k l*) = *min* (*nat-of-natural k*) (*nat-of-natural l*)
⟨*proof*⟩

lemma *nat-of-natural-max* [*simp*]:
nat-of-natural (*max k l*) = *max* (*nat-of-natural k*) (*nat-of-natural l*)
⟨*proof*⟩

instantiation *natural* :: {*semiring-div*, *normalization-semidom*}
begin

lift-definition *normalize-natural* :: *natural* ⇒ *natural*
is *normalize* :: *nat* ⇒ *nat*
⟨*proof*⟩

declare *normalize-natural.rep-eq* [*simp*]

lift-definition *unit-factor-natural* :: *natural* ⇒ *natural*
is *unit-factor* :: *nat* ⇒ *nat*
⟨*proof*⟩

declare *unit-factor-natural.rep-eq* [*simp*]

lift-definition *divide-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *divide* :: *nat* ⇒ *nat* ⇒ *nat*
⟨*proof*⟩

declare *divide-natural.rep-eq* [*simp*]

lift-definition *modulo-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *modulo* :: *nat* ⇒ *nat* ⇒ *nat*
⟨*proof*⟩

declare *modulo-natural.rep-eq* [*simp*]

instance
⟨*proof*⟩

end

lift-definition *natural-of-integer* :: *integer* ⇒ *natural*
is *nat* :: *int* ⇒ *nat*
⟨*proof*⟩

lift-definition *integer-of-natural* :: *natural* ⇒ *integer*
is *of-nat* :: *nat* ⇒ *int*
⟨*proof*⟩

lemma *natural-of-integer-of-natural* [*simp*]:
natural-of-integer (*integer-of-natural n*) = *n*
⟨*proof*⟩

lemma *integer-of-natural-of-integer* [*simp*]:
integer-of-natural (*natural-of-integer k*) = *max 0 k*
⟨*proof*⟩

lemma *int-of-integer-of-natural* [*simp*]:
int-of-integer (*integer-of-natural n*) = *of-nat* (*nat-of-natural n*)
⟨*proof*⟩

lemma *integer-of-natural-of-nat* [*simp*]:
integer-of-natural (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma [*measure-function*]:
is-measure nat-of-natural
⟨*proof*⟩

65.5 Inductive representation of target language naturals

lift-definition *Suc* :: *natural* ⇒ *natural*
is *Nat.Suc*
⟨*proof*⟩

declare *Suc.rep-eq* [*simp*]

old-rep-datatype *0*::*natural Suc*
⟨*proof*⟩

lemma *natural-cases* [*case-names nat*, *cases type*: *natural*]:
  fixes m :: *natural*
  assumes $\bigwedge n.\ m = \textit{of-nat}\ n \Longrightarrow P$
  shows P
  ⟨*proof*⟩

lemma [*simp*, *code*]: *size-natural* = *nat-of-natural*
⟨*proof*⟩

lemma [*simp*, *code*]: *size* = *nat-of-natural*
⟨*proof*⟩

lemma *natural-decr* [*termination-simp*]:
  $n \neq 0 \Longrightarrow \textit{nat-of-natural}\ n - \textit{Nat.Suc}\ 0 < \textit{nat-of-natural}\ n$
  ⟨*proof*⟩

lemma *natural-zero-minus-one*: $(0::\textit{natural}) - 1 = 0$
  ⟨*proof*⟩

lemma *Suc-natural-minus-one*: $\textit{Suc}\ n - 1 = n$
  ⟨*proof*⟩

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

65.6 Code refinement for target language naturals

lift-definition *Nat* :: *integer* ⇒ *natural*
  is *nat*
  ⟨*proof*⟩

lemma [*code-post*]:
  $\textit{Nat}\ 0 = 0$
  $\textit{Nat}\ 1 = 1$
  $\textit{Nat}\ (\textit{numeral}\ k) = \textit{numeral}\ k$
  ⟨*proof*⟩

lemma [*code abstype*]:
  $\textit{Nat}\ (\textit{integer-of-natural}\ n) = n$
  ⟨*proof*⟩

lemma [*code*]:
  $\textit{natural-of-nat}\ n = \textit{natural-of-integer}\ (\textit{integer-of-nat}\ n)$
  ⟨*proof*⟩

lemma [*code abstract*]:
  $\textit{integer-of-natural}\ (\textit{natural-of-integer}\ k) = \textit{max}\ 0\ k$
  ⟨*proof*⟩

lemma [*code-abbrev*]:
natural-of-integer (*Code-Numeral.Pos k*) = *numeral k*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-natural 0 = *0*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-natural 1 = *1*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-natural (*Code-Numeral.Suc n*) = *integer-of-natural n* + *1*
⟨*proof*⟩

lemma [*code*]:
nat-of-natural = *nat-of-integer* ∘ *integer-of-natural*
⟨*proof*⟩

lemma [*code*, *code-unfold*]:
case-natural f g n = (*if n* = *0 then f else g* (*n* − *1*))
⟨*proof*⟩

declare *natural.rec* [*code del*]

lemma [*code abstract*]:
integer-of-natural (*m* + *n*) = *integer-of-natural m* + *integer-of-natural n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-natural (*m* − *n*) = *max 0* (*integer-of-natural m* − *integer-of-natural n*)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-natural (*m* ∗ *n*) = *integer-of-natural m* ∗ *integer-of-natural n*
⟨*proof*⟩

lemma [*code*]:
normalize n = *n* **for** *n* :: *natural*
⟨*proof*⟩

lemma [*code*]:
unit-factor n = *of-bool* (*n* ≠ *0*) **for** *n* :: *natural*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-natural (*m div n*) = *integer-of-natural m div integer-of-natural n*

⟨*proof*⟩

lemma [*code abstract*]:
integer-of-natural (*m mod n*) = *integer-of-natural m mod integer-of-natural n*
⟨*proof*⟩

lemma [*code*]:
HOL.equal m n ⟷ *HOL.equal* (*integer-of-natural m*) (*integer-of-natural n*)
⟨*proof*⟩

lemma [*code nbe*]: *HOL.equal n* (*n*::*natural*) ⟷ *True*
⟨*proof*⟩

lemma [*code*]: $m \leq n$ ⟷ *integer-of-natural m* ≤ *integer-of-natural n*
⟨*proof*⟩

lemma [*code*]: $m < n$ ⟷ *integer-of-natural m* < *integer-of-natural n*
⟨*proof*⟩

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

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

lifting-update *natural.lifting*
lifting-forget *natural.lifting*

code-reflect *Code-Numeral*
datatypes *natural*
functions *Code-Numeral.Suc 0* :: *natural 1* :: *natural*
plus :: *natural* ⇒ - *minus* :: *natural* ⇒ -
times :: *natural* ⇒ - *divide* :: *natural* ⇒ -
modulo :: *natural* ⇒ -
integer-of-natural natural-of-integer

end

66 Setup for Lifting/Transfer for the set type

theory *Lifting-Set*
imports *Lifting*
begin

66.1 Relator and predicator properties

lemma *rel-setD1*: ⟦ *rel-set R A B*; $x \in A$ ⟧ ⟹ ∃ $y \in B$. *R x y*
and *rel-setD2*: ⟦ *rel-set R A B*; $y \in B$ ⟧ ⟹ ∃ $x \in A$. *R x y*
⟨*proof*⟩

lemma *rel-set-conversep* [*simp*]: *rel-set* A^{-1-1} = (*rel-set* A)$^{-1-1}$
⟨*proof*⟩

lemma *rel-set-eq* [*relator-eq*]: *rel-set* (*op* =) = (*op* =)
⟨*proof*⟩

lemma *rel-set-mono*[*relator-mono*]:
assumes $A \leq B$
shows *rel-set* $A \leq$ *rel-set* B
⟨*proof*⟩

lemma *rel-set-OO*[*relator-distr*]: *rel-set* R *OO* *rel-set* S = *rel-set* (R *OO* S)
⟨*proof*⟩

lemma *Domainp-set*[*relator-domain*]:
Domainp (*rel-set* T) = (λA. *Ball* A (*Domainp* T))
⟨*proof*⟩

lemma *left-total-rel-set*[*transfer-rule*]:
left-total $A \Longrightarrow$ *left-total* (*rel-set* A)
⟨*proof*⟩

lemma *left-unique-rel-set*[*transfer-rule*]:
left-unique $A \Longrightarrow$ *left-unique* (*rel-set* A)
⟨*proof*⟩

lemma *right-total-rel-set* [*transfer-rule*]:
right-total $A \Longrightarrow$ *right-total* (*rel-set* A)
⟨*proof*⟩

lemma *right-unique-rel-set* [*transfer-rule*]:
right-unique $A \Longrightarrow$ *right-unique* (*rel-set* A)
⟨*proof*⟩

lemma *bi-total-rel-set* [*transfer-rule*]:
bi-total $A \Longrightarrow$ *bi-total* (*rel-set* A)
⟨*proof*⟩

lemma *bi-unique-rel-set* [*transfer-rule*]:
bi-unique $A \Longrightarrow$ *bi-unique* (*rel-set* A)
⟨*proof*⟩

lemma *set-relator-eq-onp* [*relator-eq-onp*]:
rel-set (*eq-onp* P) = *eq-onp* (λA. *Ball* A P)
⟨*proof*⟩

lemma *bi-unique-rel-set-lemma*:
assumes *bi-unique* R **and** *rel-set* R X Y
obtains f **where** Y = *image* f X **and** *inj-on* f X **and** $\forall x \in X$. R x (f x)

⟨*proof*⟩

66.2 Quotient theorem for the Lifting package

lemma *Quotient-set*[*quot-map*]:
assumes *Quotient R Abs Rep T*
shows *Quotient* (*rel-set R*) (*image Abs*) (*image Rep*) (*rel-set T*)
⟨*proof*⟩

66.3 Transfer rules for the Transfer package

66.3.1 Unconditional transfer rules

context includes *lifting-syntax*
begin

lemma *empty-transfer* [*transfer-rule*]: (*rel-set A*) {} {}
⟨*proof*⟩

lemma *insert-transfer* [*transfer-rule*]:
(*A* ===> *rel-set A* ===> *rel-set A*) *insert insert*
⟨*proof*⟩

lemma *union-transfer* [*transfer-rule*]:
(*rel-set A* ===> *rel-set A* ===> *rel-set A*) *union union*
⟨*proof*⟩

lemma *Union-transfer* [*transfer-rule*]:
(*rel-set* (*rel-set A*) ===> *rel-set A*) *Union Union*
⟨*proof*⟩

lemma *image-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-set A* ===> *rel-set B*) *image image*
⟨*proof*⟩

lemma *UNION-transfer* [*transfer-rule*]:
(*rel-set A* ===> (*A* ===> *rel-set B*) ===> *rel-set B*) *UNION UNION*
⟨*proof*⟩

lemma *Ball-transfer* [*transfer-rule*]:
(*rel-set A* ===> (*A* ===> *op* =) ===> *op* =) *Ball Ball*
⟨*proof*⟩

lemma *Bex-transfer* [*transfer-rule*]:
(*rel-set A* ===> (*A* ===> *op* =) ===> *op* =) *Bex Bex*
⟨*proof*⟩

lemma *Pow-transfer* [*transfer-rule*]:
(*rel-set A* ===> *rel-set* (*rel-set A*)) *Pow Pow*
⟨*proof*⟩

lemma *rel-set-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> *op* =) ===> *rel-set A* ===> *rel-set B* ===> *op* =) *rel-set rel-set*
⟨*proof*⟩

lemma *bind-transfer* [*transfer-rule*]:
(*rel-set A* ===> (*A* ===> *rel-set B*) ===> *rel-set B*) *Set.bind Set.bind*
⟨*proof*⟩

lemma *INF-parametric* [*transfer-rule*]:
(*rel-set A* ===> (*A* ===> *HOL.eq*) ===> *HOL.eq*) *INFIMUM INFIMUM*
⟨*proof*⟩

lemma *SUP-parametric* [*transfer-rule*]:
(*rel-set R* ===> (*R* ===> *HOL.eq*) ===> *HOL.eq*) *SUPREMUM SUPREMUM*
⟨*proof*⟩

66.3.2 Rules requiring bi-unique, bi-total or right-total relations

lemma *member-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *rel-set A* ===> *op* =) (*op* ∈) (*op* ∈)
⟨*proof*⟩

lemma *right-total-Collect-transfer*[*transfer-rule*]:
assumes *right-total A*
shows ((*A* ===> *op* =) ===> *rel-set A*) (λ*P*. *Collect* (λ*x*. *P x* ∧ *Domainp A x*)) *Collect*
⟨*proof*⟩

lemma *Collect-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows ((*A* ===> *op* =) ===> *rel-set A*) *Collect Collect*
⟨*proof*⟩

lemma *inter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-set A* ===> *rel-set A* ===> *rel-set A*) *inter inter*
⟨*proof*⟩

lemma *Diff-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-set A* ===> *rel-set A* ===> *rel-set A*) (*op* −) (*op* −)
⟨*proof*⟩

lemma *subset-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*rel-set A* ===> *rel-set A* ===> *op* =) (*op* ⊆) (*op* ⊆)

⟨*proof*⟩

declare *right-total-UNIV-transfer*[*transfer-rule*]

lemma *UNIV-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows (*rel-set A*) *UNIV UNIV*
⟨*proof*⟩

lemma *right-total-Compl-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set A* ===> *rel-set A*) (λS. *uminus S* ∩ *Collect* (*Domainp A*)) *uminus*
⟨*proof*⟩

lemma *Compl-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set A* ===> *rel-set A*) *uminus uminus*
⟨*proof*⟩

lemma *right-total-Inter-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λS. $\bigcap$ *S* ∩ *Collect* (*Domainp A*)) *Inter*
⟨*proof*⟩

lemma *Inter-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) *Inter Inter*
⟨*proof*⟩

lemma *filter-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ((*A* ===> *op*=) ===> *rel-set A* ===> *rel-set A*) *Set.filter Set.filter*
⟨*proof*⟩

lemma *finite-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set A* ===> *op* =) *finite finite*
⟨*proof*⟩

lemma *card-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set A* ===> *op* =) *card card*
⟨*proof*⟩

lemma *vimage-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique B*
shows ((*A* ===> *B*) ===> *rel-set B* ===> *rel-set A*) *vimage vimage*
⟨*proof*⟩

lemma *Image-parametric* [*transfer-rule*]:
  assumes *bi-unique A*
  shows (*rel-set* (*rel-prod A B*) ===> *rel-set A* ===> *rel-set B*) *op* ‘‘ *op* ‘‘
  ⟨*proof*⟩

end

lemma (**in** *comm-monoid-set*) *F-parametric* [*transfer-rule*]:
  fixes $A :: 'b \Rightarrow 'c \Rightarrow bool$
  assumes *bi-unique A*
  shows *rel-fun* (*rel-fun A* (*op* =)) (*rel-fun* (*rel-set A*) (*op* =)) *F F*
⟨*proof*⟩

lemmas *sum-parametric* = *sum.F-parametric*
lemmas *prod-parametric* = *prod.F-parametric*

lemma *rel-set-UNION*:
  assumes [*transfer-rule*]: *rel-set Q A B rel-fun Q* (*rel-set R*) *f g*
  shows *rel-set R* (*UNION A f*) (*UNION B g*)
  ⟨*proof*⟩

end

67 The datatype of finite lists

theory *List*
imports *Sledgehammer Code-Numeral Lifting-Set*
begin

datatype (*set*: *'a*) *list* =
    Nil ([])
  | *Cons* (*hd*: *'a*) (*tl*: *'a list*) (**infixr** # *65*)
for
  map: *map*
  rel: *list-all2*
  pred: *list-all*
where
  tl [] = []

datatype-compat *list*

lemma [*case-names Nil Cons*, *cases type*: *list*]:
  — for backward compatibility – names of variables differ
  $(y = [] \Longrightarrow P) \Longrightarrow (\bigwedge a\ list.\ y = a \# list \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma [*case-names Nil Cons*, *induct type*: *list*]:
  — for backward compatibility – names of variables differ
  $P\ [] \Longrightarrow (\bigwedge a\ list.\ P\ list \Longrightarrow P\ (a \# list)) \Longrightarrow P\ list$

⟨*proof*⟩

Compatibility:

⟨*ML*⟩

lemmas *inducts* = *list.induct*
lemmas *recs* = *list.rec*
lemmas *cases* = *list.case*

⟨*ML*⟩

lemmas *set-simps* = *list.set*

syntax
— list Enumeration
-list :: *args* => *'a list* ([(-)])

translations
[*x*, *xs*] == *x*#[*xs*]
[*x*] == *x*#[]

67.1 Basic list processing functions

primrec (*nonexhaustive*) *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*)

lemma *set-rec*: *set xs* = *rec-list* {} (λ*x* -. *insert x*) *xs*
⟨*proof*⟩

definition *coset* :: *'a list* ⇒ *'a set* **where**
[*simp*]: *coset xs* = − *set 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*)

Special syntax for filter:

syntax (*ASCII*)
  -filter :: [*pttrn*, *'a list*, *bool*] => *'a list* ((*1*[-<−-./ -]))
syntax
  -filter :: [*pttrn*, *'a list*, *bool*] => *'a list* ((*1*[-←- ./ -]))
translations
  [*x*<−*xs* . *P*] ⇌ *CONST filter* (λ*x*. *P*) *xs*

primrec *fold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a list* ⇒ *'b* ⇒ *'b* **where**
fold-Nil: *fold f* [] = *id* |
fold-Cons: *fold f* (*x* # *xs*) = *fold f xs* ∘ *f x*

primrec *foldr* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a list* ⇒ *'b* ⇒ *'b* **where**
foldr-Nil: *foldr f* [] = *id* |
foldr-Cons: *foldr f* (*x* # *xs*) = *f x* ∘ *foldr f xs*

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 *concat*:: *'a list list* ⇒ *'a list* **where**
concat [] = [] |
concat (*x* # *xs*) = *x* @ *concat xs*

primrec *drop*:: *nat* ⇒ *'a list* ⇒ *'a list* **where**
drop-Nil: *drop n* [] = [] |
drop-Cons: *drop n* (*x* # *xs*) = (*case n of 0* ⇒ *x* # *xs* | *Suc m* ⇒ *drop m xs*)
  — Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec *take*:: *nat* ⇒ *'a list* ⇒ *'a list* **where**
take-Nil:*take n* [] = [] |
take-Cons: *take n* (*x* # *xs*) = (*case n of 0* ⇒ [] | *Suc m* ⇒ *x* # *take m xs*)
  — Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec (*nonexhaustive*) *nth* :: *'a list* => *nat* => *'a* (**infixl** ! *100*) **where**
nth-Cons: (*x* # *xs*) ! *n* = (*case n of 0* ⇒ *x* | *Suc k* ⇒ *xs* ! *k*)
  — Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec *list-update* :: *'a list* ⇒ *nat* ⇒ *'a* ⇒ *'a list* **where**
list-update [] *i v* = [] |
list-update (*x* # *xs*) *i v* =
  (*case i of 0* ⇒ *v* # *xs* | *Suc j* ⇒ *x* # *list-update xs j v*)

nonterminal *lupdbinds* **and** *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 *product* :: '*a list* ⇒ '*b list* ⇒ ('*a* × '*b*) *list* **where**
product [] - = [] |
product (*x*#*xs*) *ys* = *map* (*Pair x*) *ys* @ *product xs ys*

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

primrec *product-lists* :: '*a list list* ⇒ '*a list list* **where**
product-lists [] = [[]] |
product-lists (*xs* # *xss*) = *concat* (*map* (λ*x*. *map* (*Cons x*) (*product-lists xss*)) *xs*)

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* [])

definition *insert* :: '*a* ⇒ '*a list* ⇒ '*a list* **where**
insert x xs = (*if x* ∈ *set xs then xs else x* # *xs*)

definition *union* :: '*a list* ⇒ '*a list* ⇒ '*a list* **where**
union = *fold insert*

hide-const (**open**) *insert union*
hide-fact (**open**) *insert-def union-def*

primrec *find* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a option* **where**
find - [] = *None* |
find P (*x*#*xs*) = (*if P x then Some x else find P xs*)

In the context of multisets, *count-list* is equivalent to *count* ∘ *mset* and it it advisable to use the latter.

primrec *count-list* :: $'a$ *list* ⇒ $'a$ ⇒ *nat* **where**
count-list [] *y* = *0* |
count-list (*x#xs*) *y* = (*if x=y then count-list xs y* + *1 else count-list xs y*)

definition
extract :: ($'a$ ⇒ *bool*) ⇒ $'a$ *list* ⇒ ($'a$ *list* ∗ $'a$ ∗ $'a$ *list*) *option*
where *extract P xs* =
(*case dropWhile* (*Not o P*) *xs of*
[] ⇒ *None* |
y#ys ⇒ *Some*(*takeWhile* (*Not o P*) *xs*, *y*, *ys*))

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

primrec *those* :: $'a$ *option list* ⇒ $'a$ *list option*
where
those [] = *Some* [] |
those (*x* # *xs*) = (*case x of*
None ⇒ *None*
| *Some y* ⇒ *map-option* (*Cons y*) (*those xs*))

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 *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*)

fun *remdups-adj* :: $'a$ *list* ⇒ $'a$ *list* **where**
remdups-adj [] = [] |
remdups-adj [*x*] = [*x*] |
remdups-adj (*x* # *y* # *xs*) = (*if x* = *y then remdups-adj* (*x* # *xs*) *else x* # *remdups-adj* (*y* # *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 *enumerate* :: *nat* ⇒ *'a list* ⇒ (*nat* × *'a*) *list* **where**
enumerate-eq-zip: *enumerate n xs* = *zip* [*n*..<*n* + *length xs*] *xs*

primrec *rotate1* :: *'a list* ⇒ *'a list* **where**
rotate1 [] = [] |
rotate1 (*x* # *xs*) = *xs* @ [*x*]

definition *rotate* :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
rotate n = *rotate1* ^^ *n*

definition *nths* :: *'a list* => *nat set* => *'a list* **where**
nths xs A = *map fst* (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*0*..<*size xs*]))

primrec *subseqs* :: *'a list* ⇒ *'a list list* **where**
subseqs [] = [[]] |
subseqs (*x*#*xs*) = (*let xss* = *subseqs xs in map* (*Cons x*) *xss* @ *xss*)

primrec *n-lists* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
n-lists 0 xs = [[]] |
n-lists (*Suc n*) *xs* = *concat* (*map* (λ*ys*. *map* (λ*y*. *y* # *ys*) *xs*) (*n-lists n xs*))

hide-const (**open**) *n-lists*

fun *splice* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
splice [] *ys* = *ys* |
splice xs [] = *xs* |
splice (*x*#*xs*) (*y*#*ys*) = *x* # *y* # *splice xs ys*

function *shuffle* **where**
  shuffle [] *ys* = {*ys*}
| *shuffle xs* [] = {*xs*}
| *shuffle* (*x* # *xs*) (*y* # *ys*) = *op* # *x* ' *shuffle xs* (*y* # *ys*) ∪ *op* # *y* ' *shuffle* (*x* # *xs*) *ys*
  ⟨*proof*⟩
termination ⟨*proof*⟩

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.

A sorted predicate w.r.t. a relation:

fun *sorted-wrt* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
sorted-wrt P [] = *True* |

$[a,\ b]\ @\ [c,\ d] = [a,\ b,\ c,\ d]$
$\mathit{length}\ [a,\ b,\ c] = \mathit{3}$
$\mathit{set}\ [a,\ b,\ c] = \{a,\ b,\ c\}$
$\mathit{map}\ f\ [a,\ b,\ c] = [f\ a,\ f\ b,\ f\ c]$
$\mathit{rev}\ [a,\ b,\ c] = [c,\ b,\ a]$
$\mathit{hd}\ [a,\ b,\ c,\ d] = a$
$\mathit{tl}\ [a,\ b,\ c,\ d] = [b,\ c,\ d]$
$\mathit{last}\ [a,\ b,\ c,\ d] = d$
$\mathit{butlast}\ [a,\ b,\ c,\ d] = [a,\ b,\ c]$
$\mathit{filter}\ (\lambda n{::}\mathit{nat}.\ n<\mathit{2})\ [\mathit{0},\mathit{2},\mathit{1}] = [\mathit{0},\mathit{1}]$
$\mathit{concat}\ [[a,\ b],\ [c,\ d,\ e],\ [],\ [f]] = [a,\ b,\ c,\ d,\ e,\ f]$
$\mathit{fold}\ f\ [a,\ b,\ c]\ x = f\ c\ (f\ b\ (f\ a\ x))$
$\mathit{foldr}\ f\ [a,\ b,\ c]\ x = f\ a\ (f\ b\ (f\ c\ x))$
$\mathit{foldl}\ f\ x\ [a,\ b,\ c] = f\ (f\ (f\ x\ a)\ b)\ c$
$\mathit{zip}\ [a,\ b,\ c]\ [x,\ y,\ z] = [(a,\ x),\ (b,\ y),\ (c,\ z)]$
$\mathit{zip}\ [a,\ b]\ [x,\ y,\ z] = [(a,\ x),\ (b,\ y)]$
$\mathit{enumerate}\ \mathit{3}\ [a,\ b,\ c] = [(\mathit{3},\ a),\ (\mathit{4},\ b),\ (\mathit{5},\ c)]$
$\mathit{List.product}\ [a,\ b]\ [c,\ d] = [(a,\ c),\ (a,\ d),\ (b,\ c),\ (b,\ d)]$
$\mathit{product\text{-}lists}\ [[a,\ b],\ [c],\ [d,\ e]] = [[a,\ c,\ d],\ [a,\ c,\ e],\ [b,\ c,\ d],\ [b,\ c,\ e]]$
$\mathit{splice}\ [a,\ b,\ c]\ [x,\ y,\ z] = [a,\ x,\ b,\ y,\ c,\ z]$
$\mathit{splice}\ [a,\ b,\ c,\ d]\ [x,\ y] = [a,\ x,\ b,\ y,\ c,\ d]$
$\mathit{shuffle}\ [a,\ b]\ [c,\ d] = \{[a,\ b,\ c,\ d],\ [a,\ c,\ b,\ d],\ [a,\ c,\ d,\ b],\ [c,\ a,\ b,\ d],\ [c,\ a,\ d,\ b],\ [c,\ d,\ a,\ b]\}$
$\mathit{take}\ \mathit{2}\ [a,\ b,\ c,\ d] = [a,\ b]$
$\mathit{take}\ \mathit{6}\ [a,\ b,\ c,\ d] = [a,\ b,\ c,\ d]$
$\mathit{drop}\ \mathit{2}\ [a,\ b,\ c,\ d] = [c,\ d]$
$\mathit{drop}\ \mathit{6}\ [a,\ b,\ c,\ d] = []$
$\mathit{takeWhile}\ (\lambda n.\ n < \mathit{3})\ [\mathit{1},\ \mathit{2},\ \mathit{3},\ \mathit{0}] = [\mathit{1},\ \mathit{2}]$
$\mathit{dropWhile}\ (\lambda n.\ n < \mathit{3})\ [\mathit{1},\ \mathit{2},\ \mathit{3},\ \mathit{0}] = [\mathit{3},\ \mathit{0}]$
$\mathit{distinct}\ [\mathit{2},\ \mathit{0},\ \mathit{1}]$
$\mathit{remdups}\ [\mathit{2},\ \mathit{0},\ \mathit{2},\ \mathit{1},\ \mathit{2}] = [\mathit{0},\ \mathit{1},\ \mathit{2}]$
$\mathit{remdups\text{-}adj}\ [\mathit{2},\ \mathit{2},\ \mathit{3},\ \mathit{1},\ \mathit{1},\ \mathit{2},\ \mathit{1}] = [\mathit{2},\ \mathit{3},\ \mathit{1},\ \mathit{2},\ \mathit{1}]$
$\mathit{List.insert}\ \mathit{2}\ [\mathit{0},\ \mathit{1},\ \mathit{2}] = [\mathit{0},\ \mathit{1},\ \mathit{2}]$
$\mathit{List.insert}\ \mathit{3}\ [\mathit{0},\ \mathit{1},\ \mathit{2}] = [\mathit{3},\ \mathit{0},\ \mathit{1},\ \mathit{2}]$
$\mathit{List.union}\ [\mathit{2},\ \mathit{3},\ \mathit{4}]\ [\mathit{0},\ \mathit{1},\ \mathit{2}] = [\mathit{4},\ \mathit{3},\ \mathit{0},\ \mathit{1},\ \mathit{2}]$
$\mathit{find}\ (\mathit{op} < \mathit{0})\ [\mathit{0},\ \mathit{0}] = \mathit{None}$
$\mathit{find}\ (\mathit{op} < \mathit{0})\ [\mathit{0},\ \mathit{1},\ \mathit{0},\ \mathit{2}] = \mathit{Some}\ \mathit{1}$
$\mathit{count\text{-}list}\ [\mathit{0},\ \mathit{1},\ \mathit{0},\ \mathit{2}]\ \mathit{0} = \mathit{2}$
$\mathit{List.extract}\ (\mathit{op} < \mathit{0})\ [\mathit{0},\ \mathit{0}] = \mathit{None}$
$\mathit{List.extract}\ (\mathit{op} < \mathit{0})\ [\mathit{0},\ \mathit{1},\ \mathit{0},\ \mathit{2}] = \mathit{Some}\ ([\mathit{0}],\ \mathit{1},\ [\mathit{0},\ \mathit{2}])$
$\mathit{remove1}\ \mathit{2}\ [\mathit{2},\ \mathit{0},\ \mathit{2},\ \mathit{1},\ \mathit{2}] = [\mathit{0},\ \mathit{2},\ \mathit{1},\ \mathit{2}]$
$\mathit{removeAll}\ \mathit{2}\ [\mathit{2},\ \mathit{0},\ \mathit{2},\ \mathit{1},\ \mathit{2}] = [\mathit{0},\ \mathit{1}]$
$[a,\ b,\ c,\ d]\ !\ \mathit{2} = c$
$[a,\ b,\ c,\ d][\mathit{2} := x] = [a,\ b,\ x,\ d]$
$\mathit{nths}\ [a,\ b,\ c,\ d,\ e]\ \{\mathit{0},\ \mathit{2},\ \mathit{3}\} = [a,\ c,\ d]$
$\mathit{subseqs}\ [a,\ b] = [[a,\ b],\ [a],\ [b],\ []]$
$\mathit{List.n\text{-}lists}\ \mathit{2}\ [a,\ b,\ c] = [[a,\ a],\ [b,\ a],\ [c,\ a],\ [a,\ b],\ [b,\ b],\ [c,\ b],\ [a,\ c],\ [b,\ c],\ [c,\ c]]$
$\mathit{rotate1}\ [a,\ b,\ c,\ d] = [b,\ c,\ d,\ a]$
$\mathit{rotate}\ \mathit{3}\ [a,\ b,\ c,\ d] = [d,\ a,\ b,\ c]$
$\mathit{replicate}\ \mathit{4}\ a = [a,\ a,\ a,\ a]$
$[\mathit{2}..<\mathit{5}] = [\mathit{2},\ \mathit{3},\ \mathit{4}]$

Figure 1: Characteristic examples

sorted-wrt P $[x]$ = *True* |
sorted-wrt P $(x \# y \# zs) = (P\ x\ y \wedge$ *sorted-wrt* $P\ (y \# zs))$

A class-based sorted predicate:

context *linorder*
begin

inductive *sorted* :: $'a$ *list* $\Rightarrow$ *bool* **where**
  Nil [*iff*]: *sorted* []
| *Cons*: $\forall y{\in}set\ xs.\ x \leq y \Longrightarrow sorted\ xs \Longrightarrow sorted\ (x \# xs)$

lemma *sorted-single* [*iff*]: *sorted* $[x]$
⟨*proof*⟩

lemma *sorted-many*: $x \leq y \Longrightarrow sorted\ (y \# zs) \Longrightarrow sorted\ (x \# y \# zs)$
⟨*proof*⟩

lemma *sorted-many-eq* [*simp*, *code*]:
  $sorted\ (x \# y \# zs) \longleftrightarrow x \leq y \wedge sorted\ (y \# zs)$
⟨*proof*⟩

lemma [*code*]:
  $sorted\ [] \longleftrightarrow True$
  $sorted\ [x] \longleftrightarrow True$
⟨*proof*⟩

primrec *insort-key* :: $('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
insort-key $f\ x\ [] = [x]$ |
insort-key $f\ x\ (y \# ys) =$
  $(if\ f\ x \leq f\ y\ then\ (x \# y \# ys)\ else\ y \# (insort\text{-}key\ f\ x\ ys))$

definition *sort-key* :: $('b \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
sort-key $f\ xs = foldr\ (insort\text{-}key\ f)\ xs\ []$

definition *insort-insert-key* :: $('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
insort-insert-key $f\ x\ xs =$
  $(if\ f\ x \in f\ `\ set\ xs\ then\ xs\ else\ insort\text{-}key\ f\ x\ xs)$

abbreviation $sort \equiv sort\text{-}key\ (\lambda x.\ x)$
abbreviation $insort \equiv insort\text{-}key\ (\lambda x.\ x)$
abbreviation $insort\text{-}insert \equiv insort\text{-}insert\text{-}key\ (\lambda x.\ x)$

end

67.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: $[(x,y).\ x \leftarrow xs,\ y \leftarrow ys,\ x \neq y]$, 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): $[e.\ x \leftarrow xs, \ldots]$ rather than `[e| x <- xs, ...]`.

The qualifiers after the dot are

generators $p \leftarrow xs$, 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 $[e.\ x \leftarrow xs]$ is optmized to $map\ (\lambda x.\ e)\ xs$.

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.

nonterminal *lc-qual* **and** *lc-quals*

syntax
-listcompr :: $'a \Rightarrow$ *lc-qual* $\Rightarrow$ *lc-quals* $\Rightarrow$ $'a$ *list* ([- . --)
-lc-gen :: $'a \Rightarrow 'a$ *list* $\Rightarrow$ *lc-qual* (- $\leftarrow$ -)
-lc-test :: *bool* $\Rightarrow$ *lc-qual* (-)

-lc-end :: *lc-quals* (])
-lc-quals :: *lc-qual* $\Rightarrow$ *lc-quals* $\Rightarrow$ *lc-quals* (, --)
-lc-abs :: $'a$ => $'b$ *list* => $'b$ *list*

syntax (*ASCII*)
-lc-gen :: $'a \Rightarrow 'a$ *list* $\Rightarrow$ *lc-qual* (- <− -)

⟨*ML*⟩

code-datatype *set coset*
hide-const (**open**) *coset*

67.1.2 [] and *op*

lemma *not-Cons-self* [*simp*]:
$xs \neq x \# xs$
⟨*proof*⟩

lemma *not-Cons-self2* [*simp*]: $x \# xs \neq xs$
⟨*proof*⟩

lemma *neq-Nil-conv*: $(xs \neq []) = (\exists\, y\ ys.\ xs = y \# ys)$

$\langle proof \rangle$

lemma *tl-Nil*: $tl\ xs = [] \longleftrightarrow xs = [] \lor (EX\ x.\ xs = [x])$
$\langle proof \rangle$

lemma *Nil-tl*: $[] = tl\ xs \longleftrightarrow xs = [] \lor (EX\ x.\ xs = [x])$
$\langle proof \rangle$

lemma *length-induct*:
$(\bigwedge xs.\ \forall ys.\ length\ ys < length\ xs \longrightarrow P\ ys \Longrightarrow P\ xs) \Longrightarrow P\ xs$
$\langle proof \rangle$

lemma *list-nonempty-induct* [*consumes 1*, *case-names single cons*]:
assumes $xs \neq []$
assumes *single*: $\bigwedge x.\ P\ [x]$
assumes *cons*: $\bigwedge x\ xs.\ xs \neq [] \Longrightarrow P\ xs \Longrightarrow P\ (x\ \#\ xs)$
shows $P\ xs$
$\langle proof \rangle$

lemma *inj-split-Cons*: $inj\text{-}on\ (\lambda(xs,\ n).\ n\#xs)\ X$
$\langle proof \rangle$

lemma *inj-on-Cons1* [*simp*]: $inj\text{-}on\ (op\ \#\ x)\ A$
$\langle proof \rangle$

67.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: $length\ (xs\ @\ ys) = length\ xs + length\ ys$
$\langle proof \rangle$

lemma *length-map* [*simp*]: $length\ (map\ f\ xs) = length\ xs$
$\langle proof \rangle$

lemma *length-rev* [*simp*]: $length\ (rev\ xs) = length\ xs$
$\langle proof \rangle$

lemma *length-tl* [*simp*]: $length\ (tl\ xs) = length\ xs - 1$
$\langle proof \rangle$

lemma *length-0-conv* [*iff*]: $(length\ xs = 0) = (xs = [])$
$\langle proof \rangle$

lemma *length-greater-0-conv* [*iff*]: $(0 < length\ xs) = (xs \neq [])$
$\langle proof \rangle$

lemma *length-pos-if-in-set*: $x : set\ xs \Longrightarrow length\ xs > 0$
$\langle proof \rangle$

lemma *length-Suc-conv*:
$(length\ xs = Suc\ n) = (\exists\, y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
$\langle proof \rangle$

lemma *Suc-length-conv*:
$(Suc\ n = length\ xs) = (\exists\, y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
$\langle proof \rangle$

lemma *impossible-Cons*: $length\ xs <= length\ ys ==> xs = x\ \#\ ys = False$
$\langle proof \rangle$

lemma *list-induct2* [*consumes 1*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow P\ []\ [] \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (x\#xs)\ (y\#ys))$
$\Longrightarrow P\ xs\ ys$
$\langle proof \rangle$

lemma *list-induct3* [*consumes 2*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow P\ []\ []\ [] \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys\ z\ zs.\ length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow P\ xs\ ys\ zs$
$\Longrightarrow P\ (x\#xs)\ (y\#ys)\ (z\#zs))$
$\Longrightarrow P\ xs\ ys\ zs$
$\langle proof \rangle$

lemma *list-induct4* [*consumes 3*, *case-names Nil Cons*]:
$length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow length\ zs = length\ ws \Longrightarrow$
$P\ []\ []\ []\ [] \Longrightarrow (\bigwedge x\ xs\ y\ ys\ z\ zs\ w\ ws.\ length\ xs = length\ ys \Longrightarrow$
$length\ ys = length\ zs \Longrightarrow length\ zs = length\ ws \Longrightarrow P\ xs\ ys\ zs\ ws \Longrightarrow$
$P\ (x\#xs)\ (y\#ys)\ (z\#zs)\ (w\#ws)) \Longrightarrow P\ xs\ ys\ zs\ ws$
$\langle proof \rangle$

lemma *list-induct2'*:
$[\![\ P\ []\ [];$
$\bigwedge x\ xs.\ P\ (x\#xs)\ [];$
$\bigwedge y\ ys.\ P\ []\ (y\#ys);$
$\bigwedge x\ xs\ y\ ys.\ P\ xs\ ys \Longrightarrow P\ (x\#xs)\ (y\#ys)\ ]\!]$
$\Longrightarrow P\ xs\ ys$
$\langle proof \rangle$

lemma *list-all2-iff*:
$list\text{-}all2\ P\ xs\ ys \longleftrightarrow length\ xs = length\ ys \wedge (\forall (x,\ y) \in set\ (zip\ xs\ ys).\ P\ x\ y)$
$\langle proof \rangle$

lemma *neq-if-length-neq*: $length\ xs \neq length\ ys \Longrightarrow (xs = ys) == False$
$\langle proof \rangle$

$\langle ML \rangle$

67.1.4 @ – append

global-interpretation *append*: *monoid append Nil*
⟨*proof*⟩

lemma *append-assoc* [*simp*]: $(xs\ @\ ys)\ @\ zs = xs\ @\ (ys\ @\ zs)$
⟨*proof*⟩

lemma *append-Nil2*: $xs\ @\ [] = xs$
⟨*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*]:
$length\ xs = length\ ys \vee length\ us = length\ vs$
$==> (xs@us = ys@vs) = (xs{=}ys \wedge us{=}vs)$
⟨*proof*⟩

lemma *append-eq-append-conv2*: $(xs\ @\ ys = zs\ @\ ts) =$
$(EX\ us.\ xs = zs\ @\ us\ \&\ us\ @\ ys = ts \mid xs\ @\ us = zs\ \&\ ys = us@\ ts)$
⟨*proof*⟩

lemma *same-append-eq* [*iff*, *induct-simp*]: $(xs\ @\ ys = xs\ @\ zs) = (ys = zs)$
⟨*proof*⟩

lemma *append1-eq-conv* [*iff*]: $(xs\ @\ [x] = ys\ @\ [y]) = (xs = ys \wedge x = y)$
⟨*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*: $xs \neq [] ==> hd\ xs\ \#\ tl\ xs = xs$
⟨*proof*⟩

lemma *hd-append*: $hd\ (xs\ @\ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
⟨*proof*⟩

lemma *hd-append2* [*simp*]: $xs \neq [] ==> hd\ (xs\ @\ ys) = hd\ xs$
⟨*proof*⟩

lemma *tl-append*: $tl\ (xs\ @\ ys) = (case\ xs\ of\ [] => tl\ ys \mid z\#zs => zs\ @\ ys)$
⟨*proof*⟩

lemma *tl-append2* [*simp*]: $xs \neq [] ==> tl\ (xs\ @\ ys) = tl\ xs\ @\ ys$
⟨*proof*⟩

lemma *Cons-eq-append-conv*: $x\#xs = ys@zs =$
$(ys = []\ \&\ x\#xs = zs \mid (EX\ ys'.\ x\#ys' = ys\ \&\ xs = ys'@zs))$
⟨*proof*⟩

lemma *append-eq-Cons-conv*: $(ys@zs = x\#xs) =$
$(ys = []\ \&\ zs = x\#xs \mid (EX\ ys'.\ ys = x\#ys'\ \&\ ys'@zs = xs))$
⟨*proof*⟩

lemma *longest-common-prefix*:
$\exists\, ps\ xs'\ ys'.\ xs = ps\ @\ xs' \wedge ys = ps\ @\ ys'$
$\wedge\ (xs' = [] \vee ys' = [] \vee hd\ xs' \neq hd\ ys')$
⟨*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*⟩

67.1.5 *map*

lemma *hd-map*: $xs \neq [] \Longrightarrow hd\ (map\ f\ xs) = f\ (hd\ xs)$
⟨*proof*⟩

lemma *map-tl*: $map\ f\ (tl\ xs) = tl\ (map\ f\ xs)$

⟨*proof*⟩

lemma *map-ext*: $(!!x.\ x : set\ xs \dashrightarrow f\ x = g\ x) ==> map\ f\ xs = map\ g\ xs$
⟨*proof*⟩

lemma *map-ident* [*simp*]: $map\ (\lambda x.\ x) = (\lambda xs.\ xs)$
⟨*proof*⟩

lemma *map-append* [*simp*]: $map\ f\ (xs\ @\ ys) = map\ f\ xs\ @\ map\ f\ ys$
⟨*proof*⟩

lemma *map-map* [*simp*]: $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$
⟨*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*]:
$xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x) \Longrightarrow map\ f\ xs = map\ g\ ys$
⟨*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*:
$(map\ f\ xs = y\#ys) = (\exists\ z\ zs.\ xs = z\#zs \wedge f\ z = y \wedge map\ f\ zs = ys)$
⟨*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) $\Longrightarrow$ *(map f xs = map f ys) = (xs = ys)*
⟨*proof*⟩

lemma *map-injective*:
map f xs = map f ys ==> *inj f* ==> *xs = ys*
⟨*proof*⟩

lemma *inj-map-eq-map*[*simp*]: *inj f* $\Longrightarrow$ *(map f xs = map f ys) = (xs = ys)*
⟨*proof*⟩

lemma *inj-mapI*: *inj f* ==> *inj (map f)*
⟨*proof*⟩

lemma *inj-mapD*: *inj (map f)* ==> *inj f*
⟨*proof*⟩

lemma *inj-map*[*iff*]: *inj (map f) = inj f*
⟨*proof*⟩

lemma *inj-on-mapI*: *inj-on f* $(\bigcup(set\ `\ A))$ $\Longrightarrow$ *inj-on (map f) A*
⟨*proof*⟩

lemma *map-idI*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = x) \Longrightarrow map\ f\ xs = xs$
⟨*proof*⟩

lemma *map-fun-upd* [*simp*]: $y \notin set\ xs \Longrightarrow map\ (f(y{:=}v))\ xs = map\ f\ xs$
⟨*proof*⟩

lemma *map-fst-zip*[*simp*]:
length xs = length ys $\Longrightarrow$ *map fst (zip xs ys) = xs*
⟨*proof*⟩

lemma *map-snd-zip*[*simp*]:
length xs = length ys $\Longrightarrow$ *map snd (zip xs ys) = ys*
⟨*proof*⟩

functor *map*: *map*
⟨*proof*⟩

declare *map.id* [*simp*]

67.1.6 *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-nonempty-induct* [*consumes 1*, *case-names single snoc*]:
  assumes $xs \neq []$
  and *single*: $\bigwedge x.\ P\ [x]$
  and *snoc'*: $\bigwedge x\ xs.\ xs \neq [] \Longrightarrow P\ xs \Longrightarrow P\ (xs@[x])$
  shows *P xs*
⟨*proof*⟩

lemma *rev-eq-Cons-iff* [*iff*]: $(rev\ xs = y\#ys) = (xs = rev\ ys\ @\ [y])$
⟨*proof*⟩

67.1.7 *set*

declare *list.set*[*code-post*] — pretty output

lemma *finite-set* [*iff*]: *finite* (*set xs*)
⟨*proof*⟩

lemma *set-append* [*simp*]: $set\ (xs\ @\ ys) = (set\ xs \cup set\ ys)$
⟨*proof*⟩

lemma *hd-in-set*[*simp*]: $xs \neq [] \Longrightarrow hd\ xs : set\ xs$
⟨*proof*⟩

lemma *set-subset-Cons*: $set\ xs \subseteq set\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *set-ConsD*: $y \in set\ (x\ \#\ xs) \Longrightarrow y{=}x \vee y \in set\ xs$
⟨*proof*⟩

lemma *set-empty* [*iff*]: $(set\ xs = \{\}) = (xs = [])$
⟨*proof*⟩

lemma *set-empty2*[*iff*]: $(\{\} = set\ xs) = (xs = [])$
⟨*proof*⟩

lemma *set-rev* [*simp*]: $set\ (rev\ xs) = set\ xs$
⟨*proof*⟩

lemma *set-map* [*simp*]: $set\ (map\ f\ xs) = f\text{`}(set\ xs)$
⟨*proof*⟩

lemma *set-filter* [*simp*]: $set\ (filter\ P\ xs) = \{x.\ x : set\ xs \wedge P\ x\}$
⟨*proof*⟩

lemma *set-upt* [*simp*]: $set[i..<j] = \{i..<j\}$
⟨*proof*⟩

lemma *split-list*: $x : set\ xs \Longrightarrow \exists ys\ zs.\ xs = ys\ @\ x\ \#\ zs$
⟨*proof*⟩

lemma *in-set-conv-decomp*: $x \in set\ xs \longleftrightarrow (\exists ys\ zs.\ xs = ys\ @\ x\ \#\ zs)$
⟨*proof*⟩

lemma *split-list-first*: $x : set\ xs \Longrightarrow \exists ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ ys$
⟨*proof*⟩

lemma *in-set-conv-decomp-first*:
$(x : set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ ys)$
⟨*proof*⟩

lemma *split-list-last*: $x \in set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ zs$
⟨*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)* $\leq$ *length xs*
⟨*proof*⟩

lemma *set-minus-filter-out*:
set xs $-$ $\{y\}$ = *set (filter* $(\lambda x.\ \neg\ (x = y))$ *xs)*
⟨*proof*⟩

lemma *append-Cons-eq-iff*:
$[\![\ x \notin set\ xs;\ x \notin set\ ys\ ]\!] \Longrightarrow$
$xs\ @\ x\ \#\ ys = xs'\ @\ x\ \#\ ys' \longleftrightarrow (xs = xs' \wedge ys = ys')$
⟨*proof*⟩

67.1.8 *filter*

lemma *filter-append* [*simp*]: *filter P (xs @ ys) = filter P xs @ filter P ys*
⟨*proof*⟩

lemma *rev-filter*: *rev (filter P xs) = filter P (rev xs)*
⟨*proof*⟩

lemma *filter-filter* [*simp*]: *filter P (filter Q xs) = filter* $(\lambda x.\ Q\ x \wedge P\ x)$ *xs*
⟨*proof*⟩

lemma *length-filter-le* [*simp*]: *length (filter P xs)* $\leq$ *length xs*
⟨*proof*⟩

lemma *sum-length-filter-compl*:
length(filter P xs) + length(filter (%x. ~P x) xs) = length xs
⟨*proof*⟩

lemma *filter-True* [*simp*]: $\forall x \in set\ xs.\ P\ x$ *==> filter P xs = xs*
⟨*proof*⟩

lemma *filter-False* [*simp*]: $\forall x \in set\ xs.\ \neg\ P\ x$ *==> filter P xs = []*
⟨*proof*⟩

lemma *filter-empty-conv*: *(filter P xs = [])* = $(\forall x{\in}set\ xs.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *filter-id-conv*: *(filter P xs = xs)* = $(\forall x{\in}set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *filter-map*: *filter P (map f xs) = map f (filter (P o f) xs)*
⟨*proof*⟩

lemma *length-filter-map*[*simp*]:
$length\ (filter\ P\ (map\ f\ xs)) = length(filter\ (P\ o\ f)\ xs)$
⟨*proof*⟩

lemma *filter-is-subset* [*simp*]: $set\ (filter\ P\ xs) \leq set\ xs$
⟨*proof*⟩

lemma *length-filter-less*:
$[\![\ x : set\ xs;\ {\sim}\ P\ x\ ]\!] \Longrightarrow length(filter\ P\ xs) < length\ xs$
⟨*proof*⟩

lemma *length-filter-conv-card*:
$length(filter\ p\ xs) = card\{i.\ i < length\ xs\ \&\ p(xs!i)\}$
⟨*proof*⟩

lemma *Cons-eq-filterD*:
$x\#xs = filter\ P\ ys \Longrightarrow$
$\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u\in set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
(**is** - $\Longrightarrow \exists\, us\ vs.\ ?P\ ys\ us\ vs$)
⟨*proof*⟩

lemma *filter-eq-ConsD*:
$filter\ P\ ys = x\#xs \Longrightarrow$
$\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u\in set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
⟨*proof*⟩

lemma *filter-eq-Cons-iff*:
$(filter\ P\ ys = x\#xs) =$
$(\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u\in set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
⟨*proof*⟩

lemma *Cons-eq-filter-iff*:
$(x\#xs = filter\ P\ ys) =$
$(\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u\in set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
⟨*proof*⟩

lemma *inj-on-filter-key-eq*:
assumes $inj\text{-}on\ f\ (insert\ y\ (set\ xs))$
shows $[x\leftarrow xs\ .\ f\ y = f\ x] = filter\ (HOL.eq\ y)\ xs$
⟨*proof*⟩

lemma *filter-cong*[*fundef-cong*]:
$xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ ys \Longrightarrow P\ x = Q\ x) \Longrightarrow filter\ P\ xs = filter\ Q\ ys$
⟨*proof*⟩

67.1.9 List partitioning

primrec *partition* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow\ 'a\ list \times\ 'a\ list$ **where**

partition P [] = ([], []) |
partition P (*x* # *xs*) =
  (*let* (*yes*, *no*) = *partition P xs*
   in if P x then (*x* # *yes*, *no*) *else* (*yes*, *x* # *no*))

lemma *partition-filter1*: *fst* (*partition P xs*) = *filter P xs*
⟨*proof*⟩

lemma *partition-filter2*: *snd* (*partition P xs*) = *filter* (*Not o P*) *xs*
⟨*proof*⟩

lemma *partition-P*:
  assumes *partition P xs* = (*yes*, *no*)
  shows (∀ *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*]

67.1.10 *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*⟩

lemma *concat-eq-concat-iff*: $\forall (x, y) \in set\ (zip\ xs\ ys).\ length\ x = length\ y ==> length\ xs = length\ ys ==> (concat\ xs = concat\ ys) = (xs = ys)$
⟨*proof*⟩

lemma *concat-injective*: $concat\ xs = concat\ ys ==> length\ xs = length\ ys ==> \forall (x, y) \in set\ (zip\ xs\ ys).\ length\ x = length\ y ==> xs = ys$
⟨*proof*⟩

67.1.11 *op* !

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-Cons-pos*[*simp*]: $0 < n \Longrightarrow (x\#xs)\ !\ n = xs\ !\ (n - 1)$
⟨*proof*⟩

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 *nth-tl*:
assumes $n < length\ (tl\ x)$ **shows** $tl\ x\ !\ n = x\ !\ Suc\ n$
⟨*proof*⟩

lemma *hd-conv-nth*: $xs \neq [] \Longrightarrow hd\ xs = xs!0$
⟨*proof*⟩

lemma *list-eq-iff-nth-eq*:
$(xs = ys) = (length\ xs = length\ ys \wedge (ALL\ i{<}length\ xs.\ xs!i = ys!i))$
⟨*proof*⟩

lemma *set-conv-nth*: $set\ xs = \{xs!i \mid i.\ i < length\ xs\}$
⟨*proof*⟩

lemma *in-set-conv-nth*: $(x \in set\ xs) = (\exists\, i < length\ xs.\ xs!i = x)$
⟨*proof*⟩

lemma *nth-equal-first-eq*:
  assumes $x \notin set\ xs$
  assumes $n \leq length\ xs$
  shows $(x \ \#\ xs)\ !\ n = x \longleftrightarrow n = 0$ (**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

lemma *nth-non-equal-first-eq*:
  assumes $x \neq y$
  shows $(x \ \#\ xs)\ !\ n = y \longleftrightarrow xs\ !\ (n - 1) = y \wedge n > 0$ (**is** $?lhs \longleftrightarrow ?rhs$)
⟨*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*:
  $(\forall\, x \in set\ xs.\ P\ x) = (\forall\, i.\ i < length\ xs\ -\!-\!> P\ (xs\ !\ i))$
⟨*proof*⟩

lemma *rev-nth*:
  $n < size\ xs \Longrightarrow rev\ xs\ !\ n = xs\ !\ (length\ xs - Suc\ n)$
⟨*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*⟩

67.1.12 *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*]: $i < \mathit{length}\ xs ==> (xs[i{:=}x])!i = x$
⟨*proof*⟩

lemma *nth-list-update-neq* [*simp*]: $i \neq j ==> xs[i{:=}x]!j = xs!j$
⟨*proof*⟩

lemma *list-update-id*[*simp*]: $xs[i := xs!i] = xs$
⟨*proof*⟩

lemma *list-update-beyond*[*simp*]: $\mathit{length}\ xs \leq i \Longrightarrow xs[i{:=}x] = xs$
⟨*proof*⟩

lemma *list-update-nonempty*[*simp*]: $xs[k{:=}x] = [] \longleftrightarrow xs{=}[]$
⟨*proof*⟩

lemma *list-update-same-conv*:
$i < \mathit{length}\ xs ==> (xs[i := x] = xs) = (xs!i = x)$
⟨*proof*⟩

lemma *list-update-append1*:
$i < \mathit{size}\ xs \Longrightarrow (xs\ @\ ys)[i{:=}x] = xs[i{:=}x]\ @\ ys$
⟨*proof*⟩

lemma *list-update-append*:
$(xs\ @\ ys)\ [n{:=}\ x] =$
$(\mathit{if}\ n < \mathit{length}\ xs\ \mathit{then}\ xs[n{:=}\ x]\ @\ ys\ \mathit{else}\ xs\ @\ (ys\ [n{-}\mathit{length}\ xs{:=}\ x]))$
⟨*proof*⟩

lemma *list-update-length* [*simp*]:
$(xs\ @\ x\ \#\ ys)[\mathit{length}\ xs := y] = (xs\ @\ y\ \#\ ys)$
⟨*proof*⟩

lemma *map-update*: $\mathit{map}\ f\ (xs[k{:=}\ y]) = (\mathit{map}\ f\ xs)[k := f\ y]$
⟨*proof*⟩

lemma *rev-update*:
$k < \mathit{length}\ xs \Longrightarrow \mathit{rev}\ (xs[k{:=}\ y]) = (\mathit{rev}\ xs)[\mathit{length}\ xs - k - 1 := y]$
⟨*proof*⟩

lemma *update-zip*:
$(\mathit{zip}\ xs\ ys)[i{:=}xy] = \mathit{zip}\ (xs[i{:=}\mathit{fst}\ xy])\ (ys[i{:=}\mathit{snd}\ xy])$
⟨*proof*⟩

lemma *set-update-subset-insert*: $\mathit{set}(xs[i{:=}x]) <= \mathit{insert}\ x\ (\mathit{set}\ xs)$
⟨*proof*⟩

lemma *set-update-subsetI*: $[|\ \mathit{set}\ xs <= A;\ x{:}A\ |] ==> \mathit{set}(xs[i := x]) <= A$
⟨*proof*⟩

lemma *set-update-memI*: $n < length\ xs \Longrightarrow x \in set\ (xs[n := x])$
⟨*proof*⟩

lemma *list-update-overwrite*[*simp*]:
$xs\ [i := x,\ i := y] = xs\ [i := y]$
⟨*proof*⟩

lemma *list-update-swap*:
$i \neq i' \Longrightarrow xs\ [i := x,\ i' := x'] = xs\ [i' := x',\ i := x]$
⟨*proof*⟩

lemma *list-update-code* [*code*]:
$[][i := y] = []$
$(x\ \#\ xs)[0 := y] = y\ \#\ xs$
$(x\ \#\ xs)[Suc\ i := y] = x\ \#\ xs[i := y]$
⟨*proof*⟩

67.1.13 *last* and *butlast*

lemma *last-snoc* [*simp*]: $last\ (xs\ @\ [x]) = x$
⟨*proof*⟩

lemma *butlast-snoc* [*simp*]: $butlast\ (xs\ @\ [x]) = xs$
⟨*proof*⟩

lemma *last-ConsL*: $xs = [] \Longrightarrow last(x\#xs) = x$
⟨*proof*⟩

lemma *last-ConsR*: $xs \neq [] \Longrightarrow last(x\#xs) = last\ xs$
⟨*proof*⟩

lemma *last-append*: $last(xs\ @\ ys) = (if\ ys = []\ then\ last\ xs\ else\ last\ ys)$
⟨*proof*⟩

lemma *last-appendL*[*simp*]: $ys = [] \Longrightarrow last(xs\ @\ ys) = last\ xs$
⟨*proof*⟩

lemma *last-appendR*[*simp*]: $ys \neq [] \Longrightarrow last(xs\ @\ ys) = last\ ys$
⟨*proof*⟩

lemma *last-tl*: $xs = [] \lor tl\ xs \neq [] \Longrightarrow last\ (tl\ xs) = last\ xs$
⟨*proof*⟩

lemma *butlast-tl*: $butlast\ (tl\ xs) = tl\ (butlast\ xs)$
⟨*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*:
$butlast\ (xs\ @\ ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs\ @\ butlast\ ys)$
⟨*proof*⟩

lemma *append-butlast-last-id* [*simp*]:
$xs \neq []\ ==>\ butlast\ xs\ @\ [last\ xs] = xs$
⟨*proof*⟩

lemma *in-set-butlastD*: $x : set\ (butlast\ xs)\ ==>\ x : set\ xs$
⟨*proof*⟩

lemma *in-set-butlast-appendI*:
$x : set\ (butlast\ xs) \mid x : set\ (butlast\ ys)\ ==>\ x : set\ (butlast\ (xs\ @\ ys))$
⟨*proof*⟩

lemma *last-drop*[*simp*]: $n < length\ xs \Longrightarrow last\ (drop\ n\ xs) = last\ xs$
⟨*proof*⟩

lemma *nth-butlast*:
assumes $n < length\ (butlast\ xs)$ **shows** $butlast\ xs\ !\ n = xs\ !\ n$
⟨*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*⟩

lemma *snoc-eq-iff-butlast*:
$xs\ @\ [x] = ys \longleftrightarrow (ys \neq []\ \&\ butlast\ ys = xs\ \&\ last\ ys = x)$
⟨*proof*⟩

corollary *longest-common-suffix*:
$\exists\, ss\ xs'\ ys'.\ xs = xs'\ @\ ss \wedge ys = ys'\ @\ ss$
$\wedge\ (xs' = [] \vee ys' = [] \vee last\ xs' \neq last\ ys')$
⟨*proof*⟩

67.1.14 *take* and *drop*

lemma *take-0* [*simp*]: $take\ 0\ xs = []$
⟨*proof*⟩

lemma *drop-0* [*simp*]: $drop\ 0\ xs = xs$
⟨*proof*⟩

lemma *take-Suc-Cons* [*simp*]: $take\ (Suc\ n)\ (x\ \#\ xs) = x\ \#\ take\ n\ xs$
⟨*proof*⟩

lemma *drop-Suc-Cons* [*simp*]: $drop\ (Suc\ n)\ (x\ \#\ xs) = drop\ n\ xs$
⟨*proof*⟩

declare *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

lemma *take-Suc*: $xs\ {}^{\sim}\!= []\ ==>\ take\ (Suc\ n)\ xs = hd\ xs\ \#\ take\ n\ (tl\ xs)$
⟨*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 *Cons-nth-drop-Suc*:
$i < length\ xs \Longrightarrow (xs!i)\ \#\ (drop\ (Suc\ i)\ xs) = drop\ i\ xs$
⟨*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 *drop-rev*: $drop\ n\ (rev\ xs) = rev\ (take\ (length\ xs - n)\ xs)$
⟨*proof*⟩

lemma *take-rev*: $take\ n\ (rev\ xs) = rev\ (drop\ (length\ xs - n)\ xs)$
⟨*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*: $n < length\ xs \Longrightarrow hd(drop\ n\ xs) = xs!n$
⟨*proof*⟩

lemma *set-take-subset-set-take*:
$m <= n \Longrightarrow set(take\ m\ xs) <= set(take\ n\ xs)$
⟨*proof*⟩

lemma *set-take-subset*: $set(take\ n\ xs) \subseteq set\ xs$
⟨*proof*⟩

lemma *set-drop-subset*: $set(drop\ n\ xs) \subseteq set\ xs$
⟨*proof*⟩

lemma *set-drop-subset-set-drop*:
$m >= n \Longrightarrow set(drop\ m\ xs) <= set(drop\ n\ xs)$
⟨*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*: $take\ (i+j)\ xs = take\ i\ xs\ @\ take\ j\ (drop\ i\ xs)$
⟨*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 *take-update-cancel*[*simp*]: $n \leq m \Longrightarrow take\ n\ (xs[m := y]) = take\ n\ xs$
⟨*proof*⟩

lemma *drop-update-cancel*[*simp*]: $n < m \Longrightarrow drop\ m\ (xs[n := x]) = drop\ m\ xs$
⟨*proof*⟩

lemma *upd-conv-take-nth-drop*:
$i < length\ xs \Longrightarrow xs[i:=a] = take\ i\ xs\ @\ a\ \#\ drop\ (Suc\ i)\ xs$
⟨*proof*⟩

lemma *take-update-swap*: $n < m \Longrightarrow take\ m\ (xs[n := x]) = (take\ m\ xs)[n := x]$
⟨*proof*⟩

lemma *drop-update-swap*: $m \leq n \Longrightarrow drop\ m\ (xs[n := x]) = (drop\ m\ xs)[n-m$

$:= x]$
⟨proof⟩

lemma *nth-image*: $l \leq size\ xs \Longrightarrow nth\ xs\ `\ \{0..<l\} = set(take\ l\ xs)$
⟨proof⟩

67.1.15 *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*]:
$[|\ x{:}set\ xs;\ \sim P(x)|] ==> takeWhile\ P\ (xs\ @\ ys) = takeWhile\ P\ xs$
⟨proof⟩

lemma *takeWhile-append2* [*simp*]:
$(!!x.\ x : set\ xs ==> P\ x) ==> takeWhile\ P\ (xs\ @\ ys) = xs\ @\ takeWhile\ P\ ys$
⟨proof⟩

lemma *takeWhile-tail*: $\neg\ P\ x ==> takeWhile\ P\ (xs\ @\ (x\#l)) = takeWhile\ P\ xs$
⟨proof⟩

lemma *takeWhile-nth*: $j < length\ (takeWhile\ P\ xs) \Longrightarrow takeWhile\ P\ xs\ !\ j = xs$
$!\ j$
⟨proof⟩

lemma *dropWhile-nth*: $j < length\ (dropWhile\ P\ xs) \Longrightarrow$
$dropWhile\ P\ xs\ !\ j = xs\ !\ (j + length\ (takeWhile\ P\ xs))$
⟨proof⟩

lemma *length-dropWhile-le*: $length\ (dropWhile\ P\ xs) \leq length\ xs$
⟨proof⟩

lemma *dropWhile-append1* [*simp*]:
$[|\ x : set\ xs;\ \sim P(x)|] ==> dropWhile\ P\ (xs\ @\ ys) = (dropWhile\ P\ xs)@ys$
⟨proof⟩

lemma *dropWhile-append2* [*simp*]:
$(!!x.\ x{:}set\ xs ==> P(x)) ==> dropWhile\ P\ (xs\ @\ ys) = dropWhile\ P\ ys$
⟨proof⟩

lemma *dropWhile-append3*:
$\neg\ P\ y \Longrightarrow dropWhile\ P\ (xs\ @\ y\ \#\ ys) = dropWhile\ P\ xs\ @\ y\ \#\ ys$
⟨proof⟩

lemma *dropWhile-last*:

$x \in set\ xs \Longrightarrow \neg\ P\ x \Longrightarrow last\ (dropWhile\ P\ xs) = last\ xs$
⟨proof⟩

lemma *set-dropWhileD*: $x \in set\ (dropWhile\ P\ xs) \Longrightarrow x \in set\ xs$
⟨proof⟩

lemma *set-takeWhileD*: $x : set\ (takeWhile\ P\ xs) ==> x : set\ xs \wedge P\ x$
⟨proof⟩

lemma *takeWhile-eq-all-conv*[*simp*]:
$(takeWhile\ P\ xs = xs) = (\forall\, x \in set\ xs.\ P\ x)$
⟨proof⟩

lemma *dropWhile-eq-Nil-conv*[*simp*]:
$(dropWhile\ P\ xs = []) = (\forall\, x \in set\ xs.\ P\ x)$
⟨proof⟩

lemma *dropWhile-eq-Cons-conv*:
$(dropWhile\ P\ xs = y\#ys) = (xs = takeWhile\ P\ xs\ @\ y\ \#\ ys\ \&\ \neg\ P\ y)$
⟨proof⟩

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*: $takeWhile\ P\ (map\ f\ xs) = map\ f\ (takeWhile\ (P \circ f)\ xs)$
⟨proof⟩

lemma *dropWhile-map*: $dropWhile\ P\ (map\ f\ xs) = map\ f\ (dropWhile\ (P \circ f)\ xs)$
⟨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*: $dropWhile\ P\ xs \neq [] \Longrightarrow \neg\ P\ (hd\ (dropWhile\ P\ xs))$
⟨proof⟩

lemma *takeWhile-eq-filter*:
assumes $\bigwedge x.\ x \in set\ (dropWhile\ P\ xs) \Longrightarrow \neg\ P\ x$
shows $takeWhile\ P\ xs = filter\ P\ xs$
⟨proof⟩

lemma *takeWhile-eq-take-P-nth*:
$[\![\bigwedge i.\ [\![i < n\ ;\ i < length\ xs]\!] \Longrightarrow P\ (xs\ !\ i)\ ;\ n < length\ xs \Longrightarrow \neg\ P\ (xs\ !\ n)$

$\rrbracket \Longrightarrow$
$takeWhile\ P\ xs = take\ n\ xs$
⟨proof⟩

lemma *nth-length-takeWhile*:
$length\ (takeWhile\ P\ xs) < length\ xs \Longrightarrow \neg\ P\ (xs\ !\ length\ (takeWhile\ P\ xs))$
⟨proof⟩

lemma *length-takeWhile-less-P-nth*:
assumes *all*: $\bigwedge\ i.\ i < j \Longrightarrow P\ (xs\ !\ i)$ **and** $j \leq length\ xs$
shows $j \leq length\ (takeWhile\ P\ xs)$
⟨proof⟩

lemma *takeWhile-neq-rev*: $\llbracket distinct\ xs;\ x \in set\ xs\rrbracket \Longrightarrow$
$takeWhile\ (\lambda y.\ y \neq x)\ (rev\ xs) = rev\ (tl\ (dropWhile\ (\lambda y.\ y \neq x)\ xs))$
⟨proof⟩

lemma *dropWhile-neq-rev*: $\llbracket distinct\ xs;\ x \in set\ xs\rrbracket \Longrightarrow$
$dropWhile\ (\lambda y.\ y \neq x)\ (rev\ xs) = x\ \#\ rev\ (takeWhile\ (\lambda y.\ y \neq x)\ xs)$
⟨proof⟩

lemma *takeWhile-not-last*:
$distinct\ xs \Longrightarrow takeWhile\ (\lambda y.\ y \neq last\ xs)\ xs = butlast\ xs$
⟨proof⟩

lemma *takeWhile-cong* [*fundef-cong*]:
$[|\ l = k;\ !!x.\ x : set\ l ==> P\ x = Q\ x\ |]$
$==> takeWhile\ P\ l = takeWhile\ Q\ k$
⟨proof⟩

lemma *dropWhile-cong* [*fundef-cong*]:
$[|\ l = k;\ !!x.\ x : set\ l ==> P\ x = Q\ x\ |]$
$==> dropWhile\ P\ l = dropWhile\ Q\ k$
⟨proof⟩

lemma *takeWhile-idem* [*simp*]:
$takeWhile\ P\ (takeWhile\ P\ xs) = takeWhile\ P\ xs$
⟨proof⟩

lemma *dropWhile-idem* [*simp*]:
$dropWhile\ P\ (dropWhile\ P\ xs) = dropWhile\ P\ xs$
⟨proof⟩

67.1.16 *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* |] ==>
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 *zip-replicate1*: *zip* (*replicate n x*) *ys* = *map* (*Pair x*) (*take n ys*)
⟨*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*: $zip\ xs\ (takeWhile\ P\ ys) = takeWhile\ (P \circ snd)\ (zip\ xs\ ys)$
⟨*proof*⟩

lemma *set-zip-leftD*: $(x,y) \in set\ (zip\ xs\ ys) \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *set-zip-rightD*: $(x,y) \in set\ (zip\ xs\ ys) \Longrightarrow y \in set\ ys$
⟨*proof*⟩

lemma *in-set-zipE*:
$(x,y) : set(zip\ xs\ ys) \Longrightarrow (\llbracket\ x : set\ xs;\ y : set\ ys\ \rrbracket \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *zip-map-fst-snd*: $zip\ (map\ fst\ zs)\ (map\ snd\ zs) = zs$
⟨*proof*⟩

lemma *zip-eq-conv*:
$length\ xs = length\ ys \Longrightarrow zip\ xs\ ys = zs \longleftrightarrow map\ fst\ zs = xs \wedge map\ snd\ zs = ys$
⟨*proof*⟩

lemma *in-set-zip*:
$p \in set\ (zip\ xs\ ys) \longleftrightarrow (\exists\, n.\ xs\ !\ n = fst\ p \wedge ys\ !\ n = snd\ p$
$\wedge\ n < length\ xs \wedge n < length\ ys)$
⟨*proof*⟩

lemma *in-set-impl-in-set-zip1*:
assumes $length\ xs = length\ ys$
assumes $x \in set\ xs$
obtains y **where** $(x,\ y) \in set\ (zip\ xs\ ys)$
⟨*proof*⟩

lemma *in-set-impl-in-set-zip2*:
assumes $length\ xs = length\ ys$
assumes $y \in set\ ys$
obtains x **where** $(x,\ y) \in set\ (zip\ xs\ ys)$
⟨*proof*⟩

lemma *pair-list-eqI*:
assumes $map\ fst\ xs = map\ fst\ ys$ **and** $map\ snd\ xs = map\ snd\ ys$
shows $xs = ys$
⟨*proof*⟩

67.1.17 *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-induct*
[*consumes 1*, *case-names Nil Cons*, *induct set*: *list-all2*]:
assumes *P*: *list-all2 P xs ys*
assumes *Nil*: *R* [] []
assumes *Cons*: ⋀*x xs y ys*.
⟦*P x y*; *list-all2 P xs ys*; *R xs ys*⟧ ⟹ *R* (*x* # *xs*) (*y* # *ys*)
shows *R xs 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*:
$length\ xs = length\ ys \Longrightarrow$
$list\text{-}all2\ P\ (xs@us)\ (ys@vs) = (list\text{-}all2\ P\ xs\ ys \wedge list\text{-}all2\ P\ us\ vs)$
⟨*proof*⟩

lemma *list-all2-appendI* [*intro?*, *trans*]:
$[\![\ list\text{-}all2\ P\ a\ b;\ list\text{-}all2\ P\ c\ d\ ]\!] \Longrightarrow list\text{-}all2\ P\ (a@c)\ (b@d)$
⟨*proof*⟩

lemma *list-all2-conv-all-nth*:
$list\text{-}all2\ P\ xs\ ys =$
$(length\ xs = length\ ys \wedge (\forall\, i < length\ xs.\ P\ (xs!i)\ (ys!i)))$
⟨*proof*⟩

lemma *list-all2-trans*:
assumes *tr*: $!!a\ b\ c.\ P1\ a\ b ==> P2\ b\ c ==> P3\ a\ c$
shows $!!bs\ cs.\ list\text{-}all2\ P1\ as\ bs ==> list\text{-}all2\ P2\ bs\ cs ==> list\text{-}all2\ P3\ as\ cs$
(**is** $!!bs\ cs.\ PROP\ ?Q\ as\ bs\ cs$)
⟨*proof*⟩

lemma *list-all2-all-nthI* [*intro?*]:
$length\ a = length\ b \Longrightarrow (\bigwedge n.\ n < length\ a \Longrightarrow P\ (a!n)\ (b!n)) \Longrightarrow list\text{-}all2\ P\ a\ b$
⟨*proof*⟩

lemma *list-all2I*:
$\forall\, x \in set\ (zip\ a\ b).\ case\text{-}prod\ P\ x \Longrightarrow length\ a = length\ b \Longrightarrow list\text{-}all2\ P\ a\ b$
⟨*proof*⟩

lemma *list-all2-nthD*:
$[\![\ list\text{-}all2\ P\ xs\ ys;\ p < size\ xs\ ]\!] \Longrightarrow P\ (xs!p)\ (ys!p)$
⟨*proof*⟩

lemma *list-all2-nthD2*:
$[\![list\text{-}all2\ P\ xs\ ys;\ p < size\ ys]\!] \Longrightarrow P\ (xs!p)\ (ys!p)$
⟨*proof*⟩

lemma *list-all2-map1*:
$list\text{-}all2\ P\ (map\ f\ as)\ bs = list\text{-}all2\ (\lambda x\ y.\ P\ (f\ x)\ y)\ as\ bs$
⟨*proof*⟩

lemma *list-all2-map2*:
$list\text{-}all2\ P\ as\ (map\ f\ bs) = list\text{-}all2\ (\lambda x\ y.\ P\ x\ (f\ y))\ as\ bs$
⟨*proof*⟩

lemma *list-all2-refl* [*intro?*]:
$(\bigwedge x.\ P\ x\ x) \Longrightarrow list\text{-}all2\ P\ xs\ xs$
⟨*proof*⟩

lemma *list-all2-update-cong*:
$\llbracket$ *list-all2 P xs ys*; *P x y* $\rrbracket \Longrightarrow$ *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*⟩

lemma *list-eq-iff-zip-eq*:
xs = *ys* $\longleftrightarrow$ *length xs* = *length ys* $\wedge$ ($\forall$ (*x*,*y*) $\in$ *set* (*zip xs ys*). *x* = *y*)
⟨*proof*⟩

lemma *list-all2-same*: *list-all2 P xs xs* $\longleftrightarrow$ ($\forall$ *x*$\in$*set xs*. *P x x*)
⟨*proof*⟩

lemma *zip-assoc*:
zip xs (*zip ys zs*) = *map* (λ((*x*, *y*), *z*). (*x*, *y*, *z*)) (*zip* (*zip xs ys*) *zs*)
⟨*proof*⟩

lemma *zip-commute*: *zip xs ys* = *map* (λ(*x*, *y*). (*y*, *x*)) (*zip ys xs*)
⟨*proof*⟩

lemma *zip-left-commute*:
zip xs (*zip ys zs*) = *map* (λ(*y*, (*x*, *z*)). (*x*, *y*, *z*)) (*zip ys* (*zip xs zs*))
⟨*proof*⟩

lemma *zip-replicate2*: *zip xs* (*replicate n y*) = *map* (λ*x*. (*x*, *y*)) (*take n xs*)
⟨*proof*⟩

67.1.18 *List.product* and *product-lists*

lemma *product-concat-map*:
List.product xs ys = *concat* (*map* (λ*x*. *map* (λ*y*. (*x*,*y*)) *ys*) *xs*)
⟨*proof*⟩

lemma *set-product*[*simp*]: *set* (*List.product xs ys*) = *set xs* $\times$ *set ys*
⟨*proof*⟩

lemma *length-product* [*simp*]:
$length\ (List.product\ xs\ ys) = length\ xs * length\ ys$
⟨*proof*⟩

lemma *product-nth*:
assumes $n < length\ xs * length\ ys$
shows $List.product\ xs\ ys\ !\ n = (xs\ !\ (n\ div\ length\ ys),\ ys\ !\ (n\ mod\ length\ ys))$
⟨*proof*⟩

lemma *in-set-product-lists-length*:
$xs \in set\ (product\text{-}lists\ xss) \Longrightarrow length\ xs = length\ xss$
⟨*proof*⟩

lemma *product-lists-set*:
$set\ (product\text{-}lists\ xss) = \{xs.\ list\text{-}all2\ (\lambda x\ ys.\ x \in set\ ys)\ xs\ xss\}$ (**is** $?L = Collect\ ?R$)
⟨*proof*⟩

67.1.19 *fold* with natural argument order

lemma *fold-simps* [*code*]: — eta-expanded variant for generated code – enables tail-recursion optimisation in Scala
$fold\ f\ []\ s = s$
$fold\ f\ (x\ \#\ xs)\ s = fold\ f\ xs\ (f\ x\ s)$
⟨*proof*⟩

lemma *fold-remove1-split*:
$[\![\ \bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ x \circ f\ y = f\ y \circ f\ x;$
$x \in set\ xs\ ]\!]$
$\Longrightarrow fold\ f\ xs = fold\ f\ (remove1\ x\ xs) \circ f\ x$
⟨*proof*⟩

lemma *fold-cong* [*fundef-cong*]:
$a = b \Longrightarrow xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x)$
$\Longrightarrow fold\ f\ xs\ a = fold\ g\ ys\ b$
⟨*proof*⟩

lemma *fold-id*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = id) \Longrightarrow fold\ f\ xs = id$
⟨*proof*⟩

lemma *fold-commute*:
$(\bigwedge x.\ x \in set\ xs \Longrightarrow h \circ g\ x = f\ x \circ h) \Longrightarrow h \circ fold\ g\ xs = fold\ f\ xs \circ h$
⟨*proof*⟩

lemma *fold-commute-apply*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow h \circ g\ x = f\ x \circ h$
shows $h\ (fold\ g\ xs\ s) = fold\ f\ xs\ (h\ s)$
⟨*proof*⟩

lemma *fold-invariant*:
$[\![\bigwedge x.\ x \in set\ xs \Longrightarrow Q\ x;\ \ P\ s;\ \ \bigwedge x\ s.\ Q\ x \Longrightarrow P\ s \Longrightarrow P\ (f\ x\ s)\ ]\!]$
$\Longrightarrow P\ (fold\ f\ xs\ s)$
⟨*proof*⟩

lemma *fold-append* [*simp*]: $fold\ f\ (xs\ @\ ys) = fold\ f\ ys \circ fold\ f\ xs$
⟨*proof*⟩

lemma *fold-map* [*code-unfold*]: $fold\ g\ (map\ f\ xs) = fold\ (g\ o\ f)\ xs$
⟨*proof*⟩

lemma *fold-filter*:
$fold\ f\ (filter\ P\ xs) = fold\ (\lambda x.\ if\ P\ x\ then\ f\ x\ else\ id)\ xs$
⟨*proof*⟩

lemma *fold-rev*:
$(\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y)$
$\Longrightarrow fold\ f\ (rev\ xs) = fold\ f\ xs$
⟨*proof*⟩

lemma *fold-Cons-rev*: $fold\ Cons\ xs = append\ (rev\ xs)$
⟨*proof*⟩

lemma *rev-conv-fold* [*code*]: $rev\ xs = fold\ Cons\ xs\ []$
⟨*proof*⟩

lemma *fold-append-concat-rev*: $fold\ append\ xss = append\ (concat\ (rev\ xss))$
⟨*proof*⟩

Finite-Set.fold and *fold*

lemma (**in** *comp-fun-commute*) *fold-set-fold-remdups*:
$Finite\text{-}Set.fold\ f\ y\ (set\ xs) = fold\ f\ (remdups\ xs)\ y$
⟨*proof*⟩

lemma (**in** *comp-fun-idem*) *fold-set-fold*:
$Finite\text{-}Set.fold\ f\ y\ (set\ xs) = fold\ f\ xs\ y$
⟨*proof*⟩

lemma *union-set-fold* [*code*]: $set\ xs \cup A = fold\ Set.insert\ xs\ A$
⟨*proof*⟩

lemma *union-coset-filter* [*code*]:
$List.coset\ xs \cup A = List.coset\ (List.filter\ (\lambda x.\ x \notin A)\ xs)$
⟨*proof*⟩

lemma *minus-set-fold* [*code*]: $A - set\ xs = fold\ Set.remove\ xs\ A$
⟨*proof*⟩

lemma *minus-coset-filter* [*code*]:
$A - List.coset\ xs = set\ (List.filter\ (\lambda x.\ x \in A)\ xs)$
⟨*proof*⟩

lemma *inter-set-filter* [*code*]:
$A \cap set\ xs = set\ (List.filter\ (\lambda x.\ x \in A)\ xs)$
⟨*proof*⟩

lemma *inter-coset-fold* [*code*]:
$A \cap List.coset\ xs = fold\ Set.remove\ xs\ A$
⟨*proof*⟩

lemma (**in** *semilattice-set*) *set-eq-fold* [*code*]:
$F\ (set\ (x\ \#\ xs)) = fold\ f\ xs\ x$
⟨*proof*⟩

lemma (**in** *complete-lattice*) *Inf-set-fold*:
$Inf\ (set\ xs) = fold\ inf\ xs\ top$
⟨*proof*⟩

declare *Inf-set-fold* [**where** $'a = 'a\ set$, *code*]

lemma (**in** *complete-lattice*) *Sup-set-fold*:
$Sup\ (set\ xs) = fold\ sup\ xs\ bot$
⟨*proof*⟩

declare *Sup-set-fold* [**where** $'a = 'a\ set$, *code*]

lemma (**in** *complete-lattice*) *INF-set-fold*:
$INFIMUM\ (set\ xs)\ f = fold\ (inf \circ f)\ xs\ top$
⟨*proof*⟩

declare *INF-set-fold* [*code*]

lemma (**in** *complete-lattice*) *SUP-set-fold*:
$SUPREMUM\ (set\ xs)\ f = fold\ (sup \circ f)\ xs\ bot$
⟨*proof*⟩

declare *SUP-set-fold* [*code*]

67.1.20 Fold variants: *foldr* and *foldl*

Correspondence

lemma *foldr-conv-fold* [*code-abbrev*]: $foldr\ f\ xs = fold\ f\ (rev\ xs)$
⟨*proof*⟩

lemma *foldl-conv-fold*: $foldl\ f\ s\ xs = fold\ (\lambda x\ s.\ f\ s\ x)\ xs\ s$
⟨*proof*⟩

lemma *foldr-conv-foldl*: — The "Third Duality Theorem" in Bird & Wadler:
$foldr\ f\ xs\ a = foldl\ (\lambda x\ y.\ f\ y\ x)\ a\ (rev\ xs)$
⟨*proof*⟩

lemma *foldl-conv-foldr*:
$foldl\ f\ a\ xs = foldr\ (\lambda x\ y.\ f\ y\ x)\ (rev\ xs)\ a$
⟨*proof*⟩

lemma *foldr-fold*:
$(\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y)$
$\Longrightarrow foldr\ f\ xs = fold\ f\ xs$
⟨*proof*⟩

lemma *foldr-cong* [*fundef-cong*]:
$a = b \Longrightarrow l = k \Longrightarrow (\bigwedge a\ x.\ x \in set\ l \Longrightarrow f\ x\ a = g\ x\ a) \Longrightarrow foldr\ f\ l\ a = foldr\ g\ k\ b$
⟨*proof*⟩

lemma *foldl-cong* [*fundef-cong*]:
$a = b \Longrightarrow l = k \Longrightarrow (\bigwedge a\ x.\ x \in set\ l \Longrightarrow f\ a\ x = g\ a\ x) \Longrightarrow foldl\ f\ a\ l = foldl\ g\ b\ k$
⟨*proof*⟩

lemma *foldr-append* [*simp*]: $foldr\ f\ (xs\ @\ ys)\ a = foldr\ f\ xs\ (foldr\ f\ ys\ a)$
⟨*proof*⟩

lemma *foldl-append* [*simp*]: $foldl\ f\ a\ (xs\ @\ ys) = foldl\ f\ (foldl\ f\ a\ xs)\ ys$
⟨*proof*⟩

lemma *foldr-map* [*code-unfold*]: $foldr\ g\ (map\ f\ xs)\ a = foldr\ (g\ o\ f)\ xs\ a$
⟨*proof*⟩

lemma *foldr-filter*:
$foldr\ f\ (filter\ P\ xs) = foldr\ (\lambda x.\ if\ P\ x\ then\ f\ x\ else\ id)\ xs$
⟨*proof*⟩

lemma *foldl-map* [*code-unfold*]:
$foldl\ g\ a\ (map\ f\ xs) = foldl\ (\lambda a\ x.\ g\ a\ (f\ x))\ a\ xs$
⟨*proof*⟩

lemma *concat-conv-foldr* [*code*]:
$concat\ xss = foldr\ append\ xss\ []$
⟨*proof*⟩

67.1.21 *upt*

lemma *upt-rec*[*code*]: $[i..{<}j] = (if\ i{<}j\ then\ i\#[Suc\ i..{<}j]\ else\ [])$
— simp does not terminate!
⟨*proof*⟩

lemmas *upt-rec-numeral*[*simp*] = *upt-rec*[*of numeral m numeral n*] **for** *m n*

lemma *upt-conv-Nil* [*simp*]: $j <= i ==> [i..<j] = []$
⟨*proof*⟩

lemma *upt-eq-Nil-conv*[*simp*]: $([i..<j] = []) = (j = 0 \vee j <= i)$
⟨*proof*⟩

lemma *upt-eq-Cons-conv*:
$([i..<j] = x\#xs) = (i < j \ \& \ i = x \ \& \ [i+1..<j] = xs)$
⟨*proof*⟩

lemma *upt-Suc-append*: $i <= j ==> [i..<(Suc\ j)] = [i..<j]@[j]$
— Only needed if *upt-Suc* is deleted from the simpset.
⟨*proof*⟩

lemma *upt-conv-Cons*: $i < j ==> [i..<j] = i \ \# \ [Suc\ i..<j]$
⟨*proof*⟩

lemma *upt-conv-Cons-Cons*: — no precondition
$m \ \# \ n \ \# \ ns = [m..<q] \longleftrightarrow n \ \# \ ns = [Suc\ m..<q]$
⟨*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 *tl-upt*: $tl\ [m..<n] = [Suc\ m..<n]$
⟨*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 *map-add-upt*: $map\ (\lambda i.\ i + n)\ [0..<m] = [n..<m + n]$
⟨*proof*⟩

lemma *nth-map-upt*: $i < n-m ==> (map\ f\ [m..<n])\ !\ i = f(m+i)$
⟨*proof*⟩

lemma *map-decr-upt*: $map\ (\lambda n.\ n - Suc\ 0)\ [Suc\ m..<Suc\ n] = [m..<n]$
⟨*proof*⟩

lemma *map-upt-Suc*: $map\ f\ [0\ ..<\ Suc\ n] = f\ 0\ \#\ map\ (\lambda i.\ f\ (Suc\ i))\ [0\ ..<\ n]$
⟨*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*:
$map\ (\lambda i.\ xs\ !\ i)\ [0..<length\ xs] = xs$
⟨*proof*⟩

lemma *list-all2-antisym*:
$[\![\ (\bigwedge x\ y.\ [\![P\ x\ y;\ Q\ y\ x]\!] \Longrightarrow x = y);\ list\text{-}all2\ P\ xs\ ys;\ list\text{-}all2\ Q\ ys\ xs\ ]\!]$
$\Longrightarrow xs = ys$
⟨*proof*⟩

lemma *take-equalityI*: $(\forall\, i.\ take\ i\ xs = take\ i\ ys) ==> xs = ys$
— 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*⟩

lemma *take-Cons-numeral* [*simp*]:
take (*numeral v*) (*x* # *xs*) = *x* # *take* (*numeral v* − *1*) *xs*
⟨*proof*⟩

lemma *drop-Cons-numeral* [*simp*]:
drop (*numeral v*) (*x* # *xs*) = *drop* (*numeral v* − *1*) *xs*
⟨*proof*⟩

lemma *nth-Cons-numeral* [*simp*]:
(*x* # *xs*) ! *numeral v* = *xs* ! (*numeral v* − *1*)
⟨*proof*⟩

67.1.22 *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*[*simp del*]

lemmas *upto-rec-numeral* [*simp*] =
upto.simps[*of numeral m numeral n*]
upto.simps[*of numeral m* − *numeral n*]
upto.simps[*of* − *numeral m numeral n*]
upto.simps[*of* − *numeral m* − *numeral n*] **for** *m n*

lemma *upto-empty*[*simp*]: *j* < *i* ⟹ [*i*..*j*] = []
⟨*proof*⟩

lemma *upto-rec1*: *i* ≤ *j* ⟹ [*i*..*j*] = *i*#[*i*+*1*..*j*]
⟨*proof*⟩

lemma *upto-rec2*: *i* ≤ *j* ⟹ [*i*..*j*] = [*i*..*j* − *1*]@[*j*]
⟨*proof*⟩

lemma *set-upto*[*simp*]: *set*[*i*..*j*] = {*i*..*j*}
⟨*proof*⟩

Tail recursive version for code generation:

definition *upto-aux* :: *int* ⇒ *int* ⇒ *int list* ⇒ *int list* **where**
upto-aux i j js = [*i*..*j*] @ *js*

lemma *upto-aux-rec* [*code*]:
upto-aux i j js = (*if j*<*i then js else upto-aux i* (*j* − *1*) (*j*#*js*))
⟨*proof*⟩

lemma *upto-code*[*code*]: $[i..j] = upto\text{-}aux\ i\ j\ []$
⟨*proof*⟩

67.1.23 *distinct* and *remdups* and *remdups-adj*

lemma *distinct-tl*: $distinct\ xs \Longrightarrow distinct\ (tl\ xs)$
⟨*proof*⟩

lemma *distinct-append* [*simp*]:
$distinct\ (xs\ @\ ys) = (distinct\ xs \wedge distinct\ ys \wedge set\ xs \cap set\ ys = \{\})$
⟨*proof*⟩

lemma *distinct-rev*[*simp*]: $distinct(rev\ xs) = distinct\ xs$
⟨*proof*⟩

lemma *set-remdups* [*simp*]: $set\ (remdups\ xs) = set\ xs$
⟨*proof*⟩

lemma *distinct-remdups* [*iff*]: $distinct\ (remdups\ xs)$
⟨*proof*⟩

lemma *distinct-remdups-id*: $distinct\ xs ==> remdups\ xs = xs$
⟨*proof*⟩

lemma *remdups-id-iff-distinct* [*simp*]: $remdups\ xs = xs \longleftrightarrow distinct\ xs$
⟨*proof*⟩

lemma *finite-distinct-list*: $finite\ A \Longrightarrow EX\ xs.\ set\ xs = A\ \&\ distinct\ xs$
⟨*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*:
$distinct(map\ f\ xs) = (distinct\ xs\ \&\ inj\text{-}on\ f\ (set\ xs))$
⟨*proof*⟩

lemma *distinct-map-filter*:
$distinct\ (map\ f\ xs) \Longrightarrow distinct\ (map\ f\ (filter\ P\ xs))$
⟨*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*]: $distinct\ xs \Longrightarrow distinct\ (take\ i\ xs)$
⟨*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*:
$\llbracket$ $distinct\ xs$;
$\bigwedge ys.\ ys \in set\ xs \Longrightarrow distinct\ ys$;
$\bigwedge ys\ zs.\ \llbracket\ ys \in set\ xs\ ;\ zs \in set\ xs\ ;\ ys \neq zs\ \rrbracket \Longrightarrow set\ ys \cap set\ zs = \{\}$
$\rrbracket \Longrightarrow distinct\ (concat\ xs)$
⟨*proof*⟩

It is best to avoid this indexed version of distinct, but sometimes it is useful.

lemma *distinct-conv-nth*:
$distinct\ xs = (\forall\, i < size\ xs.\ \forall\, j < size\ xs.\ i \neq j$ --> $xs!i \neq xs!j)$
⟨*proof*⟩

lemma *nth-eq-iff-index-eq*:
$\llbracket\ distinct\ xs;\ i < length\ xs;\ j < length\ xs\ \rrbracket \Longrightarrow (xs!i = xs!j) = (i = j)$
⟨*proof*⟩

lemma *distinct-Ex1*:
$distinct\ xs \Longrightarrow x \in set\ xs \Longrightarrow (\exists!i.\ i < length\ xs \wedge xs\ !\ i = x)$
⟨*proof*⟩

lemma *inj-on-nth*: $distinct\ xs \Longrightarrow \forall\, i \in I.\ i < length\ xs \Longrightarrow inj\text{-}on\ (nth\ xs)\ I$
⟨*proof*⟩

lemma *bij-betw-nth*:

assumes *distinct xs* $A = \{..<\mathit{length}\ xs\}$ $B = \mathit{set}\ xs$
shows *bij-betw* (*op* ! *xs*) *A B*
⟨*proof*⟩

lemma *set-update-distinct*: $[\![$ *distinct xs*; $n < \mathit{length}\ xs$ $]\!] \Longrightarrow$
$\mathit{set}(xs[n := x]) = \mathit{insert}\ x\ (\mathit{set}\ xs - \{xs!n\})$
⟨*proof*⟩

lemma *distinct-swap*[*simp*]: $[\![\ i < \mathit{size}\ xs;\ j < \mathit{size}\ xs\ ]\!] \Longrightarrow$
$\mathit{distinct}(xs[i := xs!j,\ j := xs!i]) = \mathit{distinct}\ xs$
⟨*proof*⟩

lemma *set-swap*[*simp*]:
$[\![\ i < \mathit{size}\ xs;\ j < \mathit{size}\ xs\ ]\!] \Longrightarrow \mathit{set}(xs[i := xs!j,\ j := xs!i]) = \mathit{set}\ xs$
⟨*proof*⟩

lemma *distinct-card*: *distinct xs* ==> *card* (*set xs*) = *size xs*
⟨*proof*⟩

lemma *card-distinct*: *card* (*set xs*) = *size xs* ==> *distinct xs*
⟨*proof*⟩

lemma *distinct-length-filter*: *distinct xs* $\Longrightarrow$ *length* (*filter P xs*) = *card* ({*x*. *P x*}
Int set xs)
⟨*proof*⟩

lemma *not-distinct-decomp*: ~ *distinct ws* ==> *EX xs ys zs y*. *ws* = *xs*@[*y*]@*ys*@[*y*]@*zs*
⟨*proof*⟩

lemma *not-distinct-conv-prefix*:
defines $\mathit{dec}\ as\ xs\ y\ ys \equiv y \in \mathit{set}\ xs \wedge \mathit{distinct}\ xs \wedge as = xs\ @\ y\ \#\ ys$
shows $\neg\mathit{distinct}\ as \longleftrightarrow (\exists\, xs\ y\ ys.\ \mathit{dec}\ as\ xs\ y\ ys)$ (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *distinct-product*:
distinct xs $\Longrightarrow$ *distinct ys* $\Longrightarrow$ *distinct* (*List.product xs ys*)
⟨*proof*⟩

lemma *distinct-product-lists*:
assumes $\forall\, xs \in \mathit{set}\ xss.\ \mathit{distinct}\ xs$
shows *distinct* (*product-lists xss*)
⟨*proof*⟩

lemma *length-remdups-concat*:
$\mathit{length}\ (\mathit{remdups}\ (\mathit{concat}\ xss)) = \mathit{card}\ (\bigcup xs{\in}\mathit{set}\ xss.\ \mathit{set}\ xs)$
⟨*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 $distinct\ xs$
shows $distinct\ (butlast\ xs)$
⟨*proof*⟩

lemma *remdups-map-remdups*:
$remdups\ (map\ f\ (remdups\ xs)) = remdups\ (map\ f\ xs)$
⟨*proof*⟩

lemma *distinct-zipI1*:
assumes $distinct\ xs$
shows $distinct\ (zip\ xs\ ys)$
⟨*proof*⟩

lemma *distinct-zipI2*:
assumes $distinct\ ys$
shows $distinct\ (zip\ xs\ ys)$
⟨*proof*⟩

lemma *set-take-disj-set-drop-if-distinct*:
$distinct\ vs \Longrightarrow i \leq j \Longrightarrow set\ (take\ i\ vs) \cap set\ (drop\ j\ vs) = \{\}$
⟨*proof*⟩

lemma *distinct-singleton*: $distinct\ [x]$ ⟨*proof*⟩

lemma *distinct-length-2-or-more*:
$distinct\ (a\ \#\ b\ \#\ xs) \longleftrightarrow (a \neq b \wedge distinct\ (a\ \#\ xs) \wedge distinct\ (b\ \#\ xs))$
⟨*proof*⟩

lemma *remdups-adj-altdef*: $(remdups\text{-}adj\ xs = ys) \longleftrightarrow$
$(\exists f{::}nat \Rightarrow nat.\ mono\ f\ \&\ f\ `\ \{0\ ..<\ size\ xs\} = \{0\ ..<\ size\ ys\}$
$\wedge (\forall i < size\ xs.\ xs!i = ys!(f\ i))$
$\wedge (\forall i.\ i + 1 < size\ xs \longrightarrow (xs!i = xs!(i+1) \longleftrightarrow f\ i = f(i+1))))$ (**is** $?L \longleftrightarrow$
$(\exists f.\ ?p\ f\ xs\ ys))$
⟨*proof*⟩

lemma *hd-remdups-adj*[*simp*]: $hd\ (remdups\text{-}adj\ xs) = hd\ xs$
⟨*proof*⟩

lemma *remdups-adj-Cons*: $remdups\text{-}adj\ (x\ \#\ xs) =$
$(case\ remdups\text{-}adj\ xs\ of\ [] \Rightarrow [x]\ |\ y\ \#\ xs \Rightarrow if\ x = y\ then\ y\ \#\ xs\ else\ x\ \#\ y\ \#\ xs)$
⟨*proof*⟩

lemma *remdups-adj-append-two*:
remdups-adj (*xs* @ [*x*,*y*]) = *remdups-adj* (*xs* @ [*x*]) @ (*if* $x = y$ *then* [] *else* [*y*])
⟨*proof*⟩

lemma *remdups-adj-adjacent*:
Suc $i <$ *length* (*remdups-adj xs*) $\Longrightarrow$ *remdups-adj xs* ! $i \neq$ *remdups-adj xs* ! *Suc* i
⟨*proof*⟩

lemma *remdups-adj-rev*[*simp*]: *remdups-adj* (*rev xs*) = *rev* (*remdups-adj xs*)
⟨*proof*⟩

lemma *remdups-adj-length*[*simp*]: *length* (*remdups-adj xs*) $\leq$ *length xs*
⟨*proof*⟩

lemma *remdups-adj-length-ge1*[*simp*]: *xs* $\neq$ [] $\Longrightarrow$ *length* (*remdups-adj xs*) $\geq$ *Suc* *0*
⟨*proof*⟩

lemma *remdups-adj-Nil-iff*[*simp*]: *remdups-adj xs* = [] $\longleftrightarrow$ *xs* = []
⟨*proof*⟩

lemma *remdups-adj-set*[*simp*]: *set* (*remdups-adj xs*) = *set xs*
⟨*proof*⟩

lemma *remdups-adj-Cons-alt*[*simp*]: *x* # *tl* (*remdups-adj* (*x* # *xs*)) = *remdups-adj* (*x* # *xs*)
⟨*proof*⟩

lemma *remdups-adj-distinct*: *distinct xs* $\Longrightarrow$ *remdups-adj xs* = *xs*
⟨*proof*⟩

lemma *remdups-adj-append*:
remdups-adj (xs_1 @ *x* # xs_2) = *remdups-adj* (xs_1 @ [*x*]) @ *tl* (*remdups-adj* (*x* # xs_2))
⟨*proof*⟩

lemma *remdups-adj-singleton*:
remdups-adj xs = [*x*] $\Longrightarrow$ *xs* = *replicate* (*length xs*) *x*
⟨*proof*⟩

lemma *remdups-adj-map-injective*:
assumes *inj f*
shows *remdups-adj* (*map f xs*) = *map f* (*remdups-adj xs*)
⟨*proof*⟩

lemma *remdups-adj-replicate*:
remdups-adj (*replicate n x*) = (*if* $n = 0$ *then* [] *else* [*x*])
⟨*proof*⟩

lemma *remdups-upt* [*simp*]: $remdups\ [m..<n] = [m..<n]$
⟨*proof*⟩

67.1.24 *insert*

lemma *in-set-insert* [*simp*]:
$x \in set\ xs \Longrightarrow List.insert\ x\ xs = xs$
⟨*proof*⟩

lemma *not-in-set-insert* [*simp*]:
$x \notin set\ xs \Longrightarrow List.insert\ x\ xs = x\ \#\ xs$
⟨*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*]: $distinct\ (List.insert\ x\ xs) = distinct\ xs$
⟨*proof*⟩

lemma *insert-remdups*:
$List.insert\ x\ (remdups\ xs) = remdups\ (List.insert\ x\ xs)$
⟨*proof*⟩

67.1.25 *List.union*

This is all one should need to know about union:

lemma *set-union*[*simp*]: $set\ (List.union\ xs\ ys) = set\ xs \cup set\ ys$
⟨*proof*⟩

lemma *distinct-union*[*simp*]: $distinct(List.union\ xs\ ys) = distinct\ ys$
⟨*proof*⟩

67.1.26 *find*

lemma *find-None-iff*: $List.find\ P\ xs = None \longleftrightarrow \neg\ (\exists\, x.\ x \in set\ xs \wedge P\ x)$
⟨*proof*⟩

lemma *find-Some-iff*:
$List.find\ P\ xs = Some\ x \longleftrightarrow$
$(\exists\, i{<}length\ xs.\ P\ (xs!i) \wedge x = xs!i \wedge (\forall\, j{<}i.\ \neg\ P\ (xs!j)))$
⟨*proof*⟩

lemma *find-cong*[*fundef-cong*]:
assumes $xs = ys$ **and** $\bigwedge x.\ x \in set\ ys \Longrightarrow P\ x = Q\ x$
shows $List.find\ P\ xs = List.find\ Q\ ys$

⟨*proof*⟩

lemma *find-dropWhile*:
$List.find\ P\ xs = (case\ dropWhile\ (Not \circ P)\ xs$
$of\ [] \Rightarrow None$
$\mid x\ \#\ \text{-} \Rightarrow Some\ x)$
⟨*proof*⟩

67.1.27 *count-list*

lemma *count-notin*[*simp*]: $x \notin set\ xs \Longrightarrow count\text{-}list\ xs\ x = 0$
⟨*proof*⟩

lemma *count-le-length*: $count\text{-}list\ xs\ x \leq length\ xs$
⟨*proof*⟩

lemma *sum-count-set*:
$set\ xs \subseteq X \Longrightarrow finite\ X \Longrightarrow sum\ (count\text{-}list\ xs)\ X = length\ xs$
⟨*proof*⟩

67.1.28 *List.extract*

lemma *extract-None-iff*: $List.extract\ P\ xs = None \longleftrightarrow \neg (\exists\ x \in set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *extract-SomeE*:
$List.extract\ P\ xs = Some\ (ys,\ y,\ zs) \Longrightarrow$
$xs = ys\ @\ y\ \#\ zs \wedge P\ y \wedge \neg (\exists\ y \in set\ ys.\ P\ y)$
⟨*proof*⟩

lemma *extract-Some-iff*:
$List.extract\ P\ xs = Some\ (ys,\ y,\ zs) \longleftrightarrow$
$xs = ys\ @\ y\ \#\ zs \wedge P\ y \wedge \neg (\exists\ y \in set\ ys.\ P\ y)$
⟨*proof*⟩

lemma *extract-Nil-code*[*code*]: $List.extract\ P\ [] = None$
⟨*proof*⟩

lemma *extract-Cons-code*[*code*]:
$List.extract\ P\ (x\ \#\ xs) = (if\ P\ x\ then\ Some\ ([],\ x,\ xs)\ else$
$(case\ List.extract\ P\ xs\ of$
$None \Rightarrow None \mid$
$Some\ (ys,\ y,\ zs) \Rightarrow Some\ (x\#ys,\ y,\ zs)))$
⟨*proof*⟩

67.1.29 *remove1*

lemma *remove1-append*:
$remove1\ x\ (xs\ @\ ys) =$
$(if\ x \in set\ xs\ then\ remove1\ x\ xs\ @\ ys\ else\ xs\ @\ remove1\ x\ ys)$

⟨proof⟩

lemma *remove1-commute*: *remove1 x (remove1 y zs) = remove1 y (remove1 x zs)*
⟨proof⟩

lemma *in-set-remove1*[*simp*]:
$a \neq b \Longrightarrow a : set(remove1\ b\ xs) = (a : set\ xs)$
⟨proof⟩

lemma *set-remove1-subset*: *set(remove1 x xs) <= set xs*
⟨proof⟩

lemma *set-remove1-eq* [*simp*]: *distinct xs ==> set(remove1 x xs) = set xs* $- \{x\}$
⟨proof⟩

lemma *length-remove1*:
length(remove1 x xs) = (if x : set xs then length xs − 1 else length xs)
⟨proof⟩

lemma *remove1-filter-not*[*simp*]:
$\neg\ P\ x \Longrightarrow remove1\ x\ (filter\ P\ xs) = filter\ P\ xs$
⟨proof⟩

lemma *filter-remove1*:
filter Q (remove1 x xs) = remove1 x (filter Q 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*:
$distinct\ xs \Longrightarrow remove1\ x\ (remdups\ xs) = remdups\ (remove1\ x\ xs)$
⟨proof⟩

lemma *remove1-idem*: $x \notin set\ xs \Longrightarrow remove1\ x\ xs = xs$
⟨proof⟩

67.1.30 *removeAll*

lemma *removeAll-filter-not-eq*:
$removeAll\ x = filter\ (\lambda y.\ x \neq y)$
⟨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*]: $x \notin set\ xs \Longrightarrow removeAll\ x\ xs = xs$
⟨*proof*⟩

lemma *removeAll-filter-not*[*simp*]:
$\neg\ P\ x \Longrightarrow removeAll\ x\ (filter\ P\ xs) = filter\ P\ xs$
⟨*proof*⟩

lemma *distinct-removeAll*:
$distinct\ xs \Longrightarrow distinct\ (removeAll\ x\ xs)$
⟨*proof*⟩

lemma *distinct-remove1-removeAll*:
distinct xs ==> remove1 x xs = removeAll x xs
⟨*proof*⟩

lemma *map-removeAll-inj-on*: $inj\text{-}on\ f\ (insert\ x\ (set\ xs)) \Longrightarrow$
$map\ f\ (removeAll\ x\ xs) = removeAll\ (f\ x)\ (map\ f\ xs)$
⟨*proof*⟩

lemma *map-removeAll-inj*: $inj\ f \Longrightarrow$
$map\ f\ (removeAll\ x\ xs) = removeAll\ (f\ x)\ (map\ f\ xs)$
⟨*proof*⟩

lemma *length-removeAll-less-eq* [*simp*]:
$length\ (removeAll\ x\ xs) \leq length\ xs$
⟨*proof*⟩

lemma *length-removeAll-less* [*termination-simp*]:
$x \in set\ xs \Longrightarrow length\ (removeAll\ x\ xs) < length\ xs$
⟨*proof*⟩

67.1.31 *replicate*

lemma *length-replicate* [*simp*]: $length\ (replicate\ n\ x) = n$
⟨*proof*⟩

lemma *replicate-eqI*:
assumes $length\ xs = n$ **and** $\bigwedge y.\ y \in set\ xs \Longrightarrow y = x$
shows $xs = replicate\ n\ x$
⟨*proof*⟩

lemma *Ex-list-of-length*: $\exists xs.\ length\ xs = n$
⟨*proof*⟩

lemma *map-replicate* [*simp*]: $map\ f\ (replicate\ n\ x) = replicate\ n\ (f\ x)$
⟨*proof*⟩

lemma *map-replicate-const*:
$map\ (\lambda\ x.\ k)\ lst = replicate\ (length\ lst)\ k$
⟨*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*]: $n \neq 0 ==> hd\ (replicate\ n\ x) = x$
⟨*proof*⟩

lemma *tl-replicate* [*simp*]: $tl\ (replicate\ n\ x) = replicate\ (n - 1)\ x$
⟨*proof*⟩

lemma *last-replicate* [*simp*]: $n \neq 0 ==> last\ (replicate\ n\ x) = x$
⟨*proof*⟩

lemma *nth-replicate*[*simp*]: $i < n ==> (replicate\ n\ x)!i = x$
⟨*proof*⟩

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[*simp*]: $take\ i\ (replicate\ k\ x) = replicate\ (min\ i\ k)\ x$
⟨*proof*⟩

lemma *drop-replicate*[*simp*]: $drop\ i\ (replicate\ k\ x) = replicate\ (k-i)\ x$
⟨*proof*⟩

lemma *set-replicate-Suc*: $set\ (replicate\ (Suc\ n)\ x) = \{x\}$
⟨*proof*⟩

lemma *set-replicate* [*simp*]: $n \neq 0 ==> set\ (replicate\ n\ x) = \{x\}$
⟨*proof*⟩

lemma *set-replicate-conv-if*: $set\ (replicate\ n\ x) = (if\ n = 0\ then\ \{\}\ else\ \{x\})$
⟨*proof*⟩

lemma *in-set-replicate*[*simp*]: $(x : set\ (replicate\ n\ y)) = (x = y\ \&\ n \neq 0)$
⟨*proof*⟩

lemma *Ball-set-replicate*[*simp*]:
$(ALL\ x : set(replicate\ n\ a).\ P\ x) = (P\ a\ |\ n{=}0)$
⟨*proof*⟩

lemma *Bex-set-replicate*[*simp*]:
$(EX\ x : set(replicate\ n\ a).\ P\ x) = (P\ a\ \&\ n{\neq}0)$
⟨*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*]: $([] = replicate\ n\ x) \longleftrightarrow n{=}0$
⟨*proof*⟩

lemma *replicate-eq-replicate*[*simp*]:
$(replicate\ m\ x = replicate\ n\ y) \longleftrightarrow (m{=}n\ \&\ (m{\neq}0 \longrightarrow x{=}y))$
⟨*proof*⟩

lemma *replicate-length-filter*:
$replicate\ (length\ (filter\ (\lambda y.\ x = y)\ xs))\ x = filter\ (\lambda y.\ x = y)\ xs$
⟨*proof*⟩

lemma *comm-append-are-replicate*:
fixes $xs\ ys :: 'a\ list$
assumes $xs \neq []\ ys \neq []$
assumes $xs\ @\ ys = ys\ @\ xs$
shows $\exists\, m\ n\ zs.\ concat\ (replicate\ m\ zs) = xs \wedge concat\ (replicate\ n\ zs) = ys$

⟨proof⟩

lemma *comm-append-is-replicate*:
fixes $xs\ ys :: 'a\ list$
assumes $xs \neq []\ ys \neq []$
assumes $xs\ @\ ys = ys\ @\ xs$
shows $\exists\, n\ zs.\ n > 1 \wedge concat\ (replicate\ n\ zs) = xs\ @\ ys$

⟨proof⟩

lemma *Cons-replicate-eq*:
$x\ \#\ xs = replicate\ n\ y \longleftrightarrow x = y \wedge n > 0 \wedge xs = replicate\ (n - 1)\ x$
⟨proof⟩

lemma *replicate-length-same*:
$(\forall\, y{\in}set\ xs.\ y = x) \Longrightarrow replicate\ (length\ xs)\ x = xs$
⟨proof⟩

lemma *foldr-replicate* [*simp*]:
$foldr\ f\ (replicate\ n\ x) = f\ x\ \hat{}\hat{}\ n$
⟨proof⟩

lemma *fold-replicate* [*simp*]:
$fold\ f\ (replicate\ n\ x) = f\ x\ \hat{}\hat{}\ n$
⟨proof⟩

67.1.32 *enumerate*

lemma *enumerate-simps* [*simp*, *code*]:
$enumerate\ n\ [] = []$
$enumerate\ n\ (x\ \#\ xs) = (n,\ x)\ \#\ enumerate\ (Suc\ n)\ xs$
⟨proof⟩

lemma *length-enumerate* [*simp*]:
$length\ (enumerate\ n\ xs) = length\ xs$
⟨proof⟩

lemma *map-fst-enumerate* [*simp*]:
$map\ fst\ (enumerate\ n\ xs) = [n..{<}n + length\ xs]$
⟨proof⟩

lemma *map-snd-enumerate* [*simp*]:
$map\ snd\ (enumerate\ n\ xs) = xs$
⟨proof⟩

lemma *in-set-enumerate-eq*:
$p \in set\ (enumerate\ n\ xs) \longleftrightarrow n \leq fst\ p \wedge fst\ p < length\ xs + n \wedge nth\ xs\ (fst\ p - n) = snd\ p$
⟨proof⟩

lemma *nth-enumerate-eq*:
assumes $m < length\ xs$
shows $enumerate\ n\ xs\ !\ m = (n + m,\ xs\ !\ m)$
⟨*proof*⟩

lemma *enumerate-replicate-eq*:
$enumerate\ n\ (replicate\ m\ a) = map\ (\lambda q.\ (q,\ a))\ [n..<n + m]$
⟨*proof*⟩

lemma *enumerate-Suc-eq*:
$enumerate\ (Suc\ n)\ xs = map\ (apfst\ Suc)\ (enumerate\ n\ xs)$
⟨*proof*⟩

lemma *distinct-enumerate* [*simp*]:
$distinct\ (enumerate\ n\ xs)$
⟨*proof*⟩

lemma *enumerate-append-eq*:
$enumerate\ n\ (xs\ @\ ys) = enumerate\ n\ xs\ @\ enumerate\ (n + length\ xs)\ ys$
⟨*proof*⟩

lemma *enumerate-map-upt*:
$enumerate\ n\ (map\ f\ [n..<m]) = map\ (\lambda k.\ (k,\ f\ k))\ [n..<m]$
⟨*proof*⟩

67.1.33 *rotate1* and *rotate*

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*]: $length\ xs <= 1 \Longrightarrow rotate1\ xs = xs$
⟨*proof*⟩

lemma *rotate-length01*[*simp*]: $length\ xs <= 1 \Longrightarrow rotate\ n\ xs = xs$
⟨*proof*⟩

lemma *rotate1-hd-tl*: $xs \neq [] \Longrightarrow rotate1\ xs = tl\ xs\ @\ [hd\ xs]$
⟨*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*]: $n\ mod\ length\ xs = 0 \Longrightarrow rotate\ n\ xs = xs$
⟨*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*: $xs \neq [] \Longrightarrow hd(rotate\ n\ xs) = xs!(n\ mod\ length\ xs)$
⟨*proof*⟩

67.1.34 *nths* — a generalization of *op* ! to sets

lemma *nths-empty* [*simp*]: $nths\ xs\ \{\} = []$
⟨*proof*⟩

lemma *nths-nil* [*simp*]: $nths\ []\ A = []$
⟨*proof*⟩

lemma *length-nths*:
$length\ (nths\ xs\ I) = card\{i.\ i < length\ xs \wedge i : I\}$
⟨*proof*⟩

lemma *nths-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 *nths-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 *nths-append*:
$nths\ (l\ @\ l')\ A = nths\ l\ A\ @\ nths\ l'\ \{j.\ j + length\ l : A\}$
⟨*proof*⟩

lemma *nths-Cons*:
$nths\ (x\ \#\ l)\ A = (if\ 0{:}A\ then\ [x]\ else\ [])\ @\ nths\ l\ \{j.\ Suc\ j : A\}$
⟨*proof*⟩

lemma *set-nths*: $set(nths\ xs\ I) = \{xs!i|i.\ i{<}size\ xs \wedge i \in I\}$
⟨*proof*⟩

lemma *set-nths-subset*: $set(nths\ xs\ I) \subseteq set\ xs$
⟨*proof*⟩

lemma *notin-set-nthsI*[*simp*]: $x \notin set\ xs \Longrightarrow x \notin set(nths\ xs\ I)$
⟨*proof*⟩

lemma *in-set-nthsD*: $x \in set(nths\ xs\ I) \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *nths-singleton* [*simp*]: $nths\ [x]\ A = (if\ 0 : A\ then\ [x]\ else\ [])$
⟨*proof*⟩

lemma *distinct-nthsI*[*simp*]: $distinct\ xs \Longrightarrow distinct\ (nths\ xs\ I)$
⟨*proof*⟩

lemma *nths-upt-eq-take* [*simp*]: *nths l* $\{..<n\}$ = *take n l*
⟨*proof*⟩

lemma *filter-in-nths*:
distinct xs $\Longrightarrow$ *filter* (%*x*. *x* $\in$ *set* (*nths xs s*)) *xs* = *nths xs s*
⟨*proof*⟩

67.1.35 *subseqs* and *List.n-lists*

lemma *length-subseqs*: *length* (*subseqs xs*) = *2* ^ *length xs*
⟨*proof*⟩

lemma *subseqs-powset*: *set* ‘ *set* (*subseqs xs*) = *Pow* (*set xs*)
⟨*proof*⟩

lemma *distinct-set-subseqs*:
assumes *distinct xs*
shows *distinct* (*map set* (*subseqs xs*))
⟨*proof*⟩

lemma *n-lists-Nil* [*simp*]: *List.n-lists n* [] = (*if n* = *0 then* [[]] *else* [])
⟨*proof*⟩

lemma *length-n-lists-elem*: *ys* $\in$ *set* (*List.n-lists n xs*) $\Longrightarrow$ *length ys* = *n*
⟨*proof*⟩

lemma *set-n-lists*: *set* (*List.n-lists n xs*) = $\{$*ys*. *length ys* = *n* $\wedge$ *set ys* $\subseteq$ *set xs*$\}$
⟨*proof*⟩

lemma *subseqs-refl*: *xs* $\in$ *set* (*subseqs xs*)
⟨*proof*⟩

lemma *subset-subseqs*: *X* $\subseteq$ *set xs* $\Longrightarrow$ *X* $\in$ *set* ‘ *set* (*subseqs xs*)
⟨*proof*⟩

lemma *Cons-in-subseqsD*: *y* # *ys* $\in$ *set* (*subseqs xs*) $\Longrightarrow$ *ys* $\in$ *set* (*subseqs xs*)
⟨*proof*⟩

lemma *subseqs-distinctD*: $[\![$ *ys* $\in$ *set* (*subseqs xs*); *distinct xs* $]\!]$ $\Longrightarrow$ *distinct ys*
⟨*proof*⟩

67.1.36 *splice*

lemma *splice-Nil2* [*simp*, *code*]: *splice xs* [] = *xs*
⟨*proof*⟩

declare *splice.simps*(*1*,*3*)[*code*]
declare *splice.simps*(*2*)[*simp del*]

lemma *length-splice*[*simp*]: *length*(*splice xs ys*) = *length xs* + *length ys*

$\langle proof \rangle$

67.1.37 *shuffle*

lemma *Nil-in-shuffle*[*simp*]: $[] \in shuffle\ xs\ ys \longleftrightarrow xs = [] \wedge ys = []$
$\langle proof \rangle$

lemma *shuffleE*:
$zs \in shuffle\ xs\ ys \Longrightarrow$
$(zs = xs \Longrightarrow ys = [] \Longrightarrow P) \Longrightarrow$
$(zs = ys \Longrightarrow xs = [] \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ xs'\ z\ zs'.\ xs = x\ \#\ xs' \Longrightarrow zs = z\ \#\ zs' \Longrightarrow x = z \Longrightarrow zs' \in shuffle\ xs'$
$ys \Longrightarrow P) \Longrightarrow$
$(\bigwedge y\ ys'\ z\ zs'.\ ys = y\ \#\ ys' \Longrightarrow zs = z\ \#\ zs' \Longrightarrow y = z \Longrightarrow zs' \in shuffle\ xs$
$ys' \Longrightarrow P) \Longrightarrow P$
$\langle proof \rangle$

lemma *Cons-in-shuffle-iff*:
$z\ \#\ zs \in shuffle\ xs\ ys \longleftrightarrow$
$(xs \neq [] \wedge hd\ xs = z \wedge zs \in shuffle\ (tl\ xs)\ ys \vee$
$ys \neq [] \wedge hd\ ys = z \wedge zs \in shuffle\ xs\ (tl\ ys))$
$\langle proof \rangle$

lemma *splice-in-shuffle* [*simp*, *intro*]: $splice\ xs\ ys \in shuffle\ xs\ ys$
$\langle proof \rangle$

lemma *Nil-in-shuffleI*: $xs = [] \Longrightarrow ys = [] \Longrightarrow [] \in shuffle\ xs\ ys$
$\langle proof \rangle$

lemma *Cons-in-shuffle-leftI*: $zs \in shuffle\ xs\ ys \Longrightarrow z\ \#\ zs \in shuffle\ (z\ \#\ xs)\ ys$
$\langle proof \rangle$

lemma *Cons-in-shuffle-rightI*: $zs \in shuffle\ xs\ ys \Longrightarrow z\ \#\ zs \in shuffle\ xs\ (z\ \#\ ys)$
$\langle proof \rangle$

lemma *finite-shuffle* [*simp*, *intro*]: $finite\ (shuffle\ xs\ ys)$
$\langle proof \rangle$

lemma *length-shuffle*: $zs \in shuffle\ xs\ ys \Longrightarrow length\ zs = length\ xs + length\ ys$
$\langle proof \rangle$

lemma *set-shuffle*: $zs \in shuffle\ xs\ ys \Longrightarrow set\ zs = set\ xs \cup set\ ys$
$\langle proof \rangle$

lemma *distinct-disjoint-shuffle*:
assumes $distinct\ xs\ distinct\ ys\ set\ xs \cap set\ ys = \{\}\ zs \in shuffle\ xs\ ys$
shows $distinct\ zs$
$\langle proof \rangle$

lemma *shuffle-commutes*: *shuffle xs ys* = *shuffle ys xs*
⟨*proof*⟩

lemma *Cons-shuffle-subset1*: *op* # *x* ' *shuffle xs ys* ⊆ *shuffle* (*x* # *xs*) *ys*
⟨*proof*⟩

lemma *Cons-shuffle-subset2*: *op* # *y* ' *shuffle xs ys* ⊆ *shuffle xs* (*y* # *ys*)
⟨*proof*⟩

lemma *filter-shuffle*:
filter P ' *shuffle xs ys* = *shuffle* (*filter P xs*) (*filter P ys*)
⟨*proof*⟩

lemma *filter-shuffle-disjoint1*:
assumes *set xs* ∩ *set ys* = {} *zs* ∈ *shuffle xs ys*
shows *filter* (λ*x*. *x* ∈ *set xs*) *zs* = *xs* (**is** *filter ?P* - = -)
and *filter* (λ*x*. *x* ∉ *set xs*) *zs* = *ys* (**is** *filter ?Q* - = -)
⟨*proof*⟩

lemma *filter-shuffle-disjoint2*:
assumes *set xs* ∩ *set ys* = {} *zs* ∈ *shuffle xs ys*
shows *filter* (λ*x*. *x* ∈ *set ys*) *zs* = *ys* *filter* (λ*x*. *x* ∉ *set ys*) *zs* = *xs*
⟨*proof*⟩

lemma *partition-in-shuffle*:
xs ∈ *shuffle* (*filter P xs*) (*filter* (λ*x*. ¬*P x*) *xs*)
⟨*proof*⟩

lemma *inv-image-partition*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *P x* ⋀*y*. *y* ∈ *set ys* ⟹ ¬*P y*
shows *partition P* −' {(*xs*, *ys*)} = *shuffle xs ys*
⟨*proof*⟩

67.1.38 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* (*case-list* [] (λ*h t*. [*h*])) *xss*) =
map (λ*xs*. *hd xs*) [*ys*←*xss* . *ys* ≠ []]
⟨*proof*⟩

lemma *transpose-aux-filter-tail*:
concat (*map* (*case-list* [] (λ*h t*. [*t*])) *xss*) =

$map\ (\lambda xs.\ tl\ xs)\ [ys \leftarrow xss\ .\ ys \neq []]$
⟨proof⟩

lemma *transpose-aux-max*:
$max\ (Suc\ (length\ xs))\ (foldr\ (\lambda xs.\ max\ (length\ xs))\ xss\ 0) =$
$Suc\ (max\ (length\ xs)\ (foldr\ (\lambda x.\ max\ (length\ x - Suc\ 0))\ [ys \leftarrow xss\ .\ ys \neq []]\ 0))$
(**is** max - $?foldB = Suc\ (max$ - $?foldA)$)
⟨proof⟩

termination *transpose*
⟨proof⟩

lemma *transpose-empty*: $(transpose\ xs = []) \longleftrightarrow (\forall x \in set\ xs.\ x = [])$
⟨proof⟩

lemma *length-transpose*:
fixes $xs :: 'a\ list\ list$
shows $length\ (transpose\ xs) = foldr\ (\lambda xs.\ max\ (length\ xs))\ xs\ 0$
⟨proof⟩

lemma *nth-transpose*:
fixes $xs :: 'a\ list\ list$
assumes $i < length\ (transpose\ xs)$
shows $transpose\ xs\ !\ i = map\ (\lambda xs.\ xs\ !\ i)\ [ys \leftarrow xs.\ i < length\ ys]$
⟨proof⟩

lemma *transpose-map-map*:
$transpose\ (map\ (map\ f)\ xs) = map\ (map\ f)\ (transpose\ xs)$
⟨proof⟩

67.1.39 (In)finiteness

lemma *finite-maxlen*:
$finite\ (M::'a\ list\ set) ==> EX\ n.\ ALL\ s{:}M.\ size\ s < n$
⟨proof⟩

lemma *lists-length-Suc-eq*:
$\{xs.\ set\ xs \subseteq A \wedge length\ xs = Suc\ n\} =$
$(\lambda(xs,\ n).\ n\#xs)\ `\ (\{xs.\ set\ xs \subseteq A \wedge length\ xs = n\} \times A)$
⟨proof⟩

lemma
assumes $finite\ A$
shows *finite-lists-length-eq*: $finite\ \{xs.\ set\ xs \subseteq A \wedge length\ xs = n\}$
and *card-lists-length-eq*: $card\ \{xs.\ set\ xs \subseteq A \wedge length\ xs = n\} = (card\ A)\hat{}n$
⟨proof⟩

lemma *finite-lists-length-le*:
assumes $finite\ A$ **shows** $finite\ \{xs.\ set\ xs \subseteq A \wedge length\ xs \leq n\}$

(**is** *finite ?S*)
⟨*proof*⟩

lemma *card-lists-length-le*:
assumes *finite A* **shows** $card\ \{xs.\ set\ xs \subseteq A \wedge length\ xs \leq n\} = (\sum i{\leq}n.\ card\ A\,\hat{}\,i)$
⟨*proof*⟩

lemma *card-lists-distinct-length-eq*:
assumes $finite\ A\ k \leq card\ A$
shows $card\ \{xs.\ length\ xs = k \wedge distinct\ xs \wedge set\ xs \subseteq A\} = \prod\{card\ A - k + 1\ ..\ card\ A\}$
⟨*proof*⟩

lemma *card-lists-distinct-length-eq'*:
assumes $k < card\ A$
shows $card\ \{xs.\ length\ xs = k \wedge distinct\ xs \wedge set\ xs \subseteq A\} = \prod\{card\ A - k + 1\ ..\ card\ A\}$
⟨*proof*⟩

lemma *infinite-UNIV-listI*: $\sim finite(UNIV{::}'a\ list\ set)$
⟨*proof*⟩

67.2 Sorting

67.2.1 *sorted-wrt*

lemma *sorted-wrt-induct*:
$[\![P\ [];\ \bigwedge x.\ P\ [x];\ \bigwedge x\ y\ zs.\ P\ (y\ \#\ zs) \Longrightarrow P\ (x\ \#\ y\ \#\ zs)]\!] \Longrightarrow P\ xs$
⟨*proof*⟩

lemma *sorted-wrt-Cons*:
assumes *transp P*
shows $sorted\text{-}wrt\ P\ (x\ \#\ xs) = ((\forall y \in set\ xs.\ P\ x\ y) \wedge sorted\text{-}wrt\ P\ xs)$
⟨*proof*⟩

lemma *sorted-wrt-ConsI*:
$[\![\ transp\ P;\ \bigwedge y.\ y \in set\ xs \Longrightarrow P\ x\ y;\ sorted\text{-}wrt\ P\ xs\ ]\!] \Longrightarrow$
$sorted\text{-}wrt\ P\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *sorted-wrt-append*:
assumes *transp P*
shows $sorted\text{-}wrt\ P\ (xs\ @\ ys) \longleftrightarrow$
$sorted\text{-}wrt\ P\ xs \wedge sorted\text{-}wrt\ P\ ys \wedge (\forall x{\in}set\ xs.\ \forall y{\in}set\ ys.\ P\ x\ y)$
⟨*proof*⟩

lemma *sorted-wrt-rev*: **assumes** *transp P*
shows $sorted\text{-}wrt\ P\ (rev\ xs) = sorted\text{-}wrt\ (\lambda x\ y.\ P\ y\ x)\ xs$
⟨*proof*⟩

lemma *sorted-wrt-mono*:
$(\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow \mathit{sorted\text{-}wrt}\ P\ xs \Longrightarrow \mathit{sorted\text{-}wrt}\ Q\ xs$
⟨*proof*⟩

Strictly Ascending Sequences of Natural Numbers

lemma *sorted-wrt-upt*[*simp*]: $\mathit{sorted\text{-}wrt}\ (op <)\ [0..<n]$
⟨*proof*⟩

Each element is greater or equal to its index:

lemma *sorted-wrt-less-idx*:
$\mathit{sorted\text{-}wrt}\ (op <)\ ns \Longrightarrow i < \mathit{length}\ ns \Longrightarrow i \le ns!i$
⟨*proof*⟩

67.2.2 *sorted*

context *linorder*
begin

lemma *sorted-Cons*: $\mathit{sorted}\ (x\#xs) = (\mathit{sorted}\ xs \wedge (\forall y \in \mathit{set}\ xs.\ x \le y))$
⟨*proof*⟩

lemma *sorted-iff-wrt*: $\mathit{sorted}\ xs = \mathit{sorted\text{-}wrt}\ (op \le)\ xs$
⟨*proof*⟩

lemma *sorted-tl*:
$\mathit{sorted}\ xs \Longrightarrow \mathit{sorted}\ (tl\ xs)$
⟨*proof*⟩

lemma *sorted-append*:
$\mathit{sorted}\ (xs@ys) = (\mathit{sorted}\ xs\ \&\ \mathit{sorted}\ ys\ \&\ (\forall x \in \mathit{set}\ xs.\ \forall y \in \mathit{set}\ ys.\ x{\le}y))$
⟨*proof*⟩

lemma *sorted-nth-mono*:
$\mathit{sorted}\ xs \Longrightarrow i \le j \Longrightarrow j < \mathit{length}\ xs \Longrightarrow xs!i \le xs!j$
⟨*proof*⟩

lemma *sorted-rev-nth-mono*:
$\mathit{sorted}\ (\mathit{rev}\ xs) \Longrightarrow i \le j \Longrightarrow j < \mathit{length}\ xs \Longrightarrow xs!j \le xs!i$
⟨*proof*⟩

lemma *sorted-nth-monoI*:
$(\bigwedge i\ j.\ [\![\ i \le j\ ;\ j < \mathit{length}\ xs\ ]\!] \Longrightarrow xs\ !\ i \le xs\ !\ j) \Longrightarrow \mathit{sorted}\ xs$
⟨*proof*⟩

lemma *sorted-equals-nth-mono*:
$\mathit{sorted}\ xs = (\forall j < \mathit{length}\ xs.\ \forall i \le j.\ xs\ !\ i \le xs\ !\ j)$
⟨*proof*⟩

lemma *sorted-map-remove1*:
sorted (*map f xs*) $\Longrightarrow$ *sorted* (*map f* (*remove1 x xs*))
⟨*proof*⟩

lemma *sorted-remove1*: *sorted xs* $\Longrightarrow$ *sorted* (*remove1 a xs*)
⟨*proof*⟩

lemma *sorted-butlast*:
assumes *xs* $\neq$ [] **and** *sorted xs*
shows *sorted* (*butlast xs*)
⟨*proof*⟩

lemma *sorted-remdups*[*simp*]:
sorted l $\Longrightarrow$ *sorted* (*remdups l*)
⟨*proof*⟩

lemma *sorted-remdups-adj*[*simp*]:
sorted xs $\Longrightarrow$ *sorted* (*remdups-adj xs*)
⟨*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* $\cup$ *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
assumes *sorted xs*
shows *sorted-take*: *sorted* (*take n xs*)
and *sorted-drop*: *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 *sorted-map-same*:
  sorted (*map f* [*x*←*xs*. *f x* = *g xs*])
⟨*proof*⟩

lemma *sorted-same*:
  sorted [*x*←*xs*. *x* = *g xs*]
⟨*proof*⟩

end

67.2.3 Sorting functions

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 *set-insort-key*:
  set (*insort-key f x xs*) = *insert x* (*set xs*)
⟨*proof*⟩

lemma *length-insort* [*simp*]:
  length (*insort-key f x xs*) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *insort-key-left-comm*:
  assumes *f x* ≠ *f y*
  shows *insort-key f y* (*insort-key f x xs*) = *insort-key f x* (*insort-key f y xs*)
⟨*proof*⟩

lemma *insort-left-comm*:
  insort x (*insort y xs*) = *insort y* (*insort x xs*)
⟨*proof*⟩

lemma *comp-fun-commute-insort*: *comp-fun-commute 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-key-conv-fold*:
assumes *inj-on f* (*set xs*)
shows *sort-key f xs* = *fold* (*insort-key f*) *xs* []
⟨*proof*⟩

lemma *sort-conv-fold*:
sort xs = *fold insort xs* []
⟨*proof*⟩

lemma *length-sort*[*simp*]: *length* (*sort-key f xs*) = *length xs*
⟨*proof*⟩

lemma *set-sort*[*simp*]: *set*(*sort-key f xs*) = *set xs*
⟨*proof*⟩

lemma *distinct-insort*: *distinct* (*insort-key f x xs*) = ($x \notin$ *set xs* $\wedge$ *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 *insort-not-Nil* [*simp*]:
insort-key f a xs $\neq$ []
⟨*proof*⟩

lemma *insort-is-Cons*: $\forall x \in$*set xs*. $f\ a \leq f\ x \Longrightarrow$ *insort-key f a xs* = *a* # *xs*
⟨*proof*⟩

lemma *sorted-sort-id*: *sorted xs* $\Longrightarrow$ *sort xs* = *xs*

⟨*proof*⟩

lemma *insort-key-remove1*:
assumes $a \in set\ xs$ **and** *sorted* $(map\ f\ xs)$ **and** $hd\ (filter\ (\lambda x.\ f\ a = f\ x)\ xs) = a$
shows $insort\text{-}key\ f\ a\ (remove1\ a\ xs) = xs$
⟨*proof*⟩

lemma *insort-remove1*:
assumes $a \in set\ xs$ **and** *sorted xs*
shows $insort\ a\ (remove1\ a\ xs) = xs$
⟨*proof*⟩

lemma *finite-sorted-distinct-unique*:
shows $finite\ A \Longrightarrow \exists!xs.\ set\ xs = A \wedge sorted\ xs \wedge distinct\ xs$
⟨*proof*⟩

lemma *insort-insert-key-triv*:
$f\ x \in f\ `\ set\ xs \Longrightarrow insort\text{-}insert\text{-}key\ f\ x\ xs = xs$
⟨*proof*⟩

lemma *insort-insert-triv*:
$x \in set\ xs \Longrightarrow insort\text{-}insert\ x\ xs = xs$
⟨*proof*⟩

lemma *insort-insert-insort-key*:
$f\ x \notin f\ `\ set\ xs \Longrightarrow insort\text{-}insert\text{-}key\ f\ x\ xs = insort\text{-}key\ f\ x\ xs$
⟨*proof*⟩

lemma *insort-insert-insort*:
$x \notin set\ xs \Longrightarrow insort\text{-}insert\ x\ xs = insort\ x\ xs$
⟨*proof*⟩

lemma *set-insort-insert*:
$set\ (insort\text{-}insert\ x\ xs) = insert\ x\ (set\ xs)$
⟨*proof*⟩

lemma *distinct-insort-insert*:
assumes *distinct xs*
shows $distinct\ (insort\text{-}insert\text{-}key\ f\ x\ xs)$
⟨*proof*⟩

lemma *sorted-insort-insert-key*:
assumes $sorted\ (map\ f\ xs)$
shows $sorted\ (map\ f\ (insort\text{-}insert\text{-}key\ f\ x\ xs))$
⟨*proof*⟩

lemma *sorted-insort-insert*:
assumes *sorted xs*

shows *sorted* (*insort-insert x xs*)
⟨*proof*⟩

lemma *filter-insort-triv*:
$\neg$ *P x* $\Longrightarrow$ *filter P* (*insort-key f x xs*) = *filter P xs*
⟨*proof*⟩

lemma *filter-insort*:
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*:
filter P (*sort-key f xs*) = *sort-key f* (*filter P xs*)
⟨*proof*⟩

lemma *remove1-insort* [*simp*]:
remove1 x (*insort x xs*) = *xs*
⟨*proof*⟩

end

lemma *sorted-upt*[*simp*]: *sorted*[*i..<j*]
⟨*proof*⟩

lemma *sort-upt* [*simp*]:
sort [*m..<n*] = [*m..<n*]
⟨*proof*⟩

lemma *sorted-upto*[*simp*]: *sorted*[*i..j*]
⟨*proof*⟩

lemma *sorted-find-Min*:
assumes *sorted xs*
assumes $\exists x \in$ *set xs. P x*
shows *List.find P xs* = *Some* (*Min* {*x*∈*set xs. P x*})
⟨*proof*⟩

lemma *sorted-enumerate* [*simp*]:
sorted (*map fst* (*enumerate n xs*))
⟨*proof*⟩

67.2.4 *transpose* on sorted lists

lemma *sorted-transpose*[*simp*]:
shows *sorted* (*rev* (*map length* (*transpose xs*)))
⟨*proof*⟩

lemma *transpose-max-length*:

foldr $(\lambda xs.\ max\ (length\ xs))\ (transpose\ xs)\ 0 = length\ [x \leftarrow xs.\ x \neq []]$
(**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: $j < length\ [ys \leftarrow xs.\ i < length\ ys]$
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*⟩

67.2.5 *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* $\Rightarrow$ $'a$ *list* **where**
sorted-list-of-set = *folding.F insort* []

sublocale *sorted-list-of-set*: *folding insort Nil*
rewrites
folding.F insort [] = *sorted-list-of-set*
⟨*proof*⟩

lemma *sorted-list-of-set-empty*:
sorted-list-of-set {} = []
⟨*proof*⟩

lemma *sorted-list-of-set-insert* [*simp*]:
finite A $\Longrightarrow$ *sorted-list-of-set* (*insert x A*) = *insort x* (*sorted-list-of-set* (A − {x}))
⟨*proof*⟩

lemma *sorted-list-of-set-eq-Nil-iff* [*simp*]:
finite A $\Longrightarrow$ *sorted-list-of-set A* = [] $\longleftrightarrow$ A = {}
⟨*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 *distinct-sorted-list-of-set*:
distinct (*sorted-list-of-set A*)
⟨*proof*⟩

lemma *sorted-list-of-set-sort-remdups* [*code*]:
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*⟩

67.2.6 *lists*: the list-forming operator over sets

inductive-set
lists :: *'a set* => *'a list set*
for *A* :: *'a set*
where
Nil [*intro*!, *simp*]: []: *lists A*
| *Cons* [*intro*!, *simp*]: [| *a*: *A*; *l*: *lists A*|] ==> *a*#*l* : *lists A*

inductive-cases *listsE* [*elim*!]: *x*#*l* : *lists A*
inductive-cases *listspE* [*elim*!]: *listsp A* (*x* # *l*)

inductive-simps *listsp-simps*[*code*]:
listsp A []
listsp A (*x* # *xs*)

lemma *listsp-mono* [*mono*]: $A \leq B$ ==> *listsp A* $\leq$ *listsp B*
⟨*proof*⟩

lemmas *lists-mono* = *listsp-mono* [*to-set*]

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-def inf-bool-def*]

lemmas *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set*]

lemma *Cons-in-lists-iff* [*simp*]: *x*#*xs* : *lists A* $\longleftrightarrow$ *x*:*A* $\wedge$ *xs* : *lists A*
⟨*proof*⟩

lemma *append-in-listsp-conv* [*iff*]:
(*listsp A* (*xs* @ *ys*)) = (*listsp A xs* $\wedge$ *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* [*code-unfold*] = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest*!]: *listsp A xs* ==> $\forall$ *x*$\in$*set xs*. *A x*
⟨*proof*⟩

lemmas *in-listsD* [*dest*!] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro*!]: $\forall x \in set\ xs.\ A\ x$ ==> *listsp A xs*
⟨*proof*⟩

lemmas *in-listsI* [*intro*!] = *in-listspI* [*to-set*]

lemma *lists-eq-set*: *lists A* = {*xs. set xs* <= *A*}
⟨*proof*⟩

lemma *lists-empty* [*simp*]: *lists* {} = {[]}
⟨*proof*⟩

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
⟨*proof*⟩

lemma *lists-image*: *lists* (*f*‘*A*) = *map f* ‘ *lists A*
⟨*proof*⟩

67.2.7 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*⟩

67.2.8 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**
set-Cons A XS = $\{z.\ \exists x\ xs.\ z = x \# xs \land x \in A \land xs \in XS\}$

lemma *set-Cons-sing-Nil* [*simp*]: *set-Cons A* {[]} = (%*x*. [*x*])‘*A*
⟨*proof*⟩

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

primrec *listset* :: $'a\ set\ list \Rightarrow 'a\ list\ set$ **where**
listset [] = {[]} |
listset (*A* # *As*) = *set-Cons A* (*listset As*)

67.3 Relations on Lists

67.3.1 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*) =
(*map-prod* (%(*x*, *xs*). *x#xs*) (%(*x*, *xs*). *x#xs*) ' (*r* <*lex*> *lexn r n*)) *Int*
{(*xs*, *ys*). *length xs* = *Suc n* $\wedge$ *length ys* = *Suc n*}

definition *lex* :: $('a \times 'a)$ *set* $\Rightarrow$ $('a\ list \times 'a\ list)$ *set* **where**
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**
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* $\wedge$ *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* $\wedge$ *length ys* = *n* $\wedge$
($\exists$ *xys x y xs' ys'*. *xs*= *xys* @ *x#xs'* $\wedge$ *ys*= *xys* @ *y* # *ys'* $\wedge$ (*x*, *y*):*r*)}
⟨*proof*⟩

By Mathias Fleury:

lemma *lexn-transI*:
assumes *trans r* **shows** *trans* (*lexn r n*)
⟨*proof*⟩

lemma *lex-conv*:
lex r =
{(*xs*,*ys*). *length xs* = *length ys* $\wedge$
($\exists$ *xys x y xs' ys'*. *xs* = *xys* @ *x* # *xs'* $\wedge$ *ys* = *xys* @ *y* # *ys'* $\wedge$ (*x*, *y*):*r*)}
⟨*proof*⟩

lemma *wf-lenlex* [*intro*!]: *wf r* ==> *wf* (*lenlex r*)
⟨*proof*⟩

lemma *lenlex-conv*:
$lenlex\ r = \{(xs,ys).\ length\ xs < length\ ys\ |$
$length\ xs = length\ ys \wedge (xs,\ ys) : lex\ r\}$
⟨*proof*⟩

lemma *Nil-notin-lex* [*iff*]: $([],\ ys) \notin lex\ r$
⟨*proof*⟩

lemma *Nil2-notin-lex* [*iff*]: $(xs,\ []) \notin lex\ r$
⟨*proof*⟩

lemma *Cons-in-lex* [*simp*]:
$((x\ \#\ xs,\ y\ \#\ ys) : lex\ r) =$
$((x,\ y) : r \wedge length\ xs = length\ ys\ |\ x = y \wedge (xs,\ ys) : lex\ r)$
⟨*proof*⟩

lemma *lex-append-rightI*:
$(xs,\ ys) \in lex\ r \Longrightarrow length\ vs = length\ us \Longrightarrow (xs\ @\ us,\ ys\ @\ vs) \in lex\ r$
⟨*proof*⟩

lemma *lex-append-leftI*:
$(ys,\ zs) \in lex\ r \Longrightarrow (xs\ @\ ys,\ xs\ @\ zs) \in lex\ r$
⟨*proof*⟩

lemma *lex-append-leftD*:
$\forall x.\ (x,x) \notin r \Longrightarrow (xs\ @\ ys,\ xs\ @\ zs) \in lex\ r \Longrightarrow (ys,\ zs) \in lex\ r$
⟨*proof*⟩

lemma *lex-append-left-iff*:
$\forall x.\ (x,x) \notin r \Longrightarrow (xs\ @\ ys,\ xs\ @\ zs) \in lex\ r \longleftrightarrow (ys,\ zs) \in lex\ r$
⟨*proof*⟩

lemma *lex-take-index*:
assumes $(xs,\ ys) \in lex\ r$
obtains i **where** $i < length\ xs$ **and** $i < length\ ys$ **and** $take\ i\ xs =$
$take\ i\ ys$
and $(xs\ !\ i,\ ys\ !\ i) \in r$
⟨*proof*⟩

67.3.2 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**
$lexord\ r = \{(x,y).\ \exists\ a\ v.\ y = x\ @\ a\ \#\ v\ \vee$
$(\exists\ u\ a\ b\ v\ w.\ (a,b) \in r \wedge x = u\ @\ (a\ \#\ v) \wedge y = u\ @\ (b\ \#\ w))\}$

lemma *lexord-Nil-left*[*simp*]: $([],y) \in lexord\ r = (\exists\ a\ x.\ y = a\ \#\ x)$

⟨*proof*⟩

lemma *lexord-Nil-right*[*simp*]: $(x,[]) \notin lexord\ r$
⟨*proof*⟩

lemma *lexord-cons-cons*[*simp*]:
$((a \# x,\ b \# y) \in lexord\ r) = ((a,b)\in r \mid (a = b\ \&\ (x,y)\in lexord\ r))$
⟨*proof*⟩

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-append-rightI*: $\exists\ b\ z.\ y = b \# z \Longrightarrow (x,\ x\ @\ y) \in lexord\ r$
⟨*proof*⟩

lemma *lexord-append-left-rightI*:
$(a,b) \in r \Longrightarrow (u\ @\ a \# x,\ u\ @\ b \# y) \in lexord\ r$
⟨*proof*⟩

lemma *lexord-append-leftI*: $(u,v) \in lexord\ r \Longrightarrow (x\ @\ u,\ x\ @\ v) \in lexord\ r$
⟨*proof*⟩

lemma *lexord-append-leftD*:
$[\![\ (x\ @\ u,\ x\ @\ v) \in lexord\ r;\ (!\ a.\ (a,a) \notin r)\ ]\!] \Longrightarrow (u,v) \in lexord\ r$
⟨*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*: $ALL\ x.\ (x,x) \notin r \Longrightarrow (xs,xs) \notin lexord\ r$
⟨*proof*⟩

By René Thiemann:

lemma *lexord-partial-trans*:
$(\bigwedge x\ y\ z.\ x \in set\ xs \Longrightarrow (x,y) \in r \Longrightarrow (y,z) \in r \Longrightarrow (x,z) \in r)$
$\Longrightarrow (xs,ys) \in lexord\ r \Longrightarrow (ys,zs) \in lexord\ r \Longrightarrow (xs,zs) \in lexord\ r$
⟨*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. $(a,b) \in r$ | $a = b$ | $(b,a) \in r$) $\Longrightarrow$ (x,y) : *lexord* r | x $= y$ | (y,x) : *lexord* r
⟨*proof*⟩

lemma *lexord-irrefl*:
irrefl $R \Longrightarrow$ *irrefl* (*lexord* R)
⟨*proof*⟩

lemma *lexord-asym*:
assumes *asym* R
shows *asym* (*lexord* R)
⟨*proof*⟩

lemma *lexord-asymmetric*:
assumes *asym* R
assumes *hyp*: $(a,\ b) \in$ *lexord* R
shows $(b,\ a) \notin$ *lexord* R
⟨*proof*⟩

Predicate version of lexicographic order integrated with Isabelle's order type classes. Author: Andreas Lochbihler

context *ord*
begin

context
notes [[*inductive-internals*]]
begin

inductive *lexordp* :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *bool*
where
Nil: *lexordp* [] (y # ys)
| *Cons*: $x < y \Longrightarrow$ *lexordp* (x # xs) (y # ys)
| *Cons-eq*:
$[\![\neg\ x < y;\ \neg\ y < x;$ *lexordp* xs ys $]\!] \Longrightarrow$ *lexordp* (x # xs) (y # ys)

end

lemma *lexordp-simps* [*simp*]:
lexordp [] ys = ($ys \neq$ [])
lexordp xs [] = *False*
lexordp (x # xs) (y # ys) $\longleftrightarrow$ $x < y \vee \neg\ y < x \wedge$ *lexordp* xs ys
⟨*proof*⟩

inductive *lexordp-eq* :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *bool* **where**
Nil: *lexordp-eq* [] ys
| *Cons*: $x < y \Longrightarrow$ *lexordp-eq* (x # xs) (y # ys)
| *Cons-eq*: $[\![\neg\ x < y;\ \neg\ y < x;$ *lexordp-eq* xs ys $]\!] \Longrightarrow$ *lexordp-eq* (x # xs) (y # ys)

lemma *lexordp-eq-simps* [*simp*]:
lexordp-eq [] *ys* = *True*
lexordp-eq xs [] $\longleftrightarrow$ *xs* = []
lexordp-eq (*x* # *xs*) [] = *False*
lexordp-eq (*x* # *xs*) (*y* # *ys*) $\longleftrightarrow$ $x < y \vee \neg\, y < x \wedge$ *lexordp-eq xs ys*
⟨*proof*⟩

lemma *lexordp-append-rightI*: *ys* $\neq$ *Nil* $\Longrightarrow$ *lexordp xs* (*xs* @ *ys*)
⟨*proof*⟩

lemma *lexordp-append-left-rightI*: $x < y \Longrightarrow$ *lexordp* (*us* @ *x* # *xs*) (*us* @ *y* # *ys*)
⟨*proof*⟩

lemma *lexordp-eq-refl*: *lexordp-eq xs xs*
⟨*proof*⟩

lemma *lexordp-append-leftI*: *lexordp us vs* $\Longrightarrow$ *lexordp* (*xs* @ *us*) (*xs* @ *vs*)
⟨*proof*⟩

lemma *lexordp-append-leftD*: ⟦ *lexordp* (*xs* @ *us*) (*xs* @ *vs*); $\forall\, a.\ \neg\, a < a$ ⟧ $\Longrightarrow$ *lexordp us vs*
⟨*proof*⟩

lemma *lexordp-irreflexive*:
assumes *irrefl*: $\forall\, x.\ \neg\, x < x$
shows $\neg$ *lexordp xs xs*
⟨*proof*⟩

lemma *lexordp-into-lexordp-eq*:
assumes *lexordp xs ys*
shows *lexordp-eq xs ys*
⟨*proof*⟩

end

declare *ord.lexordp-simps* [*simp*, *code*]
declare *ord.lexordp-eq-simps* [*code*, *simp*]

lemma *lexord-code* [*code*, *code-unfold*]: *lexordp* = *ord.lexordp less*
⟨*proof*⟩

context *order*
begin

lemma *lexordp-antisym*:
assumes *lexordp xs ys lexordp ys xs*
shows *False*
⟨*proof*⟩

lemma *lexordp-irreflexive'*: $\neg$ *lexordp xs xs*
$\langle proof \rangle$

end

context *linorder* **begin**

lemma *lexordp-cases* [*consumes 1*, *case-names Nil Cons Cons-eq*, *cases pred*: *lexordp*]:
  assumes *lexordp xs ys*
  obtains (*Nil*) *y ys'* **where** $xs = []$ $ys = y \# ys'$
  | (*Cons*) *x xs' y ys'* **where** $xs = x \# xs'$ $ys = y \# ys'$ $x < y$
  | (*Cons-eq*) *x xs' ys'* **where** $xs = x \# xs'$ $ys = x \# ys'$ *lexordp xs' ys'*
$\langle proof \rangle$

lemma *lexordp-induct* [*consumes 1*, *case-names Nil Cons Cons-eq*, *induct pred*: *lexordp*]:
  assumes *major*: *lexordp xs ys*
  and *Nil*: $\bigwedge y\ ys.\ P\ []\ (y \# ys)$
  and *Cons*: $\bigwedge x\ xs\ y\ ys.\ x < y \Longrightarrow P\ (x \# xs)\ (y \# ys)$
  and *Cons-eq*: $\bigwedge x\ xs\ ys.\ [\![\ lexordp\ xs\ ys;\ P\ xs\ ys\ ]\!] \Longrightarrow P\ (x \# xs)\ (x \# ys)$
  shows *P xs ys*
$\langle proof \rangle$

lemma *lexordp-iff*:
  $lexordp\ xs\ ys \longleftrightarrow (\exists x\ vs.\ ys = xs\ @\ x \# vs) \vee (\exists us\ a\ b\ vs\ ws.\ a < b \wedge xs = us\ @\ a \# vs \wedge ys = us\ @\ b \# ws)$
  (**is** *?lhs = ?rhs*)
$\langle proof \rangle$

lemma *lexordp-conv-lexord*:
  $lexordp\ xs\ ys \longleftrightarrow (xs,\ ys) \in lexord\ \{(x,\ y).\ x < y\}$
$\langle proof \rangle$

lemma *lexordp-eq-antisym*:
  assumes *lexordp-eq xs ys lexordp-eq ys xs*
  shows $xs = ys$
$\langle proof \rangle$

lemma *lexordp-eq-trans*:
  assumes *lexordp-eq xs ys* **and** *lexordp-eq ys zs*
  shows *lexordp-eq xs zs*
$\langle proof \rangle$

lemma *lexordp-trans*:
  assumes *lexordp xs ys lexordp ys zs*
  shows *lexordp xs zs*
$\langle proof \rangle$

lemma *lexordp-linear*: $lexordp\ xs\ ys \vee xs = ys \vee lexordp\ ys\ xs$
⟨*proof*⟩

lemma *lexordp-conv-lexordp-eq*: $lexordp\ xs\ ys \longleftrightarrow lexordp\text{-}eq\ xs\ ys \wedge \neg\ lexordp\text{-}eq\ ys\ xs$
(**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

lemma *lexordp-eq-conv-lexord*: $lexordp\text{-}eq\ xs\ ys \longleftrightarrow xs = ys \vee lexordp\ xs\ ys$
⟨*proof*⟩

lemma *lexordp-eq-linear*: $lexordp\text{-}eq\ xs\ ys \vee lexordp\text{-}eq\ ys\ xs$
⟨*proof*⟩

lemma *lexordp-linorder*: *class.linorder lexordp-eq lexordp*
⟨*proof*⟩

end

lemma *sorted-insort-is-snoc*: $sorted\ xs \Longrightarrow \forall x \in set\ xs.\ a \geq x \Longrightarrow insort\ a\ xs = xs\ @\ [a]$
⟨*proof*⟩

67.3.3 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*[*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*: $f\ x < f\ y ==> (x,\ y) \in measures\ (f\#fs)$
⟨*proof*⟩

lemma *measures-lesseq*: $f\ x <= f\ y ==> (x,\ y) \in measures\ fs ==> (x,\ y) \in measures\ (f\#fs)$
⟨*proof*⟩

67.3.4 Lifting Relations to Lists: one element

definition $listrel1 :: ('a \times 'a)\ set \Rightarrow ('a\ list \times 'a\ list)\ set$ **where**
$listrel1\ r = \{(xs,ys).$

$\exists\, us\ z\ z'\ vs.\ xs = us\ @\ z\ \#\ vs \wedge (z,z') \in r \wedge ys = us\ @\ z'\ \#\ vs\}$

lemma *listrel1I*:
$[\![\ (x,\ y) \in r;\ \ xs = us\ @\ x\ \#\ vs;\ \ ys = us\ @\ y\ \#\ vs\ ]\!] \Longrightarrow$
$(xs,\ ys) \in listrel1\ r$
⟨*proof*⟩

lemma *listrel1E*:
$[\![\ (xs,\ ys) \in listrel1\ r;$
$!!x\ y\ us\ vs.\ [\![\ (x,\ y) \in r;\ \ xs = us\ @\ x\ \#\ vs;\ \ ys = us\ @\ y\ \#\ vs\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *not-Nil-listrel1* [*iff*]: $([],\ xs) \notin listrel1\ r$
⟨*proof*⟩

lemma *not-listrel1-Nil* [*iff*]: $(xs,\ []) \notin listrel1\ r$
⟨*proof*⟩

lemma *Cons-listrel1-Cons* [*iff*]:
$(x\ \#\ xs,\ y\ \#\ ys) \in listrel1\ r \longleftrightarrow$
$(x,y) \in r \wedge xs = ys \vee x = y \wedge (xs,\ ys) \in listrel1\ r$
⟨*proof*⟩

lemma *listrel1I1*: $(x,y) \in r \Longrightarrow (x\ \#\ xs,\ y\ \#\ xs) \in listrel1\ r$
⟨*proof*⟩

lemma *listrel1I2*: $(xs,\ ys) \in listrel1\ r \Longrightarrow (x\ \#\ xs,\ x\ \#\ ys) \in listrel1\ r$
⟨*proof*⟩

lemma *append-listrel1I*:
$(xs,\ ys) \in listrel1\ r \wedge us = vs \vee xs = ys \wedge (us,\ vs) \in listrel1\ r$
$\Longrightarrow (xs\ @\ us,\ ys\ @\ vs) \in listrel1\ r$
⟨*proof*⟩

lemma *Cons-listrel1E1*[*elim*!]:
assumes $(x\ \#\ xs,\ ys) \in listrel1\ r$
and $\bigwedge y.\ ys = y\ \#\ xs \Longrightarrow (x,\ y) \in r \Longrightarrow R$
and $\bigwedge zs.\ ys = x\ \#\ zs \Longrightarrow (xs,\ zs) \in listrel1\ r \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *Cons-listrel1E2*[*elim*!]:
assumes $(xs,\ y\ \#\ ys) \in listrel1\ r$
and $\bigwedge x.\ xs = x\ \#\ ys \Longrightarrow (x,\ y) \in r \Longrightarrow R$
and $\bigwedge zs.\ xs = y\ \#\ zs \Longrightarrow (zs,\ ys) \in listrel1\ r \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *snoc-listrel1-snoc-iff*:
$(xs\ @\ [x],\ ys\ @\ [y]) \in listrel1\ r$
$\longleftrightarrow (xs,\ ys) \in listrel1\ r \wedge x = y \vee xs = ys \wedge (x,y) \in r$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *listrel1-eq-len*: $(xs,ys) \in listrel1\ r \Longrightarrow length\ xs = length\ ys$
⟨*proof*⟩

lemma *listrel1-mono*:
$r \subseteq s \Longrightarrow listrel1\ r \subseteq listrel1\ s$
⟨*proof*⟩

lemma *listrel1-converse*: $listrel1\ (r\hat{}{-1}) = (listrel1\ r)\hat{}{-1}$
⟨*proof*⟩

lemma *in-listrel1-converse*:
$(x,y) : listrel1\ (r\hat{}{-1}) \longleftrightarrow (x,y) : (listrel1\ r)\hat{}{-1}$
⟨*proof*⟩

lemma *listrel1-iff-update*:
$(xs,ys) \in (listrel1\ r)$
$\longleftrightarrow (\exists\, y\ n.\ (xs\ !\ n,\ y) \in r \wedge n < length\ xs \wedge ys = xs[n{:=}y])$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

Accessible part and wellfoundedness:

lemma *Cons-acc-listrel1I* [*intro*!]:
$x \in Wellfounded.acc\ r \Longrightarrow xs \in Wellfounded.acc\ (listrel1\ r) \Longrightarrow (x\ \#\ xs) \in Wellfounded.acc\ (listrel1\ r)$
⟨*proof*⟩

lemma *lists-accD*: $xs \in lists\ (Wellfounded.acc\ r) \Longrightarrow xs \in Wellfounded.acc\ (listrel1\ r)$
⟨*proof*⟩

lemma *lists-accI*: $xs \in Wellfounded.acc\ (listrel1\ r) \Longrightarrow xs \in lists\ (Wellfounded.acc\ r)$
⟨*proof*⟩

lemma *wf-listrel1-iff* [*simp*]: $wf\,(listrel1\ r) = wf\ r$
⟨*proof*⟩

67.3.5 Lifting Relations to Lists: all elements

inductive-set
$listrel :: ('a \times 'b)\ set \Rightarrow ('a\ list \times 'b\ list)\ set$
for $r :: ('a \times 'b)\ set$
where
Nil: $([],[]) \in listrel\ r$

| *Cons*: $[|\ (x,y) \in r;\ (xs,ys) \in listrel\ r\ |] ==> (x\#xs,\ y\#ys) \in listrel\ r$

inductive-cases *listrel-Nil1* [*elim*!]: $([],xs) \in listrel\ r$
inductive-cases *listrel-Nil2* [*elim*!]: $(xs,[]) \in listrel\ r$
inductive-cases *listrel-Cons1* [*elim*!]: $(y\#ys,xs) \in listrel\ r$
inductive-cases *listrel-Cons2* [*elim*!]: $(xs,y\#ys) \in listrel\ r$

lemma *listrel-eq-len*: $(xs,\ ys) \in listrel\ r \Longrightarrow length\ xs = length\ ys$
⟨*proof*⟩

lemma *listrel-iff-zip* [*code-unfold*]: $(xs,ys) : listrel\ r \longleftrightarrow$
$length\ xs = length\ ys\ \&\ (\forall (x,y) \in set(zip\ xs\ ys).\ (x,y) \in r)$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *listrel-iff-nth*: $(xs,ys) : listrel\ r \longleftrightarrow$
$length\ xs = length\ ys\ \&\ (\forall n < length\ xs.\ (xs!n,\ ys!n) \in r)$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *listrel-mono*: $r \subseteq s \Longrightarrow listrel\ r \subseteq listrel\ s$
⟨*proof*⟩

lemma *listrel-subset*: $r \subseteq A \times A \Longrightarrow listrel\ r \subseteq lists\ A \times lists\ A$
⟨*proof*⟩

lemma *listrel-refl-on*: $refl\text{-}on\ A\ r \Longrightarrow refl\text{-}on\ (lists\ A)\ (listrel\ r)$
⟨*proof*⟩

lemma *listrel-sym*: $sym\ r \Longrightarrow sym\ (listrel\ r)$
⟨*proof*⟩

lemma *listrel-trans*: $trans\ r \Longrightarrow trans\ (listrel\ r)$
⟨*proof*⟩

theorem *equiv-listrel*: $equiv\ A\ r \Longrightarrow equiv\ (lists\ A)\ (listrel\ r)$
⟨*proof*⟩

lemma *listrel-rtrancl-refl*[*iff*]: $(xs,xs) : listrel(r\hat{}*)$
⟨*proof*⟩

lemma *listrel-rtrancl-trans*:
$[\![\ (xs,ys) : listrel(r\hat{}*);\ \ (ys,zs) : listrel(r\hat{}*)\ ]\!]$
$\Longrightarrow (xs,zs) : listrel(r\hat{}*)$
⟨*proof*⟩

lemma *listrel-Nil* [*simp*]: $listrel\ r\ ``\ \{[]\} = \{[]\}$
⟨*proof*⟩

lemma *listrel-Cons*:
$listrel\ r\ ``\ \{x\#xs\} = set\text{-}Cons\ (r``\{x\})\ (listrel\ r\ ``\ \{xs\})$
⟨*proof*⟩

Relating *listrel1*, *listrel* and closures:

lemma *listrel1-rtrancl-subset-rtrancl-listrel1*:
$listrel1\ (r\hat{\ }*) \subseteq (listrel1\ r)\hat{\ }*$
⟨*proof*⟩

lemma *rtrancl-listrel1-eq-len*: $(x,y) \in (listrel1\ r)\hat{\ }* \Longrightarrow length\ x = length\ y$
⟨*proof*⟩

lemma *rtrancl-listrel1-ConsI1*:
$(xs,ys) : (listrel1\ r)\hat{\ }* \Longrightarrow (x\#xs,x\#ys) : (listrel1\ r)\hat{\ }*$
⟨*proof*⟩

lemma *rtrancl-listrel1-ConsI2*:
$(x,y) \in r\hat{\ }* \Longrightarrow (xs,\ ys) \in (listrel1\ r)\hat{\ }*$
$\Longrightarrow (x\ \#\ xs,\ y\ \#\ ys) \in (listrel1\ r)\hat{\ }*$
⟨*proof*⟩

lemma *listrel1-subset-listrel*:
$r \subseteq r' \Longrightarrow refl\ r' \Longrightarrow listrel1\ r \subseteq listrel(r')$
⟨*proof*⟩

lemma *listrel-reflcl-if-listrel1*:
$(xs,ys) : listrel1\ r \Longrightarrow (xs,ys) : listrel(r\hat{\ }*)$
⟨*proof*⟩

lemma *listrel-rtrancl-eq-rtrancl-listrel1*: $listrel\ (r\hat{\ }*) = (listrel1\ r)\hat{\ }*$
⟨*proof*⟩

lemma *rtrancl-listrel1-if-listrel*:
$(xs,ys) : listrel\ r \Longrightarrow (xs,ys) : (listrel1\ r)\hat{\ }*$
⟨*proof*⟩

lemma *listrel-subset-rtrancl-listrel1*: $listrel\ r \subseteq (listrel1\ r)\hat{\ }*$
⟨*proof*⟩

67.4 Size function

lemma [*measure-function*]: $is\text{-}measure\ f \Longrightarrow is\text{-}measure\ (size\text{-}list\ f)$
⟨*proof*⟩

lemma [*measure-function*]: $is\text{-}measure\ f \Longrightarrow is\text{-}measure\ (size\text{-}option\ f)$
⟨*proof*⟩

lemma *size-list-estimation*[*termination-simp*]:

$x \in set\ xs \Longrightarrow y < f\ x \Longrightarrow y < size\text{-}list\ f\ xs$
⟨proof⟩

lemma *size-list-estimation'*[*termination-simp*]:
$x \in set\ xs \Longrightarrow y \leq f\ x \Longrightarrow y \leq size\text{-}list\ f\ xs$
⟨proof⟩

lemma *size-list-map*[*simp*]: $size\text{-}list\ f\ (map\ g\ xs) = size\text{-}list\ (f\ o\ g)\ xs$
⟨proof⟩

lemma *size-list-append*[*simp*]: $size\text{-}list\ f\ (xs\ @\ ys) = size\text{-}list\ f\ xs + size\text{-}list\ f\ ys$
⟨proof⟩

lemma *size-list-pointwise*[*termination-simp*]:
$(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x \leq g\ x) \Longrightarrow size\text{-}list\ f\ xs \leq size\text{-}list\ g\ xs$
⟨proof⟩

67.5 Monad operation

definition $bind :: 'a\ list \Rightarrow ('a \Rightarrow 'b\ list) \Rightarrow 'b\ list$ **where**
$bind\ xs\ f = concat\ (map\ f\ xs)$

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

lemma *bind-simps* [*simp*]:
$List.bind\ []\ f = []$
$List.bind\ (x\ \#\ xs)\ f = f\ x\ @\ List.bind\ xs\ f$
⟨proof⟩

lemma *list-bind-cong* [*fundef-cong*]:
assumes $xs = ys\ (\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x)$
shows $List.bind\ xs\ f = List.bind\ ys\ g$
⟨proof⟩

lemma *set-list-bind*: $set\ (List.bind\ xs\ f) = (\bigcup x{\in}set\ xs.\ set\ (f\ x))$
⟨proof⟩

67.6 Transfer

definition $embed\text{-}list :: nat\ list \Rightarrow int\ list$ **where**
$embed\text{-}list\ l = map\ int\ l$

definition $nat\text{-}list :: int\ list \Rightarrow bool$ **where**
$nat\text{-}list\ l = nat\text{-}set\ (set\ l)$

definition $return\text{-}list :: int\ list \Rightarrow nat\ list$ **where**
$return\text{-}list\ l = map\ nat\ l$

lemma *transfer-nat-int-list-return-embed*: $nat\text{-}list\ l \longrightarrow$
$embed\text{-}list\ (return\text{-}list\ l) = l$

⟨*proof*⟩

lemma *transfer-nat-int-list-functions*:
$l\ @\ m = \mathit{return\text{-}list}\ (\mathit{embed\text{-}list}\ l\ @\ \mathit{embed\text{-}list}\ m)$
$[] = \mathit{return\text{-}list}\ []$
⟨*proof*⟩

67.7 Code generation

Optional tail recursive version of *map*. Can avoid stack overflow in some target languages.

fun *map-tailrec-rev* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
$\mathit{map\text{-}tailrec\text{-}rev}\ f\ []\ bs = bs\ |$
$\mathit{map\text{-}tailrec\text{-}rev}\ f\ (a\#as)\ bs = \mathit{map\text{-}tailrec\text{-}rev}\ f\ as\ (f\ a\ \#\ bs)$

lemma *map-tailrec-rev*:
$\mathit{map\text{-}tailrec\text{-}rev}\ f\ as\ bs = \mathit{rev}(\mathit{map}\ f\ as)\ @\ bs$
⟨*proof*⟩

definition *map-tailrec* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
$\mathit{map\text{-}tailrec}\ f\ as = \mathit{rev}\ (\mathit{map\text{-}tailrec\text{-}rev}\ f\ as\ [])$

Code equation:

lemma *map-eq-map-tailrec*: $\mathit{map} = \mathit{map\text{-}tailrec}$
⟨*proof*⟩

67.7.1 Counterparts for set-related operations

definition *member* :: $'a\ list \Rightarrow 'a \Rightarrow bool$ **where**
[*code-abbrev*]: $\mathit{member}\ xs\ x \longleftrightarrow x \in \mathit{set}\ xs$

Use *member* only for generating executable code. Otherwise use $x \in \mathit{set}\ xs$ instead — it is much easier to reason about.

lemma *member-rec* [*code*]:
$\mathit{member}\ (x\ \#\ xs)\ y \longleftrightarrow x = y \lor \mathit{member}\ xs\ y$
$\mathit{member}\ []\ y \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *in-set-member* :
$x \in \mathit{set}\ xs \longleftrightarrow \mathit{member}\ xs\ x$
⟨*proof*⟩

lemmas *list-all-iff* [*code-abbrev*] = *fun-cong*[*OF list.pred-set*]

definition *list-ex* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**
list-ex-iff [*code-abbrev*]: $\mathit{list\text{-}ex}\ P\ xs \longleftrightarrow \mathit{Bex}\ (\mathit{set}\ xs)\ P$

definition *list-ex1* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**

list-ex1-iff [*code-abbrev*]: *list-ex1 P xs* $\longleftrightarrow$ ($\exists!$ *x*. *x* $\in$ *set xs* $\wedge$ *P x*)

Usually you should prefer $\forall x \in set\ xs$, $\exists x \in set\ xs$ and $\exists!x.\ x \in set\ xs \wedge$ - over *list-all*, *list-ex* and *list-ex1* in specifications.

lemma *list-all-simps* [*code*]:
list-all P (*x* # *xs*) $\longleftrightarrow$ *P x* $\wedge$ *list-all P xs*
list-all P [] $\longleftrightarrow$ *True*
⟨*proof*⟩

lemma *list-ex-simps* [*simp*, *code*]:
list-ex P (*x* # *xs*) $\longleftrightarrow$ *P x* $\vee$ *list-ex P xs*
list-ex P [] $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *list-ex1-simps* [*simp*, *code*]:
list-ex1 P [] = *False*
list-ex1 P (*x* # *xs*) = (*if P x then list-all* (λy. $\neg$ *P y* $\vee$ *x* = *y*) *xs else list-ex1 P xs*)
⟨*proof*⟩

lemma *Ball-set-list-all*:
Ball (*set xs*) *P* $\longleftrightarrow$ *list-all P xs*
⟨*proof*⟩

lemma *Bex-set-list-ex*:
Bex (*set xs*) *P* $\longleftrightarrow$ *list-ex P xs*
⟨*proof*⟩

lemma *list-all-append* [*simp*]:
list-all P (*xs* @ *ys*) $\longleftrightarrow$ *list-all P xs* $\wedge$ *list-all P ys*
⟨*proof*⟩

lemma *list-ex-append* [*simp*]:
list-ex P (*xs* @ *ys*) $\longleftrightarrow$ *list-ex P xs* $\vee$ *list-ex P ys*
⟨*proof*⟩

lemma *list-all-rev* [*simp*]:
list-all P (*rev xs*) $\longleftrightarrow$ *list-all P xs*
⟨*proof*⟩

lemma *list-ex-rev* [*simp*]:
list-ex P (*rev xs*) $\longleftrightarrow$ *list-ex P xs*
⟨*proof*⟩

lemma *list-all-length*:
list-all P xs $\longleftrightarrow$ ($\forall n$ < *length xs*. *P* (*xs* ! *n*))
⟨*proof*⟩

lemma *list-ex-length*:

$list\text{-}ex\ P\ xs \longleftrightarrow (\exists\, n < length\ xs.\ P\ (xs\ !\ n))$
$\langle proof \rangle$

lemmas $list\text{-}all\text{-}cong\ [fundef\text{-}cong] = list.pred\text{-}cong$

lemma $list\text{-}ex\text{-}cong\ [fundef\text{-}cong]$:
$xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x) \Longrightarrow list\text{-}ex\ f\ xs = list\text{-}ex\ g\ ys$
$\langle proof \rangle$

definition $can\text{-}select :: ('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$ **where**
$[code\text{-}abbrev]$: $can\text{-}select\ P\ A = (\exists !x{\in}A.\ P\ x)$

lemma $can\text{-}select\text{-}set\text{-}list\text{-}ex1\ [code]$:
$can\text{-}select\ P\ (set\ A) = list\text{-}ex1\ P\ A$
$\langle proof \rangle$

Executable checks for relations on sets

definition $listrel1p :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
$listrel1p\ r\ xs\ ys = ((xs,\ ys) \in listrel1\ \{(x,\ y).\ r\ x\ y\})$

lemma $[code\text{-}unfold]$:
$(xs,\ ys) \in listrel1\ r = listrel1p\ (\lambda x\ y.\ (x,\ y) \in r)\ xs\ ys$
$\langle proof \rangle$

lemma $[code]$:
$listrel1p\ r\ []\ xs = False$
$listrel1p\ r\ xs\ [] = False$
$listrel1p\ r\ (x\ \#\ xs)\ (y\ \#\ ys) \longleftrightarrow$
$r\ x\ y \wedge xs = ys \vee x = y \wedge listrel1p\ r\ xs\ ys$
$\langle proof \rangle$

definition
$lexordp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
$lexordp\ r\ xs\ ys = ((xs,\ ys) \in lexord\ \{(x,\ y).\ r\ x\ y\})$

lemma $[code\text{-}unfold]$:
$(xs,\ ys) \in lexord\ r = lexordp\ (\lambda x\ y.\ (x,\ y) \in r)\ xs\ ys$
$\langle proof \rangle$

lemma $[code]$:
$lexordp\ r\ xs\ [] = False$
$lexordp\ r\ []\ (y\#ys) = True$
$lexordp\ r\ (x\ \#\ xs)\ (y\ \#\ ys) = (r\ x\ y\ |\ (x = y\ \&\ lexordp\ r\ xs\ ys))$
$\langle proof \rangle$

Bounded quantification and summation over nats.

lemma $atMost\text{-}upto\ [code\text{-}unfold]$:
$\{..n\} = set\ [0..{<}Suc\ n]$
$\langle proof \rangle$

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*]:
 $(\forall\, m\le n::nat.\ P\ m) \longleftrightarrow (\forall\, m \in \{0..n\}.\ P\ m)$
 ⟨*proof*⟩

lemma *ex-nat-less* [*code-unfold*]:
 $(\exists\, m\le n::nat.\ P\ m) \longleftrightarrow (\exists\, m \in \{0..n\}.\ P\ m)$
 ⟨*proof*⟩

Bounded *LEAST* operator:

definition $Bleast\ S\ P = (LEAST\ x.\ x \in S \wedge P\ x)$

definition $abort\text{-}Bleast\ S\ P = (LEAST\ x.\ x \in S \wedge P\ x)$

declare [[*code abort*: *abort-Bleast*]]

lemma *Bleast-code* [*code*]:
 $Bleast\ (set\ xs)\ P = (case\ filter\ P\ (sort\ xs)\ of$
  $x\#xs \Rightarrow x\ |$
  $[] \Rightarrow abort\text{-}Bleast\ (set\ xs)\ P)$
⟨*proof*⟩

declare *Bleast-def* [*symmetric*, *code-unfold*]

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*]

67.7.2 Optimizing by rewriting

definition *null* :: $'a\ list \Rightarrow bool$ **where**
[*code-abbrev*]: $null\ xs \longleftrightarrow xs = []$

Efficient emptyness check is implemented by *null*.

lemma *null-rec* [*code*]:
$null\ (x \# xs) \longleftrightarrow False$
$null\ [] \longleftrightarrow True$
⟨*proof*⟩

lemma *eq-Nil-null*:
$xs = [] \longleftrightarrow null\ xs$
⟨*proof*⟩

lemma *equal-Nil-null* [*code-unfold*]:
$HOL.equal\ xs\ [] \longleftrightarrow null\ xs$
$HOL.equal\ [] = null$
⟨*proof*⟩

definition *maps* :: $('a \Rightarrow 'b\ list) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
[*code-abbrev*]: $maps\ f\ xs = concat\ (map\ f\ xs)$

definition *map-filter* :: $('a \Rightarrow 'b\ option) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
[*code-post*]: $map\text{-}filter\ f\ xs = map\ (the \circ f)\ (filter\ (\lambda x.\ f\ x \neq None)\ xs)$

Operations *maps* and *map-filter* avoid intermediate lists on execution – do not use for proving.

lemma *maps-simps* [*code*]:
$maps\ f\ (x \# xs) = f\ x\ @\ maps\ f\ xs$
$maps\ f\ [] = []$
⟨*proof*⟩

lemma *map-filter-simps* [*code*]:

map-filter f $(x \# xs) = ($*case f x of None* $\Rightarrow$ *map-filter f xs* $\mid$ *Some y* $\Rightarrow y \#$ *map-filter f xs*$)$
map-filter f $[] = []$
⟨*proof*⟩

lemma *concat-map-maps*:
concat (*map f xs*) = *maps f xs*
⟨*proof*⟩

lemma *map-filter-map-filter* [*code-unfold*]:
map f (*filter P xs*) = *map-filter* (λx. *if P x then Some* (*f x*) *else None*) *xs*
⟨*proof*⟩

Optimized code for $\forall\, i \in \{a..b::int\}$ and $\forall\, n:\{a..<b::nat\}$ and similiarly for $\exists$.

definition *all-interval-nat* :: (*nat* $\Rightarrow$ *bool*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
all-interval-nat P i j $\longleftrightarrow$ $(\forall\, n \in \{i..<j\}.\ P\ n)$

lemma [*code*]:
all-interval-nat P i j $\longleftrightarrow$ $i \geq j \vee P\ i \wedge$ *all-interval-nat P* (*Suc i*) *j*
⟨*proof*⟩

lemma *list-all-iff-all-interval-nat* [*code-unfold*]:
list-all P $[i..<j]$ $\longleftrightarrow$ *all-interval-nat P i j*
⟨*proof*⟩

lemma *list-ex-iff-not-all-inverval-nat* [*code-unfold*]:
list-ex P $[i..<j]$ $\longleftrightarrow$ $\neg$ (*all-interval-nat* (*Not* $\circ$ *P*) *i j*)
⟨*proof*⟩

definition *all-interval-int* :: (*int* $\Rightarrow$ *bool*) $\Rightarrow$ *int* $\Rightarrow$ *int* $\Rightarrow$ *bool* **where**
all-interval-int P i j $\longleftrightarrow$ $(\forall\, k \in \{i..j\}.\ P\ k)$

lemma [*code*]:
all-interval-int P i j $\longleftrightarrow$ $i > j \vee P\ i \wedge$ *all-interval-int P* $(i + 1)$ *j*
⟨*proof*⟩

lemma *list-all-iff-all-interval-int* [*code-unfold*]:
list-all P $[i..j]$ $\longleftrightarrow$ *all-interval-int P i j*
⟨*proof*⟩

lemma *list-ex-iff-not-all-inverval-int* [*code-unfold*]:
list-ex P $[i..j]$ $\longleftrightarrow$ $\neg$ (*all-interval-int* (*Not* $\circ$ *P*) *i j*)
⟨*proof*⟩

optimized code (tail-recursive) for *length*

definition *gen-length* :: *nat* $\Rightarrow$ *'a list* $\Rightarrow$ *nat*
where *gen-length n xs* = *n* + *length xs*

lemma *gen-length-code* [*code*]:
  gen-length n [] = *n*
  gen-length n (*x* # *xs*) = *gen-length* (*Suc n*) *xs*
⟨*proof*⟩

declare *list.size*(*3*−*4*)[*code del*]

lemma *length-code* [*code*]: *length* = *gen-length 0*
⟨*proof*⟩

hide-const (**open**) *member null maps map-filter all-interval-nat all-interval-int gen-length*

67.7.3 Pretty lists

⟨*ML*⟩

code-printing
  type-constructor *list* ⇀
    (*SML*) - *list*
    and (*OCaml*) - *list*
    and (*Haskell*) ![(-)]
    and (*Scala*) *List*[(-)]
| **constant** *Nil* ⇀
    (*SML*) []
    and (*OCaml*) []
    and (*Haskell*) []
    and (*Scala*) !*Nil*
| **class-instance** *list* :: *equal* ⇀
    (*Haskell*) −
| **constant** *HOL.equal* :: *'a list* ⇒ *'a list* ⇒ *bool* ⇀
    (*Haskell*) **infix** *4* ==

⟨*ML*⟩

code-reserved *SML*
  list

code-reserved *OCaml*
  list

67.7.4 Use convenient predefined operations

code-printing
  constant *op* @ ⇀
    (*SML*) **infixr** *7* @
    and (*OCaml*) **infixr** *6* @
    and (*Haskell*) **infixr** *5* ++
    and (*Scala*) **infixl** *7* ++
| **constant** *map* ⇀

  $(Haskell)$ *map*
| **constant** *filter* $\rightharpoonup$
  $(Haskell)$ *filter*
| **constant** *concat* $\rightharpoonup$
  $(Haskell)$ *concat*
| **constant** *List.maps* $\rightharpoonup$
  $(Haskell)$ *concatMap*
| **constant** *rev* $\rightharpoonup$
  $(Haskell)$ *reverse*
| **constant** *zip* $\rightharpoonup$
  $(Haskell)$ *zip*
| **constant** *List.null* $\rightharpoonup$
  $(Haskell)$ *null*
| **constant** *takeWhile* $\rightharpoonup$
  $(Haskell)$ *takeWhile*
| **constant** *dropWhile* $\rightharpoonup$
  $(Haskell)$ *dropWhile*
| **constant** *list-all* $\rightharpoonup$
  $(Haskell)$ *all*
| **constant** *list-ex* $\rightharpoonup$
  $(Haskell)$ *any*

67.7.5 Implementation of sets by lists

lemma *is-empty-set* [*code*]:
  Set.is-empty (*set xs*) $\longleftrightarrow$ *List.null xs*
  ⟨*proof*⟩

lemma *empty-set* [*code*]:
  $\{\}$ = *set* []
  ⟨*proof*⟩

lemma *UNIV-coset* [*code*]:
  UNIV = *List.coset* []
  ⟨*proof*⟩

lemma *compl-set* [*code*]:
  $-$ *set xs* = *List.coset xs*
  ⟨*proof*⟩

lemma *compl-coset* [*code*]:
  $-$ *List.coset xs* = *set xs*
  ⟨*proof*⟩

lemma [*code*]:
  $x \in$ *set xs* $\longleftrightarrow$ *List.member xs x*
  $x \in$ *List.coset xs* $\longleftrightarrow$ $\neg$ *List.member xs x*
  ⟨*proof*⟩

lemma *insert-code* [*code*]:
$insert\ x\ (set\ xs) = set\ (List.insert\ x\ xs)$
$insert\ x\ (List.coset\ xs) = List.coset\ (removeAll\ x\ xs)$
⟨*proof*⟩

lemma *remove-code* [*code*]:
$Set.remove\ x\ (set\ xs) = set\ (removeAll\ x\ xs)$
$Set.remove\ x\ (List.coset\ xs) = List.coset\ (List.insert\ x\ xs)$
⟨*proof*⟩

lemma *filter-set* [*code*]:
$Set.filter\ P\ (set\ xs) = set\ (filter\ P\ xs)$
⟨*proof*⟩

lemma *image-set* [*code*]:
$image\ f\ (set\ xs) = set\ (map\ f\ xs)$
⟨*proof*⟩

lemma *subset-code* [*code*]:
$set\ xs \leq B \longleftrightarrow (\forall\, x{\in}set\ xs.\ x \in B)$
$A \leq List.coset\ ys \longleftrightarrow (\forall\, y{\in}set\ ys.\ y \notin A)$
$List.coset\ []\ \leq set\ []\ \longleftrightarrow False$
⟨*proof*⟩

A frequent case – avoid intermediate sets

lemma [*code-unfold*]:
$set\ xs \subseteq set\ ys \longleftrightarrow list\text{-}all\ (\lambda x.\ x \in set\ ys)\ xs$
⟨*proof*⟩

lemma *Ball-set* [*code*]:
$Ball\ (set\ xs)\ P \longleftrightarrow list\text{-}all\ P\ xs$
⟨*proof*⟩

lemma *Bex-set* [*code*]:
$Bex\ (set\ xs)\ P \longleftrightarrow list\text{-}ex\ P\ xs$
⟨*proof*⟩

lemma *card-set* [*code*]:
$card\ (set\ xs) = length\ (remdups\ xs)$
⟨*proof*⟩

lemma *the-elem-set* [*code*]:
$the\text{-}elem\ (set\ [x]) = x$
⟨*proof*⟩

lemma *Pow-set* [*code*]:
$Pow\ (set\ []) = \{\{\}\}$
$Pow\ (set\ (x\ \#\ xs)) = (let\ A = Pow\ (set\ xs)\ in\ A \cup insert\ x\ `\ A)$
⟨*proof*⟩

definition *map-project* :: ($'a \Rightarrow 'b$ *option*) $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'b$ *set* **where**
map-project f A = {*b*. ∃ *a* ∈ *A*. *f a* = *Some b*}

lemma [*code*]:
map-project f (*set xs*) = *set* (*List.map-filter f xs*)
⟨*proof*⟩

hide-const (**open**) *map-project*

Operations on relations

lemma *product-code* [*code*]:
Product-Type.product (*set xs*) (*set ys*) = *set* [(*x*, *y*). *x* ← *xs*, *y* ← *ys*]
⟨*proof*⟩

lemma *Id-on-set* [*code*]:
Id-on (*set xs*) = *set* [(*x*, *x*). *x* ← *xs*]
⟨*proof*⟩

lemma [*code*]:
R `` *S* = *List.map-project* (%(*x*, *y*). *if x* : *S then Some y else None*) *R*
⟨*proof*⟩

lemma *trancl-set-ntrancl* [*code*]:
trancl (*set xs*) = *ntrancl* (*card* (*set xs*) − *1*) (*set xs*)
⟨*proof*⟩

lemma *set-relcomp* [*code*]:
set xys O set yzs = *set* ([(*fst xy*, *snd yz*). *xy* ← *xys*, *yz* ← *yzs*, *snd xy* = *fst yz*])
⟨*proof*⟩

lemma *wf-set* [*code*]:
wf (*set xs*) = *acyclic* (*set xs*)
⟨*proof*⟩

67.8 Setup for Lifting/Transfer

67.8.1 Transfer rules for the Transfer package

context includes *lifting-syntax*
begin

lemma *tl-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A*) *tl tl*
⟨*proof*⟩

lemma *butlast-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A*) *butlast butlast*
⟨*proof*⟩

lemma *map-rec*: *map f xs = rec-list Nil (%x - y. Cons (f x) y) xs*
⟨*proof*⟩

lemma *append-transfer* [*transfer-rule*]:
(*list-all2 A ===> list-all2 A ===> list-all2 A*) *append append*
⟨*proof*⟩

lemma *rev-transfer* [*transfer-rule*]:
(*list-all2 A ===> list-all2 A*) *rev rev*
⟨*proof*⟩

lemma *filter-transfer* [*transfer-rule*]:
((*A ===> op =*) *===> list-all2 A ===> list-all2 A*) *filter filter*
⟨*proof*⟩

lemma *fold-transfer* [*transfer-rule*]:
((*A ===> B ===> B*) *===> list-all2 A ===> B ===> B*) *fold fold*
⟨*proof*⟩

lemma *foldr-transfer* [*transfer-rule*]:
((*A ===> B ===> B*) *===> list-all2 A ===> B ===> B*) *foldr foldr*
⟨*proof*⟩

lemma *foldl-transfer* [*transfer-rule*]:
((*B ===> A ===> B*) *===> B ===> list-all2 A ===> B*) *foldl foldl*
⟨*proof*⟩

lemma *concat-transfer* [*transfer-rule*]:
(*list-all2* (*list-all2 A*) *===> list-all2 A*) *concat concat*
⟨*proof*⟩

lemma *drop-transfer* [*transfer-rule*]:
(*op = ===> list-all2 A ===> list-all2 A*) *drop drop*
⟨*proof*⟩

lemma *take-transfer* [*transfer-rule*]:
(*op = ===> list-all2 A ===> list-all2 A*) *take take*
⟨*proof*⟩

lemma *list-update-transfer* [*transfer-rule*]:
(*list-all2 A ===> op = ===> A ===> list-all2 A*) *list-update list-update*
⟨*proof*⟩

lemma *takeWhile-transfer* [*transfer-rule*]:
((*A ===> op =*) *===> list-all2 A ===> list-all2 A*) *takeWhile takeWhile*
⟨*proof*⟩

lemma *dropWhile-transfer* [*transfer-rule*]:
((*A ===> op =*) *===> list-all2 A ===> list-all2 A*) *dropWhile dropWhile*

⟨*proof*⟩

lemma *zip-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 B* ===> *list-all2* (*rel-prod A B*)) *zip zip*
⟨*proof*⟩

lemma *product-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 B* ===> *list-all2* (*rel-prod A B*)) *List.product List.product*
⟨*proof*⟩

lemma *product-lists-transfer* [*transfer-rule*]:
(*list-all2* (*list-all2 A*) ===> *list-all2* (*list-all2 A*)) *product-lists product-lists*
⟨*proof*⟩

lemma *insert-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *list-all2 A* ===> *list-all2 A*) *List.insert List.insert*
⟨*proof*⟩

lemma *find-transfer* [*transfer-rule*]:
((*A* ===> *op* =) ===> *list-all2 A* ===> *rel-option A*) *List.find List.find*
⟨*proof*⟩

lemma *those-transfer* [*transfer-rule*]:
(*list-all2* (*rel-option P*) ===> *rel-option* (*list-all2 P*)) *those those*
⟨*proof*⟩

lemma *remove1-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *list-all2 A* ===> *list-all2 A*) *remove1 remove1*
⟨*proof*⟩

lemma *removeAll-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *list-all2 A* ===> *list-all2 A*) *removeAll removeAll*
⟨*proof*⟩

lemma *distinct-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *op* =) *distinct distinct*
⟨*proof*⟩

lemma *remdups-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A*) *remdups remdups*
⟨*proof*⟩

lemma *remdups-adj-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*

shows (*list-all2 A* ===> *list-all2 A*) *remdups-adj remdups-adj*
⟨*proof*⟩

lemma *replicate-transfer* [*transfer-rule*]:
(*op* = ===> *A* ===> *list-all2 A*) *replicate replicate*
⟨*proof*⟩

lemma *length-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *op* =) *length length*
⟨*proof*⟩

lemma *rotate1-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A*) *rotate1 rotate1*
⟨*proof*⟩

lemma *rotate-transfer* [*transfer-rule*]:
(*op* = ===> *list-all2 A* ===> *list-all2 A*) *rotate rotate*
⟨*proof*⟩

lemma *nths-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *rel-set* (*op* =) ===> *list-all2 A*) *nths nths*
⟨*proof*⟩

lemma *subseqs-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2* (*list-all2 A*)) *subseqs subseqs*
⟨*proof*⟩

lemma *partition-transfer* [*transfer-rule*]:
((*A* ===> *op* =) ===> *list-all2 A* ===> *rel-prod* (*list-all2 A*) (*list-all2 A*))
partition partition
⟨*proof*⟩

lemma *lists-transfer* [*transfer-rule*]:
(*rel-set A* ===> *rel-set* (*list-all2 A*)) *lists lists*
⟨*proof*⟩

lemma *set-Cons-transfer* [*transfer-rule*]:
(*rel-set A* ===> *rel-set* (*list-all2 A*) ===> *rel-set* (*list-all2 A*))
set-Cons set-Cons
⟨*proof*⟩

lemma *listset-transfer* [*transfer-rule*]:
(*list-all2* (*rel-set A*) ===> *rel-set* (*list-all2 A*)) *listset listset*
⟨*proof*⟩

lemma *null-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *op* =) *List.null List.null*
⟨*proof*⟩

lemma *list-all-transfer* [*transfer-rule*]:
((*A* ===> *op* =) ===> *list-all2 A* ===> *op* =) *list-all list-all*
⟨*proof*⟩

lemma *list-ex-transfer* [*transfer-rule*]:
((*A* ===> *op* =) ===> *list-all2 A* ===> *op* =) *list-ex list-ex*
⟨*proof*⟩

lemma *splice-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A* ===> *list-all2 A*) *splice splice*
⟨*proof*⟩

lemma *shuffle-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A* ===> *rel-set* (*list-all2 A*)) *shuffle shuffle*
⟨*proof*⟩

lemma *rtrancl-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A bi-total A*
shows (*rel-set* (*rel-prod A A*) ===> *rel-set* (*rel-prod A A*)) *rtrancl rtrancl*
⟨*proof*⟩

lemma *monotone-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *A* ===> *op* =) ===> (*B* ===> *B* ===> *op* =) ===> (*A* ===> *B*) ===> *op* =) *monotone monotone*
⟨*proof*⟩

lemma *fun-ord-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total C*
shows ((*A* ===> *B* ===> *op* =) ===> (*C* ===> *A*) ===> (*C* ===> *B*) ===> *op* =) *fun-ord fun-ord*
⟨*proof*⟩

lemma *fun-lub-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique A*
shows ((*rel-set A* ===> *B*) ===> *rel-set* (*C* ===> *A*) ===> *C* ===> *B*) *fun-lub fun-lub*
⟨*proof*⟩

end

end

68 Sum and product over lists

theory *Groups-List*
imports *List*
begin

locale *monoid-list* = *monoid*
begin

definition F :: $'a$ *list* $\Rightarrow$ $'a$
where
eq-foldr [*code*]: $F\ xs = foldr\ f\ xs\ \mathbf{1}$

lemma *Nil* [*simp*]:
$F\ [] = \mathbf{1}$
⟨*proof*⟩

lemma *Cons* [*simp*]:
$F\ (x\ \#\ xs) = x * F\ xs$
⟨*proof*⟩

lemma *append* [*simp*]:
$F\ (xs\ @\ ys) = F\ xs * F\ ys$
⟨*proof*⟩

end

locale *comm-monoid-list* = *comm-monoid* + *monoid-list*
begin

lemma *rev* [*simp*]:
$F\ (rev\ xs) = F\ xs$
⟨*proof*⟩

end

locale *comm-monoid-list-set* = *list*: *comm-monoid-list* + *set*: *comm-monoid-set*
begin

lemma *distinct-set-conv-list*:
$distinct\ xs \Longrightarrow set.F\ g\ (set\ xs) = list.F\ (map\ g\ xs)$
⟨*proof*⟩

lemma *set-conv-list* [*code*]:
$set.F\ g\ (set\ xs) = list.F\ (map\ g\ (remdups\ xs))$
⟨*proof*⟩

end

68.1 List summation

context *monoid-add*
begin

sublocale *sum-list*: *monoid-list plus 0*

defines
  sum-list = *sum-list.F* ⟨*proof*⟩

end

context *comm-monoid-add*
begin

sublocale *sum-list*: *comm-monoid-list plus 0*
rewrites
  monoid-list.F plus 0 = *sum-list*
⟨*proof*⟩

sublocale *sum*: *comm-monoid-list-set plus 0*
rewrites
  monoid-list.F plus 0 = *sum-list*
  and *comm-monoid-set.F plus 0* = *sum*
⟨*proof*⟩

end

Some syntactic sugar for summing a function over a list:

syntax (*ASCII*)
  -sum-list :: *pttrn* => *'a list* => *'b* => *'b*   ((*3SUM* -<--. -) [*0*, *51*, *10*] *10*)
syntax
  -sum-list :: *pttrn* => *'a list* => *'b* => *'b*   ((*3*$\sum$ -←-. -) [*0*, *51*, *10*] *10*)
translations — Beware of argument permutation!
  $\sum$ *x*←*xs*. *b* == *CONST sum-list* (*CONST map* (λ*x*. *b*) *xs*)

TODO duplicates

lemmas *sum-list-simps* = *sum-list.Nil sum-list.Cons*
lemmas *sum-list-append* = *sum-list.append*
lemmas *sum-list-rev* = *sum-list.rev*

lemma (**in** *monoid-add*) *fold-plus-sum-list-rev*:
  fold plus xs = *plus* (*sum-list* (*rev xs*))
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *sum-list-map-remove1*:
  x ∈ *set xs* ⟹ *sum-list* (*map f xs*) = *f x* + *sum-list* (*map f* (*remove1 x xs*))
  ⟨*proof*⟩

lemma (**in** *monoid-add*) *size-list-conv-sum-list*:
  size-list f xs = *sum-list* (*map f xs*) + *size xs*
  ⟨*proof*⟩

lemma (**in** *monoid-add*) *length-concat*:
  length (*concat xss*) = *sum-list* (*map length xss*)
  ⟨*proof*⟩

lemma (**in** *monoid-add*) *length-product-lists*:
  $length\ (product\text{-}lists\ xss) = foldr\ op * (map\ length\ xss)\ 1$
⟨*proof*⟩

lemma (**in** *monoid-add*) *sum-list-map-filter*:
  assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow \neg\ P\ x \Longrightarrow f\ x = 0$
  shows $sum\text{-}list\ (map\ f\ (filter\ P\ xs)) = sum\text{-}list\ (map\ f\ xs)$
  ⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *distinct-sum-list-conv-Sum*:
  $distinct\ xs \Longrightarrow sum\text{-}list\ xs = Sum\ (set\ xs)$
  ⟨*proof*⟩

lemma *sum-list-upt*[*simp*]:
  $m \leq n \Longrightarrow sum\text{-}list\ [m..<n] = \sum\ \{m..<n\}$
⟨*proof*⟩

context *ordered-comm-monoid-add*
begin

lemma *sum-list-nonneg*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow 0 \leq x) \Longrightarrow 0 \leq sum\text{-}list\ xs$
⟨*proof*⟩

lemma *sum-list-nonpos*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow x \leq 0) \Longrightarrow sum\text{-}list\ xs \leq 0$
⟨*proof*⟩

lemma *sum-list-nonneg-eq-0-iff*:
  $(\bigwedge x.\ x \in set\ xs \Longrightarrow 0 \leq x) \Longrightarrow sum\text{-}list\ xs = 0 \longleftrightarrow (\forall x \in set\ xs.\ x = 0)$
⟨*proof*⟩

end

context *canonically-ordered-monoid-add*
begin

lemma *sum-list-eq-0-iff* [*simp*]:
  $sum\text{-}list\ ns = 0 \longleftrightarrow (\forall\ n \in set\ ns.\ n = 0)$
⟨*proof*⟩

lemma *member-le-sum-list*:
  $x \in set\ xs \Longrightarrow x \leq sum\text{-}list\ xs$
⟨*proof*⟩

lemma *elem-le-sum-list*:
  $k < size\ ns \Longrightarrow ns\ !\ k \leq sum\text{-}list\ (ns)$
⟨*proof*⟩

end

lemma (**in** *ordered-cancel-comm-monoid-diff*) *sum-list-update*:
$k < size\ xs \Longrightarrow sum\text{-}list\ (xs[k := x]) = sum\text{-}list\ xs + x - xs\ !\ k$
⟨*proof*⟩

lemma (**in** *monoid-add*) *sum-list-triv*:
$(\sum x \leftarrow xs.\ r) = of\text{-}nat\ (length\ xs) * r$
⟨*proof*⟩

lemma (**in** *monoid-add*) *sum-list-0* [*simp*]:
$(\sum x \leftarrow xs.\ 0) = 0$
⟨*proof*⟩

For non-Abelian groups *xs* needs to be reversed on one side:

lemma (**in** *ab-group-add*) *uminus-sum-list-map*:
$-\ sum\text{-}list\ (map\ f\ xs) = sum\text{-}list\ (map\ (uminus \circ f)\ xs)$
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *sum-list-addf*:
$(\sum x \leftarrow xs.\ f\ x + g\ x) = sum\text{-}list\ (map\ f\ xs) + sum\text{-}list\ (map\ g\ xs)$
⟨*proof*⟩

lemma (**in** *ab-group-add*) *sum-list-subtractf*:
$(\sum x \leftarrow xs.\ f\ x - g\ x) = sum\text{-}list\ (map\ f\ xs) - sum\text{-}list\ (map\ g\ xs)$
⟨*proof*⟩

lemma (**in** *semiring-0*) *sum-list-const-mult*:
$(\sum x \leftarrow xs.\ c * f\ x) = c * (\sum x \leftarrow xs.\ f\ x)$
⟨*proof*⟩

lemma (**in** *semiring-0*) *sum-list-mult-const*:
$(\sum x \leftarrow xs.\ f\ x * c) = (\sum x \leftarrow xs.\ f\ x) * c$
⟨*proof*⟩

lemma (**in** *ordered-ab-group-add-abs*) *sum-list-abs*:
$|sum\text{-}list\ xs| \leq sum\text{-}list\ (map\ abs\ xs)$
⟨*proof*⟩

lemma *sum-list-mono*:
fixes $f\ g :: 'a \Rightarrow 'b::\{monoid\text{-}add,\ ordered\text{-}ab\text{-}semigroup\text{-}add\}$
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*⟩

lemma (**in** *monoid-add*) *sum-list-distinct-conv-sum-set*:
$distinct\ xs \Longrightarrow sum\text{-}list\ (map\ f\ xs) = sum\ f\ (set\ xs)$
⟨*proof*⟩

lemma (**in** *monoid-add*) *interv-sum-list-conv-sum-set-nat*:
$sum\text{-}list\ (map\ f\ [m..<n]) = sum\ f\ (set\ [m..<n])$

⟨*proof*⟩

lemma (**in** *monoid-add*) *interv-sum-list-conv-sum-set-int*:
sum-list (*map f* [*k..l*]) = *sum f* (*set* [*k..l*])
⟨*proof*⟩

General equivalence between *sum-list* and *sum*

lemma (**in** *monoid-add*) *sum-list-sum-nth*:
sum-list xs = ($\sum$ *i* = *0* ..< *length xs*. *xs* ! *i*)
⟨*proof*⟩

lemma *sum-list-map-eq-sum-count*:
sum-list (*map f xs*) = *sum* (λx. *count-list xs x* $*$ *f x*) (*set xs*)
⟨*proof*⟩

lemma *sum-list-map-eq-sum-count2*:
assumes *set xs* $\subseteq$ *X finite X*
shows *sum-list* (*map f xs*) = *sum* (λx. *count-list xs x* $*$ *f x*) *X*
⟨*proof*⟩

lemma *sum-list-nonneg*:
($\bigwedge x$. *x* $\in$ *set xs* $\Longrightarrow$ (*x* :: *'a* :: *ordered-comm-monoid-add*) $\geq$ *0*) $\Longrightarrow$ *sum-list xs* $\geq$ *0*
⟨*proof*⟩

lemma (**in** *monoid-add*) *sum-list-map-filter'*:
sum-list (*map f* (*filter P xs*)) = *sum-list* (*map* (λx. *if P x then f x else 0*) *xs*)
⟨*proof*⟩

lemma *sum-list-cong* [*fundef-cong*]:
assumes *xs* = *ys*
assumes $\bigwedge x$. *x* $\in$ *set xs* $\Longrightarrow$ *f x* = *g x*
shows *sum-list* (*map f xs*) = *sum-list* (*map g ys*)
⟨*proof*⟩

Summation of a strictly ascending sequence with length n can be upper-bounded by summation over $\{0..<n\}$.

lemma *sorted-wrt-less-sum-mono-lowerbound*:
fixes *f* :: *nat* $\Rightarrow$ (*'b*::*ordered-comm-monoid-add*)
assumes *mono*: $\bigwedge x\ y$. $x \leq y \Longrightarrow f\ x \leq f\ y$
shows *sorted-wrt* (*op* <) *ns* $\Longrightarrow$
($\sum i \in \{0..<length\ ns\}$. *f i*) $\leq$ ($\sum i \leftarrow ns$. *f i*)
⟨*proof*⟩

68.2 Further facts about *List.n-lists*

lemma *length-n-lists*: *length* (*List.n-lists n xs*) = *length xs* ^ *n*
⟨*proof*⟩

lemma *distinct-n-lists*:
  assumes *distinct xs*
  shows *distinct* (*List.n-lists n xs*)
⟨*proof*⟩

68.3 Tools setup

lemmas *sum-code* = *sum.set-conv-list*

lemma *sum-set-upto-conv-sum-list-int* [*code-unfold*]:
  sum f (*set* [*i..j*::*int*]) = *sum-list* (*map f* [*i..j*])
  ⟨*proof*⟩

lemma *sum-set-upt-conv-sum-list-nat* [*code-unfold*]:
  sum f (*set* [*m..*<*n*]) = *sum-list* (*map f* [*m..*<*n*])
  ⟨*proof*⟩

lemma *sum-list-transfer*[*transfer-rule*]:
  includes *lifting-syntax*
  assumes [*transfer-rule*]: *A 0 0*
  assumes [*transfer-rule*]: (*A* ===> *A* ===> *A*) *op* + *op* +
  shows (*list-all2 A* ===> *A*) *sum-list sum-list*
  ⟨*proof*⟩

68.4 List product

context *monoid-mult*
begin

sublocale *prod-list*: *monoid-list times 1*
defines
  prod-list = *prod-list.F* ⟨*proof*⟩

end

context *comm-monoid-mult*
begin

sublocale *prod-list*: *comm-monoid-list times 1*
rewrites
  monoid-list.F times 1 = *prod-list*
⟨*proof*⟩

sublocale *prod*: *comm-monoid-list-set times 1*
rewrites
  monoid-list.F times 1 = *prod-list*
  and *comm-monoid-set.F times 1* = *prod*
⟨*proof*⟩

end

lemma *prod-list-cong* [*fundef-cong*]:
  assumes $xs = ys$
  assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x$
  shows $prod\text{-}list\ (map\ f\ xs) = prod\text{-}list\ (map\ g\ ys)$
⟨*proof*⟩

lemma *prod-list-zero-iff*:
  $prod\text{-}list\ xs = 0 \longleftrightarrow (0 :: 'a :: \{semiring\text{-}no\text{-}zero\text{-}divisors,\ semiring\text{-}1\}) \in set\ xs$
  ⟨*proof*⟩

Some syntactic sugar:

syntax (*ASCII*)
  -prod-list :: *pttrn* => *'a list* => *'b* => *'b*   ((*3PROD* -<--. -) [*0*, *51*, *10*] *10*)
syntax
  -prod-list :: *pttrn* => *'a list* => *'b* => *'b*   ((*3*∏-←-. -) [*0*, *51*, *10*] *10*)
translations — Beware of argument permutation!
  $\prod x{\leftarrow}xs.\ b \rightleftharpoons CONST\ prod\text{-}list\ (CONST\ map\ (\lambda x.\ b)\ xs)$

end

69 A HOL random engine

theory *Random*
imports *List Groups-List*
begin

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

69.1 Auxiliary functions

fun *log* :: *natural* ⇒ *natural* ⇒ *natural* **where**
  $log\ b\ i = (if\ b \leq 1 \lor i < b\ then\ 1\ else\ 1 + log\ b\ (i\ div\ b))$

definition *inc-shift* :: *natural* ⇒ *natural* ⇒ *natural* **where**
  $inc\text{-}shift\ v\ k = (if\ v = k\ then\ 1\ else\ k + 1)$

definition *minus-shift* :: *natural* ⇒ *natural* ⇒ *natural* ⇒ *natural* **where**
  $minus\text{-}shift\ r\ k\ l = (if\ k < l\ then\ r + k - l\ else\ k - l)$

69.2 Random seeds

type-synonym *seed* = *natural* × *natural*

primrec *next* :: *seed* ⇒ *natural* × *seed* **where**
  $next\ (v,\ w) = (let$
    $k = \ v\ div\ 53668;$
    $v' = minus\text{-}shift\ 2147483563\ ((v\ mod\ 53668) * 40014)\ (k * 12211);$

$l = w\ div\ 52774;$
$w' = minus\text{-}shift\ 2147483399\ ((w\ mod\ 52774) * 40692)\ (l * 3791);$
$z = minus\text{-}shift\ 2147483562\ v'\ (w' + 1) + 1$
$in\ (z, (v', w')))$

definition *split-seed* :: *seed* ⇒ *seed* × *seed* **where**
$split\text{-}seed\ s = (let$
$(v, w) = s;$
$(v', w') = snd\ (next\ s);$
$v'' = inc\text{-}shift\ 2147483562\ v;$
$w'' = inc\text{-}shift\ 2147483398\ w$
$in\ ((v'', w'), (v', w'')))$

69.3 Base selectors

fun *iterate* :: $natural \Rightarrow ('b \Rightarrow 'a \Rightarrow 'b \times 'a) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \times 'a$ **where**
$iterate\ k\ f\ x = (if\ k = 0\ then\ Pair\ x\ else\ f\ x \circ\!\!\rightarrow iterate\ (k - 1)\ f)$

definition *range* :: $natural \Rightarrow seed \Rightarrow natural \times seed$ **where**
$range\ k = iterate\ (log\ 2147483561\ k)$
$(\lambda l.\ next \circ\!\!\rightarrow (\lambda v.\ Pair\ (v + l * 2147483561)))\ 1$
$\circ\!\!\rightarrow (\lambda v.\ Pair\ (v\ mod\ k))$

lemma *range*:
$k > 0 \Longrightarrow fst\ (range\ k\ s) < k$
⟨*proof*⟩

definition *select* :: $'a\ list \Rightarrow seed \Rightarrow 'a \times seed$ **where**
$select\ xs = range\ (natural\text{-}of\text{-}nat\ (length\ xs))$
$\circ\!\!\rightarrow (\lambda k.\ Pair\ (nth\ xs\ (nat\text{-}of\text{-}natural\ k)))$

lemma *select*:
assumes $xs \neq []$
shows $fst\ (select\ xs\ s) \in set\ xs$
⟨*proof*⟩

primrec *pick* :: $(natural \times 'a)\ list \Rightarrow natural \Rightarrow 'a$ **where**
$pick\ (x\ \#\ xs)\ i = (if\ i < fst\ x\ then\ snd\ x\ else\ pick\ xs\ (i - fst\ x))$

lemma *pick-member*:
$i < sum\text{-}list\ (map\ fst\ xs) \Longrightarrow pick\ xs\ i \in set\ (map\ snd\ xs)$
⟨*proof*⟩

lemma *pick-drop-zero*:
$pick\ (filter\ (\lambda(k, \text{-}).\ k > 0)\ xs) = pick\ xs$
⟨*proof*⟩

lemma *pick-same*:
$l < length\ xs \Longrightarrow Random.pick\ (map\ (Pair\ 1)\ xs)\ (natural\text{-}of\text{-}nat\ l) = nth\ xs\ l$

⟨*proof*⟩

definition *select-weight* :: $(natural \times 'a)$ *list* $\Rightarrow$ *seed* $\Rightarrow$ $'a \times$ *seed* **where**
select-weight xs = *range* (*sum-list* (*map fst xs*))
$\circ\!\!\rightarrow$ (λk. *Pair* (*pick xs k*))

lemma *select-weight-member*:
assumes *0* < *sum-list* (*map fst xs*)
shows *fst* (*select-weight xs s*) $\in$ *set* (*map snd xs*)
⟨*proof*⟩

lemma *select-weight-cons-zero*:
select-weight ((*0*, *x*) # *xs*) = *select-weight xs*
⟨*proof*⟩

lemma *select-weight-drop-zero*:
select-weight (*filter* (λ(*k*, -). *k* > *0*) *xs*) = *select-weight xs*
⟨*proof*⟩

lemma *select-weight-select*:
assumes *xs* $\neq$ []
shows *select-weight* (*map* (*Pair 1*) *xs*) = *select xs*
⟨*proof*⟩

69.4 *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** $\circ>$ *60*)
no-notation *scomp* (**infixl** $\circ\!\!\rightarrow$ *60*)

end

70 Maps

theory *Map*
imports *List*
begin

type-synonym ('*a*, '*b*) *map* = '*a* $\Rightarrow$ '*b option* (**infixr** $\rightharpoonup$ *0*)

abbreviation
empty :: $'a \rightharpoonup 'b$ **where**
$empty \equiv \lambda x.\ None$

definition
map-comp :: $('b \rightharpoonup 'c) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'c)$ (**infixl** $\circ_m$ *55*) **where**
$f \circ_m g = (\lambda k.\ case\ g\ k\ of\ None \Rightarrow None \mid Some\ v \Rightarrow f\ v)$

definition
map-add :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ (**infixl** ++ *100*) **where**
$m1 \mathbin{++} m2 = (\lambda x.\ case\ m2\ x\ of\ None \Rightarrow m1\ x \mid Some\ y \Rightarrow Some\ y)$

definition
restrict-map :: $('a \rightharpoonup 'b) \Rightarrow 'a\ set \Rightarrow ('a \rightharpoonup 'b)$ (**infixl** |' *110*) **where**
$m|`A = (\lambda x.\ if\ x \in A\ then\ m\ x\ else\ None)$

notation (*latex* **output**)
restrict-map (-$\restriction$- [*111*,*110*] *110*)

definition
dom :: $('a \rightharpoonup 'b) \Rightarrow 'a\ set$ **where**
$dom\ m = \{a.\ m\ a \neq None\}$

definition
ran :: $('a \rightharpoonup 'b) \Rightarrow 'b\ set$ **where**
$ran\ m = \{b.\ \exists\, a.\ m\ a = Some\ b\}$

definition
map-le :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$ (**infix** $\subseteq_m$ *50*) **where**
$(m_1 \subseteq_m m_2) \longleftrightarrow (\forall\, a \in dom\ m_1.\ m_1\ a = m_2\ a)$

nonterminal *maplets* **and** *maplet*

syntax
-maplet :: $['a, 'a] \Rightarrow maplet$ (- /$\mapsto$/ -)
-maplets :: $['a, 'a] \Rightarrow maplet$ (- /[$\mapsto$]/ -)
:: $maplet \Rightarrow maplets$ (-)
-Maplets :: $[maplet, maplets] \Rightarrow maplets$ (-,/ -)
-MapUpd :: $['a \rightharpoonup 'b, maplets] \Rightarrow 'a \rightharpoonup 'b$ (-/'(-') [*900*, *0*] *900*)
-Map :: $maplets \Rightarrow 'a \rightharpoonup 'b$ ((*1*[-]))

syntax (*ASCII*)
-maplet :: $['a, 'a] \Rightarrow maplet$ (- /|->/ -)
-maplets :: $['a, 'a] \Rightarrow maplet$ (- /[|->]/ -)

translations
-MapUpd m (*-Maplets xy ms*) $\rightleftharpoons$ *-MapUpd* (*-MapUpd m xy*) *ms*
-MapUpd m (*-maplet x y*) $\rightleftharpoons$ $m(x := CONST\ Some\ y)$
-Map ms $\rightleftharpoons$ *-MapUpd* (*CONST empty*) *ms*

$-Map\ (-Maplets\ ms1\ ms2) \leftharpoondown -MapUpd\ (-Map\ ms1)\ ms2$
$-Maplets\ ms1\ (-Maplets\ ms2\ ms3) \leftharpoondown -Maplets\ (-Maplets\ ms1\ ms2)\ ms3$

primrec *map-of* :: $('a \times 'b)\ list \Rightarrow 'a \rightharpoonup 'b$
where
map-of [] = *empty*
| *map-of* (*p* # *ps*) = (*map-of ps*)(*fst p* $\mapsto$ *snd p*)

definition *map-upds* :: $('a \rightharpoonup 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'a \rightharpoonup 'b$
where *map-upds m xs ys* = *m* ++ *map-of* (*rev* (*zip xs ys*))
translations
-MapUpd m (*-maplets x y*) $\rightleftharpoons$ *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*⟩

70.1 *empty*

lemma *empty-upd-none* [*simp*]: *empty*(*x* := *None*) = *empty*
⟨*proof*⟩

70.2 *map-upd*

lemma *map-upd-triv*: *t k* = *Some x* $\Longrightarrow$ *t*(*k*$\mapsto$*x*) = *t*
⟨*proof*⟩

lemma *map-upd-nonempty* [*simp*]: *t*(*k*$\mapsto$*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*$\mapsto$*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*$\mapsto$*b*)))
⟨*proof*⟩

70.3 *map-of*

lemma *map-of-eq-None-iff*:
(*map-of xys x* = *None*) = (*x* $\notin$ *fst* ` (*set xys*))

$\langle proof \rangle$

lemma *map-of-eq-Some-iff* [*simp*]:
  $distinct(map\ fst\ xys) \Longrightarrow (map\text{-}of\ xys\ x = Some\ y) = ((x,y) \in set\ xys)$
$\langle proof \rangle$

lemma *Some-eq-map-of-iff* [*simp*]:
  $distinct(map\ fst\ xys) \Longrightarrow (Some\ y = map\text{-}of\ xys\ x) = ((x,y) \in set\ xys)$
$\langle proof \rangle$

lemma *map-of-is-SomeI* [*simp*]: $\llbracket\ distinct(map\ fst\ xys);\ (x,y) \in set\ xys\ \rrbracket$
  $\Longrightarrow map\text{-}of\ xys\ x = Some\ y$
$\langle proof \rangle$

lemma *map-of-zip-is-None* [*simp*]:
  $length\ xs = length\ ys \Longrightarrow (map\text{-}of\ (zip\ xs\ ys)\ x = None) = (x \notin set\ xs)$
$\langle proof \rangle$

lemma *map-of-zip-is-Some*:
  assumes $length\ xs = length\ ys$
  shows $x \in set\ xs \longleftrightarrow (\exists\, y.\ map\text{-}of\ (zip\ xs\ ys)\ x = Some\ y)$
$\langle proof \rangle$

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 $map\text{-}of\ (zip\ xs\ ys) = map\text{-}of\ (zip\ xs\ zs)$
$\langle proof \rangle$

lemma *map-of-zip-inject*:
  assumes $length\ ys = length\ xs$
    and $length\ zs = length\ xs$
    and *dist*: $distinct\ xs$
    and *map-of*: $map\text{-}of\ (zip\ xs\ ys) = map\text{-}of\ (zip\ xs\ zs)$
  shows $ys = zs$
  $\langle proof \rangle$

lemma *map-of-zip-nth*:
  assumes $length\ xs = length\ ys$
  assumes $distinct\ xs$
  assumes $i < length\ ys$
  shows $map\text{-}of\ (zip\ xs\ ys)\ (xs\ !\ i) = Some\ (ys\ !\ i)$
$\langle proof \rangle$

lemma *map-of-zip-map*:
  $map\text{-}of\ (zip\ xs\ (map\ f\ xs)) = (\lambda x.\ if\ x \in set\ xs\ then\ Some\ (f\ x)\ else\ None)$

⟨*proof*⟩

lemma *finite-range-map-of*: *finite* (*range* (*map-of xys*))
⟨*proof*⟩

lemma *map-of-SomeD*: *map-of xs k* = *Some y* $\Longrightarrow$ (*k*, *y*) $\in$ *set xs*
⟨*proof*⟩

lemma *map-of-mapk-SomeI*:
inj f $\Longrightarrow$ *map-of t k* = *Some x* $\Longrightarrow$
map-of (*map* (*case-prod* (λ*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
⟨*proof*⟩

lemma *weak-map-of-SomeI*: (*k*, *x*) $\in$ *set l* $\Longrightarrow$ $\exists$ *x*. *map-of l k* = *Some x*
⟨*proof*⟩

lemma *map-of-filter-in*:
map-of xs k = *Some z* $\Longrightarrow$ *P k z* $\Longrightarrow$ *map-of* (*filter* (*case-prod P*) *xs*) *k* = *Some*
z
⟨*proof*⟩

lemma *map-of-map*:
map-of (*map* (λ(*k*, *v*). (*k*, *f v*)) *xs*) = *map-option f* $\circ$ *map-of xs*
⟨*proof*⟩

lemma *dom-map-option*:
dom (λ*k*. *map-option* (*f k*) (*m k*)) = *dom m*
⟨*proof*⟩

lemma *dom-map-option-comp* [*simp*]:
dom (*map-option g* $\circ$ *m*) = *dom m*
⟨*proof*⟩

70.4 *map-option* related

lemma *map-option-o-empty* [*simp*]: *map-option f o empty* = *empty*
⟨*proof*⟩

lemma *map-option-o-map-upd* [*simp*]:
map-option f o m($a \mapsto b$) = (*map-option f o m*)($a \mapsto f\ b$)
⟨*proof*⟩

70.5 *map-comp* related

lemma *map-comp-empty* [*simp*]:
m $\circ_m$ *empty* = *empty*
empty $\circ_m$ *m* = *empty*
⟨*proof*⟩

lemma *map-comp-simps* [*simp*]:

$m2\ k = None \Longrightarrow (m1 \circ_m m2)\ k = None$
$m2\ k = Some\ k' \Longrightarrow (m1 \circ_m m2)\ k = m1\ k'$
⟨proof⟩

lemma *map-comp-Some-iff*:
$((m1 \circ_m m2)\ k = Some\ v) = (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = Some\ v)$
⟨proof⟩

lemma *map-comp-None-iff*:
$((m1 \circ_m m2)\ k = None) = (m2\ k = None \vee (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = None))$
⟨proof⟩

70.6 ++

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*:
$((m ++ n)\ k = Some\ x) = (n\ k = Some\ x \mid n\ k = None\ \&\ m\ k = Some\ x)$
⟨proof⟩

lemma *map-add-SomeD* [*dest*!]:
$(m ++ n)\ k = Some\ x \Longrightarrow n\ k = Some\ x \vee n\ k = None \wedge m\ k = Some\ x$
⟨proof⟩

lemma *map-add-find-right* [*simp*]: $n\ k = Some\ xx \Longrightarrow (m ++ n)\ k = Some\ xx$
⟨proof⟩

lemma *map-add-None* [*iff*]: $((m ++ n)\ k = None) = (n\ k = None\ \&\ m\ k = None)$
⟨proof⟩

lemma *map-add-upd*[*simp*]: $f ++ g(x \mapsto y) = (f ++ g)(x \mapsto y)$
⟨proof⟩

lemma *map-add-upds*[*simp*]: $m1 ++ (m2(xs[\mapsto]ys)) = (m1++m2)(xs[\mapsto]ys)$
⟨proof⟩

lemma *map-add-upd-left*: $m \notin dom\ e2 \Longrightarrow e1(m \mapsto u1) ++ e2 = (e1 ++ e2)(m \mapsto u1)$
⟨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*) $\Longrightarrow$ *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*:
$m(ks[\mapsto]vs)$ = *foldl* (λm (k, v). $m(k \mapsto v)$) m (*zip ks vs*)
⟨*proof*⟩

lemma *map-add-map-of-foldr*:
m ++ *map-of ps* = *foldr* (λ(k, v) m. $m(k \mapsto v)$) *ps* m
⟨*proof*⟩

70.7 *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*]:

$m(x := y)|`D = (\mathit{if}\ x \in D\ \mathit{then}\ (m|`(D-\{x\}))(x := y)\ \mathit{else}\ m|`D)$
⟨proof⟩

lemma *fun-upd-None-restrict* [*simp*]:
$(m|`D)(x := \mathit{None}) = (\mathit{if}\ x \in D\ \mathit{then}\ m|`(D - \{x\})\ \mathit{else}\ m|`D)$
⟨proof⟩

lemma *fun-upd-restrict*: $(m|`D)(x := y) = (m|`(D-\{x\}))(x := y)$
⟨proof⟩

lemma *fun-upd-restrict-conv* [*simp*]:
$x \in D \Longrightarrow (m|`D)(x := y) = (m|`(D-\{x\}))(x := y)$
⟨proof⟩

lemma *map-of-map-restrict*:
$\mathit{map\text{-}of}\ (\mathit{map}\ (\lambda k.\ (k, f\ k))\ ks) = (\mathit{Some} \circ f)\ |`\ \mathit{set}\ ks$
⟨proof⟩

lemma *restrict-complement-singleton-eq*:
$f\ |`\ (-\ \{x\}) = f(x := \mathit{None})$
⟨proof⟩

70.8 *map-upds*

lemma *map-upds-Nil1* [*simp*]: $m([]\ [\mapsto]\ bs) = m$
⟨proof⟩

lemma *map-upds-Nil2* [*simp*]: $m(as\ [\mapsto]\ []) = m$
⟨proof⟩

lemma *map-upds-Cons* [*simp*]: $m(a\#as\ [\mapsto]\ b\#bs) = (m(a\mapsto b))(as[\mapsto]bs)$
⟨proof⟩

lemma *map-upds-append1* [*simp*]: $\mathit{size}\ xs < \mathit{size}\ ys \Longrightarrow$
$m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys)(x \mapsto ys!\mathit{size}\ xs)$
⟨proof⟩

lemma *map-upds-list-update2-drop* [*simp*]:
$\mathit{size}\ xs \le i \Longrightarrow m(xs[\mapsto]ys[i:=y]) = m(xs[\mapsto]ys)$
⟨proof⟩

lemma *map-upd-upds-conv-if*:
$(f(x\mapsto y))(xs\ [\mapsto]\ ys) =$
$(\mathit{if}\ x \in \mathit{set}(\mathit{take}\ (\mathit{length}\ ys)\ xs)\ \mathit{then}\ f(xs\ [\mapsto]\ ys)$
$\mathit{else}\ (f(xs\ [\mapsto]\ ys))(x\mapsto y))$
⟨proof⟩

lemma *map-upds-twist* [*simp*]:
$a \notin \mathit{set}\ as \Longrightarrow m(a\mapsto b)(as[\mapsto]bs) = m(as[\mapsto]bs)(a\mapsto b)$

⟨*proof*⟩

lemma *map-upds-apply-nontin* [*simp*]:
$x \notin set\ xs \Longrightarrow (f(xs[\mapsto]ys))\ x = f\ x$
⟨*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*⟩

70.9 *dom*

lemma *dom-eq-empty-conv* [*simp*]: $dom\ f = \{\} \longleftrightarrow f = empty$
⟨*proof*⟩

lemma *domI*: $m\ a = Some\ b \Longrightarrow a \in dom\ m$
⟨*proof*⟩

lemma *domD*: $a \in dom\ m \Longrightarrow \exists\, b.\ m\ a = Some\ b$
⟨*proof*⟩

lemma *domIff* [*iff*, *simp del*, *code-unfold*]: $a \in dom\ m \longleftrightarrow m\ a \neq None$
⟨*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*]: $length\ xs = length\ ys \Longrightarrow dom\ (map\text{-}of\ (zip\ xs\ ys)) = set\ xs$
⟨*proof*⟩

lemma *finite-dom-map-of*: $finite\ (dom\ (map\text{-}of\ l))$
⟨*proof*⟩

lemma *dom-map-upds* [*simp*]:
$dom(m(xs[\mapsto]ys)) = set(take\ (length\ ys)\ xs) \cup dom\ m$
⟨*proof*⟩

lemma *dom-map-add* [*simp*]: $dom\ (m\ {+}{+}\ n) = dom\ n \cup dom\ m$
⟨*proof*⟩

lemma *dom-override-on* [*simp*]:
$dom\ (override\text{-}on\ f\ g\ A) =$
$(dom\ f\ - \{a.\ a \in A - dom\ g\}) \cup \{a.\ a \in A \cap dom\ g\}$
⟨*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*]:
$dom\ (\lambda x.\ Some\ (f\ x)) = UNIV$
⟨*proof*⟩

lemma *finite-map-freshness*:
$finite\ (dom\ (f :: 'a \rightharpoonup 'b)) \Longrightarrow \neg\ finite\ (UNIV :: 'a\ set) \Longrightarrow$
$\exists x.\ f\ x = None$
⟨*proof*⟩

lemma *dom-minus*:
$f\ x = None \Longrightarrow dom\ f - insert\ x\ A = dom\ f - A$
⟨*proof*⟩

lemma *insert-dom*:
$f\ x = Some\ y \Longrightarrow insert\ x\ (dom\ f) = dom\ f$
⟨*proof*⟩

lemma *map-of-map-keys*:
$set\ xs = dom\ m \Longrightarrow map\text{-}of\ (map\ (\lambda k.\ (k,\ the\ (m\ k)))\ xs) = m$

⟨*proof*⟩

lemma *map-of-eqI*:
assumes *set-eq*: *set* (*map fst xs*) = *set* (*map fst ys*)
assumes *map-eq*: $\forall k{\in}set$ (*map fst xs*). *map-of xs k* = *map-of ys k*
shows *map-of xs* = *map-of ys*
⟨*proof*⟩

lemma *map-of-eq-dom*:
assumes *map-of xs* = *map-of ys*
shows *fst ' set xs* = *fst ' set ys*
⟨*proof*⟩

lemma *finite-set-of-finite-maps*:
assumes *finite A finite B*
shows *finite* {*m*. *dom m* = *A* ∧ *ran m* ⊆ *B*} (**is** *finite ?S*)
⟨*proof*⟩

70.10 *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-map-add*:
assumes *dom m1* ∩ *dom m2* = {}
shows *ran* (*m1* ++ *m2*) = *ran m1* ∪ *ran m2*
⟨*proof*⟩

lemma *finite-ran*:
assumes *finite* (*dom p*)
shows *finite* (*ran p*)
⟨*proof*⟩

lemma *ran-distinct*:
assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd ' set al*
⟨*proof*⟩

lemma *ran-map-of-zip*:
assumes *length xs* = *length ys distinct xs*
shows *ran* (*map-of* (*zip xs ys*)) = *set ys*
⟨*proof*⟩

lemma *ran-map-option*: $ran\ (\lambda x.\ map\text{-}option\ f\ (m\ x)) = f\ `\ ran\ m$
⟨*proof*⟩

70.11 *map-le*

lemma *map-le-empty* [*simp*]: $empty \subseteq_m g$
⟨*proof*⟩

lemma *upd-None-map-le* [*simp*]: $f(x := None) \subseteq_m f$
⟨*proof*⟩

lemma *map-le-upd*[*simp*]: $f \subseteq_m g ==> f(a := b) \subseteq_m g(a := b)$
⟨*proof*⟩

lemma *map-le-imp-upd-le* [*simp*]: $m1 \subseteq_m m2 \Longrightarrow m1(x := None) \subseteq_m m2(x \mapsto y)$
⟨*proof*⟩

lemma *map-le-upds* [*simp*]:
$f \subseteq_m g \Longrightarrow f(as\ [\mapsto]\ bs) \subseteq_m g(as\ [\mapsto]\ bs)$
⟨*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*: $f \subseteq_m f\ {++}\ g \longleftrightarrow f\ {++}\ g = g\ {++}\ f$
⟨*proof*⟩

lemma *map-add-le-mapE*: $f\ {++}\ g \subseteq_m h \Longrightarrow g \subseteq_m h$
⟨*proof*⟩

lemma *map-add-le-mapI*: $[\![\ f \subseteq_m h;\ g \subseteq_m h\ ]\!] \Longrightarrow f\ {++}\ g \subseteq_m h$
⟨*proof*⟩

lemma *map-add-subsumed1*: $f \subseteq_m g \Longrightarrow f{++}g = g$
⟨*proof*⟩

lemma *map-add-subsumed2*: $f \subseteq_m g \Longrightarrow g{+}{+}f = g$
⟨*proof*⟩

lemma *dom-eq-singleton-conv*: $dom\ f = \{x\} \longleftrightarrow (\exists\, v.\ f = [x \mapsto v])$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

70.12 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

71 Finite types as explicit enumerations

theory *Enum*
imports *Map Groups-List*
begin

71.1 Class *enum*

class *enum* =
fixes *enum* :: *'a list*
fixes *enum-all* :: (*'a* ⇒ *bool*) ⇒ *bool*
fixes *enum-ex* :: (*'a* ⇒ *bool*) ⇒ *bool*
assumes *UNIV-enum*: *UNIV* = *set enum*
and *enum-distinct*: *distinct enum*
assumes *enum-all-UNIV*: *enum-all P* ⟷ *Ball UNIV P*
assumes *enum-ex-UNIV*: *enum-ex P* ⟷ *Bex UNIV P*
— tailored towards simple instantiation
begin

subclass *finite* ⟨*proof*⟩

lemma *enum-UNIV*:
set enum = *UNIV*
⟨*proof*⟩

lemma *in-enum*: $x \in$ *set enum*
⟨*proof*⟩

lemma *enum-eq-I*:
assumes $\bigwedge x.\ x \in set\ xs$
shows *set enum* = *set xs*
⟨*proof*⟩

lemma *card-UNIV-length-enum*:
card (*UNIV* :: *'a set*) = *length enum*
⟨*proof*⟩

lemma *enum-all* [*simp*]:
enum-all = *HOL.All*
⟨*proof*⟩

lemma *enum-ex* [*simp*]:
enum-ex = *HOL.Ex*
⟨*proof*⟩

end

71.2 Implementations using *enum*

71.2.1 Unbounded operations and quantifiers

lemma *Collect-code* [*code*]:
Collect P = *set* (*filter P enum*)
⟨*proof*⟩

lemma *vimage-code* [*code*]:
f −' *B* = *set* (*filter* (%*x*. *f x* : *B*) *enum-class.enum*)
⟨*proof*⟩

definition *card-UNIV* :: *'a itself* ⇒ *nat*
where
[*code del*]: *card-UNIV TYPE*(*'a*) = *card* (*UNIV* :: *'a set*)

lemma [*code*]:
card-UNIV TYPE(*'a* :: *enum*) = *card* (*set* (*Enum.enum* :: *'a list*))
⟨*proof*⟩

lemma *all-code* [*code*]: $(\forall x.\ P\ x) \longleftrightarrow$ *enum-all P*
⟨*proof*⟩

lemma *exists-code* [*code*]: $(\exists x.\ P\ x) \longleftrightarrow$ *enum-ex P*
⟨*proof*⟩

lemma *exists1-code* [*code*]: $(\exists !x.\ P\ x) \longleftrightarrow$ *list-ex1 P enum*
⟨*proof*⟩

71.2.2 An executable choice operator

definition
$[code\ del]$: $enum\text{-}the = The$

lemma $[code]$:
$The\ P = (case\ filter\ P\ enum\ of\ [x] => x \mid \text{-} => enum\text{-}the\ P)$
$\langle proof \rangle$

declare $[[code\ abort: enum\text{-}the]]$

code-printing
constant $enum\text{-}the \rightharpoonup (Eval)\ (fn\ '\text{-} => raise\ Match)$

71.2.3 Equality and order on functions

instantiation $fun :: (enum, equal)\ equal$
begin

definition
$HOL.equal\ f\ g \longleftrightarrow (\forall x \in set\ enum.\ f\ x = g\ x)$

instance $\langle proof \rangle$

end

lemma $[code]$:
$HOL.equal\ f\ g \longleftrightarrow enum\text{-}all\ (\%x.\ f\ x = g\ x)$
$\langle proof \rangle$

lemma $[code\ nbe]$:
$HOL.equal\ (f :: \text{-} \Rightarrow \text{-})\ f \longleftrightarrow True$
$\langle proof \rangle$

lemma $order\text{-}fun\ [code]$:
fixes $f\ g :: 'a::enum \Rightarrow 'b::order$
shows $f \le g \longleftrightarrow enum\text{-}all\ (\lambda x.\ f\ x \le g\ x)$
and $f < g \longleftrightarrow f \le g \wedge enum\text{-}ex\ (\lambda x.\ f\ x \neq g\ x)$
$\langle proof \rangle$

71.2.4 Operations on relations

lemma $[code]$:
$Id = image\ (\lambda x.\ (x, x))\ (set\ Enum.enum)$
$\langle proof \rangle$

lemma $tranclp\text{-}unfold\ [code]$:
$tranclp\ r\ a\ b \longleftrightarrow (a, b) \in trancl\ \{(x, y).\ r\ x\ y\}$
$\langle proof \rangle$

lemma *rtranclp-rtrancl-eq* [*code*]:
$rtranclp\ r\ x\ y \longleftrightarrow (x,\ y) \in rtrancl\ \{(x,\ y).\ r\ x\ y\}$
⟨*proof*⟩

lemma *max-ext-eq* [*code*]:
$max\text{-}ext\ R = \{(X,\ Y).\ finite\ X \wedge finite\ Y \wedge Y \neq \{\} \wedge (\forall x.\ x \in X \longrightarrow (\exists xa \in Y.\ (x,\ xa) \in R))\}$
⟨*proof*⟩

lemma *max-extp-eq* [*code*]:
$max\text{-}extp\ r\ x\ y \longleftrightarrow (x,\ y) \in max\text{-}ext\ \{(x,\ y).\ r\ x\ y\}$
⟨*proof*⟩

lemma *mlex-eq* [*code*]:
$f <*mlex*> R = \{(x,\ y).\ f\ x < f\ y \vee (f\ x \leq f\ y \wedge (x,\ y) \in R)\}$
⟨*proof*⟩

71.2.5 Bounded accessible part

primrec $bacc :: ('a \times 'a)\ set \Rightarrow nat \Rightarrow 'a\ set$
where
$bacc\ r\ 0 = \{x.\ \forall\ y.\ (y,\ x) \notin r\}$
| $bacc\ r\ (Suc\ n) = (bacc\ r\ n \cup \{x.\ \forall y.\ (y,\ x) \in r \longrightarrow y \in bacc\ r\ n\})$

lemma *bacc-subseteq-acc*:
$bacc\ r\ n \subseteq Wellfounded.acc\ r$
⟨*proof*⟩

lemma *bacc-mono*:
$n \leq m \Longrightarrow bacc\ r\ n \subseteq bacc\ r\ m$
⟨*proof*⟩

lemma *bacc-upper-bound*:
$bacc\ (r :: ('a \times 'a)\ set)\ \ (card\ (UNIV :: 'a::finite\ set)) = (\bigcup n.\ bacc\ r\ n)$
⟨*proof*⟩

lemma *acc-subseteq-bacc*:
assumes *finite r*
shows $Wellfounded.acc\ r \subseteq (\bigcup n.\ bacc\ r\ n)$
⟨*proof*⟩

lemma *acc-bacc-eq*:
fixes $A :: ('a :: finite \times 'a)\ set$
assumes *finite A*
shows $Wellfounded.acc\ A = bacc\ A\ (card\ (UNIV :: 'a\ set))$
⟨*proof*⟩

lemma [*code*]:
fixes $xs :: ('a::finite \times 'a)\ list$

shows *Wellfounded.acc (set xs) = bacc (set xs) (card-UNIV TYPE('a))*
⟨*proof*⟩

71.3 Default instances for *enum*

lemma *map-of-zip-enum-is-Some*:
assumes *length ys = length (enum :: 'a::enum list)*
shows *∃ y. map-of (zip (enum :: 'a::enum list) ys) x = Some y*
⟨*proof*⟩

lemma *map-of-zip-enum-inject*:
fixes *xs ys :: 'b::enum list*
assumes *length: length xs = length (enum :: 'a::enum list)*
length ys = length (enum :: 'a::enum list)
and *map-of: the ∘ map-of (zip (enum :: 'a::enum list) xs) = the ∘ map-of (zip (enum :: 'a::enum list) ys)*
shows *xs = ys*
⟨*proof*⟩

definition *all-n-lists :: (('a :: enum) list ⇒ bool) ⇒ nat ⇒ bool*
where
all-n-lists P n ⟷ (∀ xs ∈ set (List.n-lists n enum). P xs)

lemma [*code*]:
all-n-lists P n ⟷ (if n = 0 then P [] else enum-all (%x. all-n-lists (%xs. P (x # xs)) (n − 1)))
⟨*proof*⟩

definition *ex-n-lists :: (('a :: enum) list ⇒ bool) ⇒ nat ⇒ bool*
where
ex-n-lists P n ⟷ (∃ xs ∈ set (List.n-lists n enum). P xs)

lemma [*code*]:
ex-n-lists P n ⟷ (if n = 0 then P [] else enum-ex (%x. ex-n-lists (%xs. P (x # xs)) (n − 1)))
⟨*proof*⟩

instantiation *fun :: (enum, enum) enum*
begin

definition
enum = map (λys. the o map-of (zip (enum::'a list) ys)) (List.n-lists (length (enum::'a::enum list)) enum)

definition
enum-all P = all-n-lists (λbs. P (the o map-of (zip enum bs))) (length (enum :: 'a list))

definition

enum-ex P = *ex-n-lists* (λ*bs*. *P* (*the o map-of* (*zip enum bs*))) (*length* (*enum* :: '*a list*))

instance ⟨*proof*⟩

end

lemma *enum-fun-code* [*code*]: *enum* = (*let enum-a* = (*enum* :: '*a*::{*enum*, *equal*} *list*)
in map (λ*ys*. *the o map-of* (*zip enum-a ys*)) (*List.n-lists* (*length enum-a*) *enum*))
⟨*proof*⟩

lemma *enum-all-fun-code* [*code*]:
enum-all P = (*let enum-a* = (*enum* :: '*a*::{*enum*, *equal*} *list*)
in all-n-lists (λ*bs*. *P* (*the o map-of* (*zip enum-a bs*))) (*length enum-a*))
⟨*proof*⟩

lemma *enum-ex-fun-code* [*code*]:
enum-ex P = (*let enum-a* = (*enum* :: '*a*::{*enum*, *equal*} *list*)
in ex-n-lists (λ*bs*. *P* (*the o map-of* (*zip enum-a bs*))) (*length enum-a*))
⟨*proof*⟩

instantiation *set* :: (*enum*) *enum*
begin

definition
enum = *map set* (*subseqs enum*)

definition
enum-all P ⟷ ($\forall$ *A*∈*set enum*. *P* (*A*::'*a set*))

definition
enum-ex P ⟷ ($\exists$ *A*∈*set enum*. *P* (*A*::'*a set*))

instance ⟨*proof*⟩

end

instantiation *unit* :: *enum*
begin

definition
enum = [()]

definition
enum-all P = *P* ()

definition
enum-ex P = *P* ()

instance ⟨*proof*⟩

end

instantiation *bool* :: *enum*
begin

definition
  $enum = [False,\ True]$

definition
  $enum\text{-}all\ P \longleftrightarrow P\ False \wedge P\ True$

definition
  $enum\text{-}ex\ P \longleftrightarrow P\ False \vee P\ True$

instance ⟨*proof*⟩

end

instantiation *prod* :: (*enum*, *enum*) *enum*
begin

definition
  $enum = List.product\ enum\ enum$

definition
  $enum\text{-}all\ P = enum\text{-}all\ (\%x.\ enum\text{-}all\ (\%y.\ P\ (x,\ y)))$

definition
  $enum\text{-}ex\ P = enum\text{-}ex\ (\%x.\ enum\text{-}ex\ (\%y.\ P\ (x,\ y)))$

instance
  ⟨*proof*⟩

end

instantiation *sum* :: (*enum*, *enum*) *enum*
begin

definition
  $enum = map\ Inl\ enum\ @\ map\ Inr\ enum$

definition
  $enum\text{-}all\ P \longleftrightarrow enum\text{-}all\ (\lambda x.\ P\ (Inl\ x)) \wedge enum\text{-}all\ (\lambda x.\ P\ (Inr\ x))$

definition

$enum\text{-}ex\ P \longleftrightarrow enum\text{-}ex\ (\lambda x.\ P\ (Inl\ x)) \lor enum\text{-}ex\ (\lambda x.\ P\ (Inr\ x))$

instance ⟨*proof*⟩

end

instantiation *option* :: (*enum*) *enum*
begin

definition
$enum = None\ \#\ map\ Some\ enum$

definition
$enum\text{-}all\ P \longleftrightarrow P\ None \land enum\text{-}all\ (\lambda x.\ P\ (Some\ x))$

definition
$enum\text{-}ex\ P \longleftrightarrow P\ None \lor enum\text{-}ex\ (\lambda x.\ P\ (Some\ x))$

instance ⟨*proof*⟩

end

71.4 Small finite types

We define small finite types for use in Quickcheck

datatype (*plugins only*: *code quickcheck extraction*) *finite-1* =
a_1

notation (**output**) a_1 (a_1)

lemma *UNIV-finite-1*:
$UNIV = \{a_1\}$
⟨*proof*⟩

instantiation *finite-1* :: *enum*
begin

definition
$enum = [a_1]$

definition
$enum\text{-}all\ P = P\ a_1$

definition
$enum\text{-}ex\ P = P\ a_1$

instance ⟨*proof*⟩

end

instantiation *finite-1* :: *linorder*
begin

definition *less-finite-1* :: *finite-1* ⇒ *finite-1* ⇒ *bool*
where
$x < (y :: \mathit{finite\text{-}1}) \longleftrightarrow \mathit{False}$

definition *less-eq-finite-1* :: *finite-1* ⇒ *finite-1* ⇒ *bool*
where
$x \leq (y :: \mathit{finite\text{-}1}) \longleftrightarrow \mathit{True}$

instance
⟨*proof*⟩

end

instance *finite-1* :: {*dense-linorder*, *wellorder*}
⟨*proof*⟩

instantiation *finite-1* :: *complete-lattice*
begin

definition [*simp*]: $\mathit{Inf} = (\lambda\text{-}.\ a_1)$
definition [*simp*]: $\mathit{Sup} = (\lambda\text{-}.\ a_1)$
definition [*simp*]: $\mathit{bot} = a_1$
definition [*simp*]: $\mathit{top} = a_1$
definition [*simp*]: $\mathit{inf} = (\lambda\text{- -}.\ a_1)$
definition [*simp*]: $\mathit{sup} = (\lambda\text{- -}.\ a_1)$

instance ⟨*proof*⟩
end

instance *finite-1* :: *complete-distrib-lattice*
⟨*proof*⟩

instance *finite-1* :: *complete-linorder* ⟨*proof*⟩

lemma *finite-1-eq*: $x = a_1$
⟨*proof*⟩

⟨*ML*⟩

instantiation *finite-1* :: *complete-boolean-algebra*
begin
definition [*simp*]: $\mathit{op}\ - = (\lambda\text{- -}.\ a_1)$
definition [*simp*]: $\mathit{uminus} = (\lambda\text{-}.\ a_1)$
instance ⟨*proof*⟩
end

instantiation *finite-1* ::
{*linordered-ring-strict*, *linordered-comm-semiring-strict*, *ordered-comm-ring*, *ordered-cancel-comm-monoid-diff*, *comm-monoid-mult*, *ordered-ring-abs*, *one*, *modulo*, *sgn*, *inverse*}
begin
definition [*simp*]: $Groups.zero = a_1$
definition [*simp*]: $Groups.one = a_1$
definition [*simp*]: $op + = (\lambda\text{- -. } a_1)$
definition [*simp*]: $op * = (\lambda\text{- -. } a_1)$
definition [*simp*]: $op\ mod = (\lambda\text{- -. } a_1)$
definition [*simp*]: $abs = (\lambda\text{-. } a_1)$
definition [*simp*]: $sgn = (\lambda\text{-. } a_1)$
definition [*simp*]: $inverse = (\lambda\text{-. } a_1)$
definition [*simp*]: $divide = (\lambda\text{- -. } a_1)$

instance ⟨*proof*⟩
end

declare [[*simproc del*: *finite-1-eq*]]
hide-const (**open**) a_1

datatype (*plugins only*: *code quickcheck extraction*) *finite-2* =
$a_1 \mid a_2$

notation (**output**) a_1 (a_1)
notation (**output**) a_2 (a_2)

lemma *UNIV-finite-2*:
$UNIV = \{a_1, a_2\}$
⟨*proof*⟩

instantiation *finite-2* :: *enum*
begin

definition
$enum = [a_1, a_2]$

definition
$enum\text{-}all\ P \longleftrightarrow P\ a_1 \wedge P\ a_2$

definition
$enum\text{-}ex\ P \longleftrightarrow P\ a_1 \vee P\ a_2$

instance ⟨*proof*⟩

end

instantiation *finite-2* :: *linorder*

begin

definition *less-finite-2* :: *finite-2* ⇒ *finite-2* ⇒ *bool*
where
$x < y \longleftrightarrow x = a_1 \wedge y = a_2$

definition *less-eq-finite-2* :: *finite-2* ⇒ *finite-2* ⇒ *bool*
where
$x \leq y \longleftrightarrow x = y \vee x < (y :: \mathit{finite\text{-}2})$

instance
⟨*proof*⟩

end

instance *finite-2* :: *wellorder*
⟨*proof*⟩

instantiation *finite-2* :: *complete-lattice*
begin

definition $\bigsqcap A = (\mathit{if}\ a_1 \in A\ \mathit{then}\ a_1\ \mathit{else}\ a_2)$
definition $\bigsqcup A = (\mathit{if}\ a_2 \in A\ \mathit{then}\ a_2\ \mathit{else}\ a_1)$
definition [*simp*]: $\mathit{bot} = a_1$
definition [*simp*]: $\mathit{top} = a_2$
definition $x \sqcap y = (\mathit{if}\ x = a_1 \vee y = a_1\ \mathit{then}\ a_1\ \mathit{else}\ a_2)$
definition $x \sqcup y = (\mathit{if}\ x = a_2 \vee y = a_2\ \mathit{then}\ a_2\ \mathit{else}\ a_1)$

lemma *neq-finite-2-*a_1*-iff* [*simp*]: $x \neq a_1 \longleftrightarrow x = a_2$
⟨*proof*⟩

lemma *neq-finite-2-*a_1*-iff'* [*simp*]: $a_1 \neq x \longleftrightarrow x = a_2$
⟨*proof*⟩

lemma *neq-finite-2-*a_2*-iff* [*simp*]: $x \neq a_2 \longleftrightarrow x = a_1$
⟨*proof*⟩

lemma *neq-finite-2-*a_2*-iff'* [*simp*]: $a_2 \neq x \longleftrightarrow x = a_1$
⟨*proof*⟩

instance
⟨*proof*⟩
end

instance *finite-2* :: *complete-distrib-lattice*
⟨*proof*⟩

instance *finite-2* :: *complete-linorder* ⟨*proof*⟩

instantiation *finite-2* :: {*field*, *idom-abs-sgn*} **begin**
definition [*simp*]: $0 = a_1$
definition [*simp*]: $1 = a_2$
definition $x + y = (case\ (x,\ y)\ of\ (a_1,\ a_1) \Rightarrow a_1 \mid (a_2,\ a_2) \Rightarrow a_1 \mid \text{-} \Rightarrow a_2)$
definition $uminus = (\lambda x :: finite\text{-}2.\ x)$
definition $op\ - = (op\ + :: finite\text{-}2 \Rightarrow \text{-})$
definition $x * y = (case\ (x,\ y)\ of\ (a_2,\ a_2) \Rightarrow a_2 \mid \text{-} \Rightarrow a_1)$
definition $inverse = (\lambda x :: finite\text{-}2.\ x)$
definition $divide = (op\ * :: finite\text{-}2 \Rightarrow \text{-})$
definition $abs = (\lambda x :: finite\text{-}2.\ x)$
definition $sgn = (\lambda x :: finite\text{-}2.\ x)$
instance
  ⟨*proof*⟩
end

lemma *two-finite-2* [*simp*]:
  $2 = a_1$
  ⟨*proof*⟩

lemma *dvd-finite-2-unfold*:
  $x\ dvd\ y \longleftrightarrow x = a_2 \lor y = a_1$
  ⟨*proof*⟩

instantiation *finite-2* :: {*ring-div*, *normalization-semidom*} **begin**
definition [*simp*]: $normalize = (id :: finite\text{-}2 \Rightarrow \text{-})$
definition [*simp*]: $unit\text{-}factor = (id :: finite\text{-}2 \Rightarrow \text{-})$
definition $x\ mod\ y = (case\ (x,\ y)\ of\ (a_2,\ a_1) \Rightarrow a_2 \mid \text{-} \Rightarrow a_1)$
instance
  ⟨*proof*⟩
end

hide-const (**open**) $a_1\ a_2$

datatype (*plugins only*: *code quickcheck extraction*) *finite-3* =
  $a_1 \mid a_2 \mid a_3$

notation (**output**) a_1 (a_1)
notation (**output**) a_2 (a_2)
notation (**output**) a_3 (a_3)

lemma *UNIV-finite-3*:
  $UNIV = \{a_1,\ a_2,\ a_3\}$
  ⟨*proof*⟩

instantiation *finite-3* :: *enum*
begin

definition

$enum = [a_1, a_2, a_3]$

definition
$enum\text{-}all\ P \longleftrightarrow P\ a_1 \wedge P\ a_2 \wedge P\ a_3$

definition
$enum\text{-}ex\ P \longleftrightarrow P\ a_1 \vee P\ a_2 \vee P\ a_3$

instance ⟨*proof*⟩

end

lemma *finite-3-not-eq-unfold*:
$x \neq a_1 \longleftrightarrow x \in \{a_2, a_3\}$
$x \neq a_2 \longleftrightarrow x \in \{a_1, a_3\}$
$x \neq a_3 \longleftrightarrow x \in \{a_1, a_2\}$
⟨*proof*⟩

instantiation *finite-3* :: *linorder*
begin

definition *less-finite-3* :: *finite-3* ⇒ *finite-3* ⇒ *bool*
where
$x < y = (case\ x\ of\ a_1 \Rightarrow y \neq a_1 \mid a_2 \Rightarrow y = a_3 \mid a_3 \Rightarrow False)$

definition *less-eq-finite-3* :: *finite-3* ⇒ *finite-3* ⇒ *bool*
where
$x \leq y \longleftrightarrow x = y \vee x < (y :: finite\text{-}3)$

instance ⟨*proof*⟩

end

instance *finite-3* :: *wellorder*
⟨*proof*⟩

instantiation *finite-3* :: *complete-lattice*
begin

definition $\bigsqcap A = (if\ a_1 \in A\ then\ a_1\ else\ if\ a_2 \in A\ then\ a_2\ else\ a_3)$
definition $\bigsqcup A = (if\ a_3 \in A\ then\ a_3\ else\ if\ a_2 \in A\ then\ a_2\ else\ a_1)$
definition [*simp*]: $bot = a_1$
definition [*simp*]: $top = a_3$
definition [*simp*]: *inf* = (*min* :: *finite-3* ⇒ -)
definition [*simp*]: *sup* = (*max* :: *finite-3* ⇒ -)

instance
⟨*proof*⟩
end

instance *finite-3* :: *complete-distrib-lattice*
⟨*proof*⟩

instance *finite-3* :: *complete-linorder* ⟨*proof*⟩

instantiation *finite-3* :: {*field*, *idom-abs-sgn*} **begin**
definition [*simp*]: $0 = a_1$
definition [*simp*]: $1 = a_2$
definition
$x + y = (case\ (x, y)\ of$
$(a_1, a_1) \Rightarrow a_1 \mid (a_2, a_3) \Rightarrow a_1 \mid (a_3, a_2) \Rightarrow a_1$
$\mid (a_1, a_2) \Rightarrow a_2 \mid (a_2, a_1) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2$
$\mid - \Rightarrow a_3)$
definition $- x = (case\ x\ of\ a_1 \Rightarrow a_1 \mid a_2 \Rightarrow a_3 \mid a_3 \Rightarrow a_2)$
definition $x - y = x + (- y :: finite\text{-}3)$
definition $x * y = (case\ (x, y)\ of\ (a_2, a_2) \Rightarrow a_2 \mid (a_3, a_3) \Rightarrow a_2 \mid (a_2, a_3) \Rightarrow a_3 \mid (a_3, a_2) \Rightarrow a_3 \mid - \Rightarrow a_1)$
definition $inverse = (\lambda x :: finite\text{-}3.\ x)$
definition $x\ div\ y = x * inverse\ (y :: finite\text{-}3)$
definition $abs = (\lambda x.\ case\ x\ of\ a_3 \Rightarrow a_2 \mid - \Rightarrow x)$
definition $sgn = (\lambda x :: finite\text{-}3.\ x)$
instance
⟨*proof*⟩
end

lemma *two-finite-3* [*simp*]:
$2 = a_3$
⟨*proof*⟩

lemma *dvd-finite-3-unfold*:
$x\ dvd\ y \longleftrightarrow x = a_2 \lor x = a_3 \lor y = a_1$
⟨*proof*⟩

instantiation *finite-3* :: {*ring-div*, *normalization-semidom*} **begin**
definition $normalize\ x = (case\ x\ of\ a_3 \Rightarrow a_2 \mid - \Rightarrow x)$
definition [*simp*]: $unit\text{-}factor = (id :: finite\text{-}3 \Rightarrow -)$
definition $x\ mod\ y = (case\ (x, y)\ of\ (a_2, a_1) \Rightarrow a_2 \mid (a_3, a_1) \Rightarrow a_3 \mid - \Rightarrow a_1)$
instance
⟨*proof*⟩
end

hide-const (**open**) $a_1\ a_2\ a_3$

datatype (*plugins only*: *code quickcheck extraction*) *finite-4* =
$a_1 \mid a_2 \mid a_3 \mid a_4$

notation (**output**) a_1 (a_1)
notation (**output**) a_2 (a_2)
notation (**output**) a_3 (a_3)
notation (**output**) a_4 (a_4)

lemma *UNIV-finite-4*:
$UNIV = \{a_1, a_2, a_3, a_4\}$
$\langle proof \rangle$

instantiation *finite-4* :: *enum*
begin

definition
$enum = [a_1, a_2, a_3, a_4]$

definition
$enum\text{-}all\ P \longleftrightarrow P\ a_1 \wedge P\ a_2 \wedge P\ a_3 \wedge P\ a_4$

definition
$enum\text{-}ex\ P \longleftrightarrow P\ a_1 \vee P\ a_2 \vee P\ a_3 \vee P\ a_4$

instance $\langle proof \rangle$

end

instantiation *finite-4* :: *complete-lattice* **begin**

$a_1 < a_2$,$a_3 < a_4$, but a_2 and a_3 are incomparable.

definition
$x < y \longleftrightarrow (case\ (x, y)\ of$
$(a_1, a_1) \Rightarrow False \mid (a_1, \text{-}) \Rightarrow True$
$\mid (a_2, a_4) \Rightarrow True$
$\mid (a_3, a_4) \Rightarrow True \mid \text{-} \Rightarrow False)$

definition
$x \leq y \longleftrightarrow (case\ (x, y)\ of$
$(a_1, \text{-}) \Rightarrow True$
$\mid (a_2, a_2) \Rightarrow True \mid (a_2, a_4) \Rightarrow True$
$\mid (a_3, a_3) \Rightarrow True \mid (a_3, a_4) \Rightarrow True$
$\mid (a_4, a_4) \Rightarrow True \mid \text{-} \Rightarrow False)$

definition
$\bigsqcap A = (if\ a_1 \in A \vee a_2 \in A \wedge a_3 \in A\ then\ a_1\ else\ if\ a_2 \in A\ then\ a_2\ else\ if\ a_3 \in A\ then\ a_3\ else\ a_4)$
definition
$\bigsqcup A = (if\ a_4 \in A \vee a_2 \in A \wedge a_3 \in A\ then\ a_4\ else\ if\ a_2 \in A\ then\ a_2\ else\ if\ a_3 \in A\ then\ a_3\ else\ a_1)$
definition [*simp*]: $bot = a_1$
definition [*simp*]: $top = a_4$

definition
$x \sqcap y = (case\ (x,\ y)\ of$
$(a_1,\ \text{-}) \Rightarrow a_1 \mid (\text{-},\ a_1) \Rightarrow a_1 \mid (a_2,\ a_3) \Rightarrow a_1 \mid (a_3,\ a_2) \Rightarrow a_1$
$\mid (a_2,\ \text{-}) \Rightarrow a_2 \mid (\text{-},\ a_2) \Rightarrow a_2$
$\mid (a_3,\ \text{-}) \Rightarrow a_3 \mid (\text{-},\ a_3) \Rightarrow a_3$
$\mid \text{-} \Rightarrow a_4)$
definition
$x \sqcup y = (case\ (x,\ y)\ of$
$(a_4,\ \text{-}) \Rightarrow a_4 \mid (\text{-},\ a_4) \Rightarrow a_4 \mid (a_2,\ a_3) \Rightarrow a_4 \mid (a_3,\ a_2) \Rightarrow a_4$
$\mid (a_2,\ \text{-}) \Rightarrow a_2 \mid (\text{-},\ a_2) \Rightarrow a_2$
$\mid (a_3,\ \text{-}) \Rightarrow a_3 \mid (\text{-},\ a_3) \Rightarrow a_3$
$\mid \text{-} \Rightarrow a_1)$

instance
⟨*proof*⟩

end

instance *finite-4* :: *complete-distrib-lattice*
⟨*proof*⟩

instantiation *finite-4* :: *complete-boolean-algebra* **begin**
definition $-\ x = (case\ x\ of\ a_1 \Rightarrow a_4 \mid a_2 \Rightarrow a_3 \mid a_3 \Rightarrow a_2 \mid a_4 \Rightarrow a_1)$
definition $x - y = x \sqcap -\ (y :: \mathit{finite\text{-}4})$
instance
⟨*proof*⟩
end

hide-const (**open**) $a_1\ a_2\ a_3\ a_4$

datatype (*plugins only*: *code quickcheck extraction*) *finite-5* =
$a_1 \mid a_2 \mid a_3 \mid a_4 \mid a_5$

notation (**output**) $a_1\ \ (a_1)$
notation (**output**) $a_2\ \ (a_2)$
notation (**output**) $a_3\ \ (a_3)$
notation (**output**) $a_4\ \ (a_4)$
notation (**output**) $a_5\ \ (a_5)$

lemma *UNIV-finite-5*:
$UNIV = \{a_1,\ a_2,\ a_3,\ a_4,\ a_5\}$
⟨*proof*⟩

instantiation *finite-5* :: *enum*
begin

definition
$enum = [a_1,\ a_2,\ a_3,\ a_4,\ a_5]$

definition
  $enum\text{-}all\ P \longleftrightarrow P\ a_1 \wedge P\ a_2 \wedge P\ a_3 \wedge P\ a_4 \wedge P\ a_5$

definition
  $enum\text{-}ex\ P \longleftrightarrow P\ a_1 \vee P\ a_2 \vee P\ a_3 \vee P\ a_4 \vee P\ a_5$

instance $\langle proof \rangle$

end

instantiation *finite-5* :: *complete-lattice*
begin

The non-distributive pentagon lattice N_5

definition
  $x < y \longleftrightarrow (case\ (x,\ y)\ of$
    $(a_1,\ a_1) \Rightarrow False \mid (a_1,\ \text{-}) \Rightarrow True$
  $\mid (a_2,\ a_3) \Rightarrow True \mid (a_2,\ a_5) \Rightarrow True$
  $\mid (a_3,\ a_5) \Rightarrow True$
  $\mid (a_4,\ a_5) \Rightarrow True \mid \text{-} \Rightarrow False)$

definition
  $x \le y \longleftrightarrow (case\ (x,\ y)\ of$
    $(a_1,\ \text{-}) \Rightarrow True$
  $\mid (a_2,\ a_2) \Rightarrow True \mid (a_2,\ a_3) \Rightarrow True \mid (a_2,\ a_5) \Rightarrow True$
  $\mid (a_3,\ a_3) \Rightarrow True \mid (a_3,\ a_5) \Rightarrow True$
  $\mid (a_4,\ a_4) \Rightarrow True \mid (a_4,\ a_5) \Rightarrow True$
  $\mid (a_5,\ a_5) \Rightarrow True \mid \text{-} \Rightarrow False)$

definition
  $\bigsqcap A =$
  $(if\ a_1 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A)\ then\ a_1$
   $else\ if\ a_2 \in A\ then\ a_2$
   $else\ if\ a_3 \in A\ then\ a_3$
   $else\ if\ a_4 \in A\ then\ a_4$
   $else\ a_5)$
definition
  $\bigsqcup A =$
  $(if\ a_5 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A)\ then\ a_5$
   $else\ if\ a_3 \in A\ then\ a_3$
   $else\ if\ a_2 \in A\ then\ a_2$
   $else\ if\ a_4 \in A\ then\ a_4$
   $else\ a_1)$
definition [*simp*]: $bot = a_1$
definition [*simp*]: $top = a_5$
definition
  $x \sqcap y = (case\ (x,\ y)\ of$
    $(a_1,\ \text{-}) \Rightarrow a_1 \mid (\text{-},\ a_1) \Rightarrow a_1 \mid (a_2,\ a_4) \Rightarrow a_1 \mid (a_4,\ a_2) \Rightarrow a_1 \mid (a_3,\ a_4) \Rightarrow a_1 \mid$
$(a_4,\ a_3) \Rightarrow a_1$

| $(a_2, \text{-}) \Rightarrow a_2$ | $(\text{-}, a_2) \Rightarrow a_2$
| $(a_3, \text{-}) \Rightarrow a_3$ | $(\text{-}, a_3) \Rightarrow a_3$
| $(a_4, \text{-}) \Rightarrow a_4$ | $(\text{-}, a_4) \Rightarrow a_4$
| - $\Rightarrow a_5$)

definition
$x \sqcup y = (case\ (x, y)\ of$
$(a_5, \text{-}) \Rightarrow a_5$ | $(\text{-}, a_5) \Rightarrow a_5$ | $(a_2, a_4) \Rightarrow a_5$ | $(a_4, a_2) \Rightarrow a_5$ | $(a_3, a_4) \Rightarrow a_5$ | $(a_4, a_3) \Rightarrow a_5$
| $(a_3, \text{-}) \Rightarrow a_3$ | $(\text{-}, a_3) \Rightarrow a_3$
| $(a_2, \text{-}) \Rightarrow a_2$ | $(\text{-}, a_2) \Rightarrow a_2$
| $(a_4, \text{-}) \Rightarrow a_4$ | $(\text{-}, a_4) \Rightarrow a_4$
| - $\Rightarrow a_1$)

instance
⟨*proof*⟩

end

hide-const (**open**) a_1 a_2 a_3 a_4 a_5

71.5 Closing up

hide-type (**open**) *finite-1 finite-2 finite-3 finite-4 finite-5*
hide-const (**open**) *enum enum-all enum-ex all-n-lists ex-n-lists ntrancl*

end

72 Character and string types

theory *String*
imports *Enum*
begin

72.1 Characters and strings

72.1.1 Characters as finite algebraic type

typedef $char = \{n{::}nat.\ n < 256\}$
morphisms *nat-of-char Abs-char*
⟨*proof*⟩

setup-lifting *type-definition-char*

definition *char-of-nat* :: *nat* ⇒ *char*
where
char-of-nat n = *Abs-char* (*n mod 256*)

lemma *char-cases* [*case-names char-of-nat*, *cases type*: *char*]:
$(\bigwedge n.\ c = \textit{char-of-nat}\ n \Longrightarrow n < 256 \Longrightarrow P) \Longrightarrow P$

⟨*proof*⟩

lemma *char-of-nat-of-char* [*simp*]:
char-of-nat (*nat-of-char* *c*) = *c*
⟨*proof*⟩

lemma *inj-nat-of-char*:
inj nat-of-char
⟨*proof*⟩

lemma *nat-of-char-eq-iff* [*simp*]:
nat-of-char c = *nat-of-char d* ⟷ *c* = *d*
⟨*proof*⟩

lemma *nat-of-char-of-nat* [*simp*]:
nat-of-char (*char-of-nat n*) = *n mod 256*
⟨*proof*⟩

lemma *char-of-nat-mod-256* [*simp*]:
char-of-nat (*n mod 256*) = *char-of-nat n*
⟨*proof*⟩

lemma *char-of-nat-quasi-inj* [*simp*]:
char-of-nat m = *char-of-nat n* ⟷ *m mod 256* = *n mod 256*
⟨*proof*⟩

lemma *inj-on-char-of-nat* [*simp*]:
inj-on char-of-nat {..<*256*}
⟨*proof*⟩

lemma *nat-of-char-mod-256* [*simp*]:
nat-of-char c mod 256 = *nat-of-char c*
⟨*proof*⟩

lemma *nat-of-char-less-256* [*simp*]:
nat-of-char c < *256*
⟨*proof*⟩

lemma *UNIV-char-of-nat*:
UNIV = *char-of-nat* ‘ {..<*256*}
⟨*proof*⟩

lemma *card-UNIV-char*:
card (*UNIV* :: *char set*) = *256*
⟨*proof*⟩

lemma *range-nat-of-char*:
range nat-of-char = {..<*256*}
⟨*proof*⟩

72.1.2 Character literals as variant of numerals

instantiation *char* :: *zero*
begin

definition *zero-char* :: *char*
  where *0* = *char-of-nat 0*

instance ⟨*proof*⟩

end

definition *Char* :: *num* ⇒ *char*
  where *Char k* = *char-of-nat* (*numeral k*)

code-datatype *0* :: *char Char*

lemma *nat-of-char-zero* [*simp*]:
  nat-of-char 0 = *0*
  ⟨*proof*⟩

lemma *nat-of-char-Char* [*simp*]:
  nat-of-char (*Char k*) = *numeral k mod 256*
  ⟨*proof*⟩

lemma *Char-eq-Char-iff*:
  Char k = *Char l* ⟷ *numeral k mod* (*256* :: *nat*) = *numeral l mod 256* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *zero-eq-Char-iff*:
  0 = *Char k* ⟷ *numeral k mod* (*256* :: *nat*) = *0*
  ⟨*proof*⟩

lemma *Char-eq-zero-iff*:
  Char k = *0* ⟷ *numeral k mod* (*256* :: *nat*) = *0*
  ⟨*proof*⟩

⟨*ML*⟩

definition *integer-of-char* :: *char* ⇒ *integer*
where
  integer-of-char = *integer-of-nat* ∘ *nat-of-char*

definition *char-of-integer* :: *integer* ⇒ *char*
where
  char-of-integer = *char-of-nat* ∘ *nat-of-integer*

lemma *integer-of-char-zero* [*simp*, *code*]:
  integer-of-char 0 = *0*

⟨*proof*⟩

lemma *integer-of-char-Char* [*simp*]:
integer-of-char (*Char k*) = *numeral k mod 256*
⟨*proof*⟩

lemma *integer-of-char-Char-code* [*code*]:
integer-of-char (*Char k*) = *integer-of-num k mod 256*
⟨*proof*⟩

lemma *nat-of-char-code* [*code*]:
nat-of-char = *nat-of-integer* ∘ *integer-of-char*
⟨*proof*⟩

lemma *char-of-nat-code* [*code*]:
char-of-nat = *char-of-integer* ∘ *integer-of-nat*
⟨*proof*⟩

instantiation *char* :: *equal*
begin

definition *equal-char*
where *equal-char* (*c* :: *char*) *d* ⟷ *c* = *d*

instance
⟨*proof*⟩

end

lemma *equal-char-simps* [*code*]:
HOL.equal (*0*::*char*) *0* ⟷ *True*
HOL.equal (*Char k*) (*Char l*) ⟷ *HOL.equal* (*numeral k mod 256* :: *nat*) (*numeral l mod 256*)
HOL.equal 0 (*Char k*) ⟷ *HOL.equal* (*numeral k mod 256* :: *nat*) *0*
HOL.equal (*Char k*) *0* ⟷ *HOL.equal* (*numeral k mod 256* :: *nat*) *0*
⟨*proof*⟩

syntax
-Char :: *str-position* ⇒ *char* (*CHR* -)
-Char-ord :: *num-const* ⇒ *char* (*CHR* -)

type-synonym *string* = *char list*

syntax
-String :: *str-position* => *string* (-)

⟨*ML*⟩

instantiation *char* :: *enum*

begin

definition
Enum.enum = [0, CHR 0x01, CHR 0x02, CHR 0x03,
CHR 0x04, CHR 0x05, CHR 0x06, CHR 0x07,
CHR 0x08, CHR 0x09, CHR ''$\boxed{\hookleftarrow}$'', CHR 0x0B,
CHR 0x0C, CHR 0x0D, CHR 0x0E, CHR 0x0F,
CHR 0x10, CHR 0x11, CHR 0x12, CHR 0x13,
CHR 0x14, CHR 0x15, CHR 0x16, CHR 0x17,
CHR 0x18, CHR 0x19, CHR 0x1A, CHR 0x1B,
CHR 0x1C, CHR 0x1D, CHR 0x1E, CHR 0x1F,
CHR '' '', CHR ''!'', CHR 0x22, CHR ''#'',
CHR ''$'', CHR ''%'', CHR ''&'', CHR 0x27,
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 ''['',
CHR 0x5C, CHR '']'', CHR ''^'', CHR ''-'',
CHR 0x60, 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 ''~'', CHR 0x7F,
CHR 0x80, CHR 0x81, CHR 0x82, CHR 0x83,
CHR 0x84, CHR 0x85, CHR 0x86, CHR 0x87,
CHR 0x88, CHR 0x89, CHR 0x8A, CHR 0x8B,
CHR 0x8C, CHR 0x8D, CHR 0x8E, CHR 0x8F,
CHR 0x90, CHR 0x91, CHR 0x92, CHR 0x93,
CHR 0x94, CHR 0x95, CHR 0x96, CHR 0x97,
CHR 0x98, CHR 0x99, CHR 0x9A, CHR 0x9B,
CHR 0x9C, CHR 0x9D, CHR 0x9E, CHR 0x9F,
CHR 0xA0, CHR 0xA1, CHR 0xA2, CHR 0xA3,
CHR 0xA4, CHR 0xA5, CHR 0xA6, CHR 0xA7,
CHR 0xA8, CHR 0xA9, CHR 0xAA, CHR 0xAB,
CHR 0xAC, CHR 0xAD, CHR 0xAE, CHR 0xAF,
CHR 0xB0, CHR 0xB1, CHR 0xB2, CHR 0xB3,
CHR 0xB4, CHR 0xB5, CHR 0xB6, CHR 0xB7,

CHR 0xB8, *CHR 0xB9*, *CHR 0xBA*, *CHR 0xBB*,
CHR 0xBC, *CHR 0xBD*, *CHR 0xBE*, *CHR 0xBF*,
CHR 0xC0, *CHR 0xC1*, *CHR 0xC2*, *CHR 0xC3*,
CHR 0xC4, *CHR 0xC5*, *CHR 0xC6*, *CHR 0xC7*,
CHR 0xC8, *CHR 0xC9*, *CHR 0xCA*, *CHR 0xCB*,
CHR 0xCC, *CHR 0xCD*, *CHR 0xCE*, *CHR 0xCF*,
CHR 0xD0, *CHR 0xD1*, *CHR 0xD2*, *CHR 0xD3*,
CHR 0xD4, *CHR 0xD5*, *CHR 0xD6*, *CHR 0xD7*,
CHR 0xD8, *CHR 0xD9*, *CHR 0xDA*, *CHR 0xDB*,
CHR 0xDC, *CHR 0xDD*, *CHR 0xDE*, *CHR 0xDF*,
CHR 0xE0, *CHR 0xE1*, *CHR 0xE2*, *CHR 0xE3*,
CHR 0xE4, *CHR 0xE5*, *CHR 0xE6*, *CHR 0xE7*,
CHR 0xE8, *CHR 0xE9*, *CHR 0xEA*, *CHR 0xEB*,
CHR 0xEC, *CHR 0xED*, *CHR 0xEE*, *CHR 0xEF*,
CHR 0xF0, *CHR 0xF1*, *CHR 0xF2*, *CHR 0xF3*,
CHR 0xF4, *CHR 0xF5*, *CHR 0xF6*, *CHR 0xF7*,
CHR 0xF8, *CHR 0xF9*, *CHR 0xFA*, *CHR 0xFB*,
CHR 0xFC, *CHR 0xFD*, *CHR 0xFE*, *CHR 0xFF*]

definition
Enum.enum-all P ⟷ *list-all P* (*Enum.enum* :: *char list*)

definition
Enum.enum-ex P ⟷ *list-ex P* (*Enum.enum* :: *char list*)

lemma *enum-char-unfold*:
Enum.enum = *map char-of-nat* [*0..<256*]
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *char-of-integer-code* [*code*]:
char-of-integer n = *Enum.enum* ! (*nat-of-integer n mod 256*)
⟨*proof*⟩

lifting-update *char.lifting*
lifting-forget *char.lifting*

72.2 Strings as dedicated type

typedef *literal* = *UNIV* :: *string set*
morphisms *explode STR* ⟨*proof*⟩

setup-lifting *type-definition-literal*

lemma *STR-inject′* [*simp*]:
STR s = *STR t* ⟷ *s* = *t*

⟨*proof*⟩

definition *implode* :: *string* ⇒ *String.literal*
where
[*code del*]: *implode* = *STR*

instantiation *literal* :: *size*
begin

definition *size-literal* :: *literal* ⇒ *nat*
where
[*code*]: *size-literal* (*s*::*literal*) = *0*

instance ⟨*proof*⟩

end

instantiation *literal* :: *equal*
begin

lift-definition *equal-literal* :: *literal* ⇒ *literal* ⇒ *bool* **is** *op* = ⟨*proof*⟩

instance ⟨*proof*⟩

end

declare *equal-literal.rep-eq*[*code*]

lemma [*code nbe*]:
fixes *s* :: *String.literal*
shows *HOL.equal s s* ⟷ *True*
⟨*proof*⟩

lifting-update *literal.lifting*
lifting-forget *literal.lifting*

72.3 Dedicated conversion for generated computations

definition *char-of-num* :: *num* ⇒ *char*
where *char-of-num* = *char-of-nat o nat-of-num*

lemma [*code-computation-unfold*]:
Char = *char-of-num*
⟨*proof*⟩

72.4 Code generator

⟨*ML*⟩

code-reserved *SML string*

code-reserved *OCaml string*
code-reserved *Scala string*

code-printing
  type-constructor *literal* ⇀
    (*SML*) *string*
    and (*OCaml*) *string*
    and (*Haskell*) *String*
    and (*Scala*) *String*

⟨*ML*⟩

code-printing
  class-instance *literal* :: *equal* ⇀
    (*Haskell*) −
| **constant** *HOL.equal* :: *literal* ⇒ *literal* ⇒ *bool* ⇀
    (*SML*) !((- : *string*) = -)
    and (*OCaml*) !((- : *string*) = -)
    and (*Haskell*) **infix** *4* ==
    and (*Scala*) **infixl** *5* ==

⟨*ML*⟩

definition *abort* :: *literal* ⇒ (*unit* ⇒ *'a*) ⇒ *'a*
where [*simp*, *code del*]: *abort* - *f* = *f* ()

lemma *abort-cong*: *msg* = *msg'* ==> *Code.abort msg f* = *Code.abort msg' f*
⟨*proof*⟩

⟨*ML*⟩

code-printing constant *Code.abort* ⇀
    (*SML*) !(*raise*/ *Fail*/ -)
    and (*OCaml*) *failwith*
    and (*Haskell*) !(*error*/ ::/ *forall a.*/ *String* −> (() −> *a*) −> *a*)
    and (*Scala*) !{/ *sys.error*((-));/ ((-)).*apply*(())/ }

hide-type (**open**) *literal*

hide-const (**open**) *implode explode*

end

73 Reflecting Pure types into HOL

theory *Typerep*
imports *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*]:
 HOL.equal (*Typerep tyco1 tys1*) (*Typerep tyco2 tys2*) ⟷ *HOL.equal tyco1 tyco2*
 ∧ *list-all2 HOL.equal tys1 tys2*
 ⟨*proof*⟩

lemma [*code nbe*]:
 HOL.equal (*x* :: *typerep*) *x* ⟷ *True*
 ⟨*proof*⟩

code-printing
 type-constructor *typerep* ⇀ (*Eval*) *Term.typ*
| **constant** *Typerep* ⇀ (*Eval*) *Term.Type/* (-, -)

code-reserved *Eval Term*

hide-const (**open**) *typerep Typerep*

end

74 Predicates as enumerations

theory *Predicate*
imports *String*
begin

74.1 The type of predicate enumerations (a monad)

datatype (*plugins only*: *extraction*) (*dead 'a*) *pred* = *Pred* (*eval*: *'a* ⇒ *bool*)

lemma *pred-eqI*:
 $(\bigwedge w.\ eval\ P\ w \longleftrightarrow eval\ Q\ w) \Longrightarrow P = Q$
 ⟨*proof*⟩

lemma *pred-eq-iff*:
$P = Q \Longrightarrow (\bigwedge w.\ eval\ P\ w \longleftrightarrow eval\ Q\ w)$
$\langle proof \rangle$

instantiation *pred* :: (*type*) *complete-lattice*
begin

definition
$P \leq Q \longleftrightarrow eval\ P \leq eval\ Q$

definition
$P < Q \longleftrightarrow eval\ P < eval\ Q$

definition
$\bot = Pred\ \bot$

lemma *eval-bot* [*simp*]:
$eval\ \bot\ = \bot$
$\langle proof \rangle$

definition
$\top = Pred\ \top$

lemma *eval-top* [*simp*]:
$eval\ \top\ = \top$
$\langle proof \rangle$

definition
$P \sqcap Q = Pred\ (eval\ P \sqcap eval\ Q)$

lemma *eval-inf* [*simp*]:
$eval\ (P \sqcap Q) = eval\ P \sqcap eval\ Q$
$\langle proof \rangle$

definition
$P \sqcup Q = Pred\ (eval\ P \sqcup eval\ Q)$

lemma *eval-sup* [*simp*]:
$eval\ (P \sqcup Q) = eval\ P \sqcup eval\ Q$
$\langle proof \rangle$

definition
$\bigsqcap A = Pred\ (INFIMUM\ A\ eval)$

lemma *eval-Inf* [*simp*]:
$eval\ (\bigsqcap A) = INFIMUM\ A\ eval$
$\langle proof \rangle$

definition

$\bigsqcup A = Pred\ (SUPREMUM\ A\ eval)$

lemma *eval-Sup* [*simp*]:
$eval\ (\bigsqcup A) = SUPREMUM\ A\ eval$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *eval-INF* [*simp*]:
$eval\ (INFIMUM\ A\ f) = INFIMUM\ A\ (eval \circ f)$
⟨*proof*⟩

lemma *eval-SUP* [*simp*]:
$eval\ (SUPREMUM\ A\ f) = SUPREMUM\ A\ (eval \circ f)$
⟨*proof*⟩

instantiation *pred* :: (*type*) *complete-boolean-algebra*
begin

definition
$-\ P = Pred\ (-\ eval\ P)$

lemma *eval-compl* [*simp*]:
$eval\ (-\ P) = -\ eval\ P$
⟨*proof*⟩

definition
$P - Q = Pred\ (eval\ P - eval\ Q)$

lemma *eval-minus* [*simp*]:
$eval\ (P - Q) = eval\ P - eval\ Q$
⟨*proof*⟩

instance ⟨*proof*⟩

end

definition *single* :: $'a \Rightarrow 'a\ pred$ **where**
$single\ x = Pred\ ((op\ =)\ x)$

lemma *eval-single* [*simp*]:
$eval\ (single\ x) = (op\ =)\ x$
⟨*proof*⟩

definition *bind* :: $'a\ pred \Rightarrow ('a \Rightarrow 'b\ pred) \Rightarrow 'b\ pred$ (**infixl** $\gg=$ *70*) **where**
$P \gg= f = (SUPREMUM\ \{x.\ eval\ P\ x\}\ f)$

lemma *eval-bind* [*simp*]:
$eval\ (P \gg= f) = eval\ (SUPREMUM\ \{x.\ eval\ P\ x\}\ f)$
⟨*proof*⟩

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*: $eval\ (single\ x)\ y \Longrightarrow (y = x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *singleE'*: $eval\ (single\ x)\ y \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *bindI*: $eval\ P\ x \Longrightarrow eval\ (Q\ x)\ y \Longrightarrow eval\ (P \gg= Q)\ y$
⟨*proof*⟩

lemma *bindE*: $eval\ (R \gg= Q)\ y \Longrightarrow (\bigwedge x.\ eval\ R\ x \Longrightarrow eval\ (Q\ x)\ y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *botE*: $eval\ \bot\ x \Longrightarrow P$
⟨*proof*⟩

lemma *supI1*: $eval\ A\ x \Longrightarrow eval\ (A \sqcup B)\ x$
⟨*proof*⟩

lemma *supI2*: $eval\ B\ x \Longrightarrow eval\ (A \sqcup B)\ x$
⟨*proof*⟩

lemma *supE*: $eval\ (A \sqcup B)\ x \Longrightarrow (eval\ A\ x \Longrightarrow P) \Longrightarrow (eval\ B\ x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *single-not-bot* [*simp*]:
$single\ x \neq \bot$
⟨*proof*⟩

lemma *not-bot*:
assumes $A \neq \bot$
obtains x **where** $eval\ A\ x$
⟨*proof*⟩

74.2 Emptiness check and definite choice

definition *is-empty* :: $'a\ pred \Rightarrow bool$ **where**
$is\text{-}empty\ A \longleftrightarrow A = \bot$

lemma *is-empty-bot*:
$is\text{-}empty\ \bot$
⟨*proof*⟩

lemma *not-is-empty-single*:
$\neg\ is\text{-}empty\ (single\ x)$
⟨*proof*⟩

lemma *is-empty-sup*:
$is\text{-}empty\ (A \sqcup B) \longleftrightarrow is\text{-}empty\ A \wedge is\text{-}empty\ B$
⟨*proof*⟩

definition *singleton* :: $(unit \Rightarrow 'a) \Rightarrow 'a\ pred \Rightarrow 'a$ **where**
$singleton\ default\ A = (if\ \exists!x.\ eval\ A\ x\ then\ THE\ x.\ eval\ A\ x\ else\ default\ ())$ **for** *default*

lemma *singleton-eqI*:
$\exists!x.\ eval\ A\ x \Longrightarrow eval\ A\ x \Longrightarrow singleton\ default\ A = x$ **for** *default*
⟨*proof*⟩

lemma *eval-singletonI*:
$\exists!x.\ eval\ A\ x \Longrightarrow eval\ A\ (singleton\ default\ A)$ **for** *default*
⟨*proof*⟩

lemma *single-singleton*:
$\exists!x.\ eval\ A\ x \Longrightarrow single\ (singleton\ default\ A) = A$ **for** *default*
⟨*proof*⟩

lemma *singleton-undefinedI*:
$\neg\ (\exists!x.\ eval\ A\ x) \Longrightarrow singleton\ default\ A = default\ ()$ **for** *default*
⟨*proof*⟩

lemma *singleton-bot*:
$singleton\ default\ \bot = default\ ()$ **for** *default*
⟨*proof*⟩

lemma *singleton-single*:
$singleton\ default\ (single\ x) = x$ **for** *default*
⟨*proof*⟩

lemma *singleton-sup-single-single*:
$singleton\ default\ (single\ x \sqcup single\ y) = (if\ x = y\ then\ x\ else\ default\ ())$ **for** *default*
⟨*proof*⟩

lemma *singleton-sup-aux*:
$singleton\ default\ (A \sqcup B) = (if\ A = \bot\ then\ singleton\ default\ B$
$else\ if\ B = \bot\ then\ singleton\ default\ A$
$else\ singleton\ default$
$(single\ (singleton\ default\ A) \sqcup single\ (singleton\ default\ B)))$ **for** *default*
⟨*proof*⟩

lemma *singleton-sup*:
$singleton\ default\ (A \sqcup B) = (if\ A = \bot\ then\ singleton\ default\ B$
$else\ if\ B = \bot\ then\ singleton\ default\ A$
$else\ if\ singleton\ default\ A = singleton\ default\ B\ then\ singleton\ default\ A\ else\ default\ ())$ **for** *default*
⟨*proof*⟩

74.3 Derived operations

definition *if-pred* :: $bool \Rightarrow unit\ pred$ **where**
if-pred-eq: $if\text{-}pred\ b = (if\ b\ then\ single\ ()\ else\ \bot)$

definition *holds* :: $unit\ pred \Rightarrow bool$ **where**
holds-eq: $holds\ P = eval\ P\ ()$

definition *not-pred* :: $unit\ pred \Rightarrow unit\ pred$ **where**

not-pred-eq: $not\text{-}pred\ P = (if\ eval\ P\ ()\ then\ \bot\ else\ single\ ())$

lemma *if-predI*: $P \Longrightarrow eval\ (if\text{-}pred\ P)\ ()$
⟨*proof*⟩

lemma *if-predE*: $eval\ (if\text{-}pred\ b)\ x \Longrightarrow (b \Longrightarrow x = () \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *not-predI*: $\neg\ P \Longrightarrow eval\ (not\text{-}pred\ (Pred\ (\lambda u.\ P)))\ ()$
⟨*proof*⟩

lemma *not-predI'*: $\neg\ eval\ P\ () \Longrightarrow eval\ (not\text{-}pred\ P)\ ()$
⟨*proof*⟩

lemma *not-predE*: $eval\ (not\text{-}pred\ (Pred\ (\lambda u.\ P)))\ x \Longrightarrow (\neg\ P \Longrightarrow thesis) \Longrightarrow thesis$
⟨*proof*⟩

lemma *not-predE'*: $eval\ (not\text{-}pred\ P)\ x \Longrightarrow (\neg\ eval\ P\ x \Longrightarrow thesis) \Longrightarrow thesis$
⟨*proof*⟩
lemma $f\ () = False \lor f\ () = True$
⟨*proof*⟩

lemma *closure-of-bool-cases* [*no-atp*]:
fixes $f :: unit \Rightarrow bool$
assumes $f = (\lambda u.\ False) \Longrightarrow P\ f$
assumes $f = (\lambda u.\ True) \Longrightarrow P\ f$
shows $P\ f$
⟨*proof*⟩

lemma *unit-pred-cases*:
assumes $P\ \bot$
assumes $P\ (single\ ())$
shows $P\ Q$
⟨*proof*⟩

lemma *holds-if-pred*:
$holds\ (if\text{-}pred\ b) = b$
⟨*proof*⟩

lemma *if-pred-holds*:
$if\text{-}pred\ (holds\ P) = P$
⟨*proof*⟩

lemma *is-empty-holds*:
$is\text{-}empty\ P \longleftrightarrow \neg\ holds\ P$
⟨*proof*⟩

definition $map :: ('a \Rightarrow 'b) \Rightarrow 'a\ pred \Rightarrow 'b\ pred$ **where**

$map\ f\ P = P \gg= (single\ o\ f)$

lemma *eval-map* [*simp*]:
$eval\ (map\ f\ P) = (\bigsqcup x \in \{x.\ eval\ P\ x\}.\ (\lambda y.\ f\ x = y))$
⟨*proof*⟩

functor *map*: *map*
⟨*proof*⟩

74.4 Implementation

datatype (*plugins only*: *code extraction*) (*dead* 'a) *seq* =
Empty
| *Insert* 'a 'a *pred*
| *Join* 'a *pred* 'a *seq*

primrec *pred-of-seq* :: 'a *seq* ⇒ 'a *pred* **where**
$pred\text{-}of\text{-}seq\ Empty = \bot$
| $pred\text{-}of\text{-}seq\ (Insert\ x\ P) = single\ x \sqcup P$
| $pred\text{-}of\text{-}seq\ (Join\ P\ xq) = P \sqcup pred\text{-}of\text{-}seq\ xq$

definition *Seq* :: (*unit* ⇒ 'a *seq*) ⇒ 'a *pred* **where**
$Seq\ f = pred\text{-}of\text{-}seq\ (f\ ())$

code-datatype *Seq*

primrec *member* :: 'a *seq* ⇒ 'a ⇒ *bool* **where**
$member\ Empty\ x \longleftrightarrow False$
| $member\ (Insert\ y\ P)\ x \longleftrightarrow x = y \lor eval\ P\ x$
| $member\ (Join\ P\ xq)\ x \longleftrightarrow eval\ P\ x \lor member\ xq\ x$

lemma *eval-member*:
$member\ xq = eval\ (pred\text{-}of\text{-}seq\ xq)$
⟨*proof*⟩

lemma *eval-code* [*code*]: $eval\ (Seq\ f) = member\ (f\ ())$
⟨*proof*⟩

lemma *single-code* [*code*]:
$single\ x = Seq\ (\lambda u.\ Insert\ x\ \bot)$
⟨*proof*⟩

primrec *apply* :: ('a ⇒ 'b *pred*) ⇒ 'a *seq* ⇒ 'b *seq* **where**
$apply\ f\ Empty = Empty$
| $apply\ f\ (Insert\ x\ P) = Join\ (f\ x)\ (Join\ (P \gg= f)\ Empty)$
| $apply\ f\ (Join\ P\ xq) = Join\ (P \gg= f)\ (apply\ f\ xq)$

lemma *apply-bind*:
$pred\text{-}of\text{-}seq\ (apply\ f\ xq) = pred\text{-}of\text{-}seq\ xq \gg= f$

⟨*proof*⟩

lemma *bind-code* [*code*]:
  $Seq\ g \gg= f = Seq\ (\lambda u.\ apply\ f\ (g\ ()))$
  ⟨*proof*⟩

lemma *bot-set-code* [*code*]:
  $\bot = Seq\ (\lambda u.\ Empty)$
  ⟨*proof*⟩

primrec *adjunct* :: $'a\ pred \Rightarrow\ 'a\ seq \Rightarrow\ 'a\ seq$ **where**
  $adjunct\ P\ Empty = Join\ P\ Empty$
| $adjunct\ P\ (Insert\ x\ Q) = Insert\ x\ (Q \sqcup P)$
| $adjunct\ P\ (Join\ Q\ xq) = Join\ Q\ (adjunct\ P\ xq)$

lemma *adjunct-sup*:
  $pred\text{-}of\text{-}seq\ (adjunct\ P\ xq) = P \sqcup pred\text{-}of\text{-}seq\ xq$
  ⟨*proof*⟩

lemma *sup-code* [*code*]:
  $Seq\ f \sqcup Seq\ g = Seq\ (\lambda u.\ case\ f\ ()$
    $of\ Empty \Rightarrow g\ ()$
     $|\ Insert\ x\ P \Rightarrow Insert\ x\ (P \sqcup Seq\ g)$
     $|\ Join\ P\ xq \Rightarrow adjunct\ (Seq\ g)\ (Join\ P\ xq))$
⟨*proof*⟩

primrec *contained* :: $'a\ seq \Rightarrow\ 'a\ pred \Rightarrow bool$ **where**
  $contained\ Empty\ Q \longleftrightarrow True$
| $contained\ (Insert\ x\ P)\ Q \longleftrightarrow eval\ Q\ x \wedge P \le Q$
| $contained\ (Join\ P\ xq)\ Q \longleftrightarrow P \le Q \wedge contained\ xq\ Q$

lemma *single-less-eq-eval*:
  $single\ x \le P \longleftrightarrow eval\ P\ x$
  ⟨*proof*⟩

lemma *contained-less-eq*:
  $contained\ xq\ Q \longleftrightarrow pred\text{-}of\text{-}seq\ xq \le Q$
  ⟨*proof*⟩

lemma *less-eq-pred-code* [*code*]:
  $Seq\ f \le Q = (case\ f\ ()$
   $of\ Empty \Rightarrow True$
    $|\ Insert\ x\ P \Rightarrow eval\ Q\ x \wedge P \le Q$
    $|\ Join\ P\ xq \Rightarrow P \le Q \wedge contained\ xq\ Q)$
  ⟨*proof*⟩

instantiation *pred* :: (*type*) *equal*
begin

definition *equal-pred*
where [*simp*]: $HOL.equal\ P\ Q \longleftrightarrow P = (Q :: 'a\ pred)$

instance ⟨*proof*⟩

end

lemma [*code*]:
$HOL.equal\ P\ Q \longleftrightarrow P \le Q \wedge Q \le P$ **for** $P\ Q :: 'a\ pred$
⟨*proof*⟩

lemma [*code nbe*]:
$HOL.equal\ P\ P \longleftrightarrow True$ **for** $P :: 'a\ pred$
⟨*proof*⟩

lemma [*code*]:
$case\text{-}pred\ f\ P = f\ (eval\ P)$
⟨*proof*⟩

lemma [*code*]:
$rec\text{-}pred\ f\ P = f\ (eval\ P)$
⟨*proof*⟩

inductive $eq :: 'a \Rightarrow 'a \Rightarrow bool$ **where** $eq\ x\ x$

lemma *eq-is-eq*: $eq\ x\ y \equiv (x = y)$
⟨*proof*⟩

primrec $null :: 'a\ seq \Rightarrow bool$ **where**
$null\ Empty \longleftrightarrow True$
$|\ null\ (Insert\ x\ P) \longleftrightarrow False$
$|\ null\ (Join\ P\ xq) \longleftrightarrow is\text{-}empty\ P \wedge null\ xq$

lemma *null-is-empty*:
$null\ xq \longleftrightarrow is\text{-}empty\ (pred\text{-}of\text{-}seq\ xq)$
⟨*proof*⟩

lemma *is-empty-code* [*code*]:
$is\text{-}empty\ (Seq\ f) \longleftrightarrow null\ (f\ ())$
⟨*proof*⟩

primrec $the\text{-}only :: (unit \Rightarrow 'a) \Rightarrow 'a\ seq \Rightarrow 'a$ **where**
$the\text{-}only\ default\ Empty = default\ ()$ **for** *default*
$|\ the\text{-}only\ default\ (Insert\ x\ P) =$
$(if\ is\text{-}empty\ P\ then\ x\ else\ let\ y = singleton\ default\ P\ in\ if\ x = y\ then\ x\ else\ default\ ())$ **for** *default*
$|\ the\text{-}only\ default\ (Join\ P\ xq) =$
$(if\ is\text{-}empty\ P\ then\ the\text{-}only\ default\ xq\ else\ if\ null\ xq\ then\ singleton\ default\ P$
$else\ let\ x = singleton\ default\ P;\ y = the\text{-}only\ default\ xq\ in$

if x = y then x else default ()) **for** *default*

lemma *the-only-singleton*:
the-only default xq = singleton default (*pred-of-seq xq*) **for** *default*
⟨*proof*⟩

lemma *singleton-code* [*code*]:
singleton default (*Seq f*) =
(*case f* () *of*
Empty ⇒ *default* ()
| *Insert x P* ⇒ *if is-empty P then x*
else let y = singleton default P in
if x = y then x else default ()
| *Join P xq* ⇒ *if is-empty P then the-only default xq*
else if null xq then singleton default P
else let x = singleton default P; *y = the-only default xq in*
if x = y then x else default ()) **for** *default*
⟨*proof*⟩

definition *the* :: *'a pred* ⇒ *'a* **where**
the A = (*THE x. eval A x*)

lemma *the-eqI*:
(*THE x. eval P x*) = *x* ⟹ *the P* = *x*
⟨*proof*⟩

lemma *the-eq* [*code*]: *the A* = *singleton* (λ*x. Code.abort* (*STR ''not-unique''*) (λ-. *the A*)) *A*
⟨*proof*⟩

code-reflect *Predicate*
datatypes *pred* = *Seq* **and** *seq* = *Empty* | *Insert* | *Join*

⟨*ML*⟩

Conversion from and to sets

definition *pred-of-set* :: *'a set* ⇒ *'a pred* **where**
pred-of-set = *Pred* ∘ (λ*A x. x* ∈ *A*)

lemma *eval-pred-of-set* [*simp*]:
eval (*pred-of-set A*) *x* ⟷ *x* ∈*A*
⟨*proof*⟩

definition *set-of-pred* :: *'a pred* ⇒ *'a set* **where**
set-of-pred = *Collect* ∘ *eval*

lemma *member-set-of-pred* [*simp*]:
x ∈ *set-of-pred P* ⟷ *Predicate.eval P x*
⟨*proof*⟩

definition *set-of-seq* :: $'a$ *seq* $\Rightarrow$ $'a$ *set* **where**
set-of-seq = *set-of-pred* $\circ$ *pred-of-seq*

lemma *member-set-of-seq* [*simp*]:
$x \in$ *set-of-seq xq* = *Predicate.member xq x*
⟨*proof*⟩

lemma *of-pred-code* [*code*]:
set-of-pred (*Predicate.Seq f*) = (*case f* () *of*
  Predicate.Empty $\Rightarrow$ {}
| *Predicate.Insert x P* $\Rightarrow$ *insert x* (*set-of-pred P*)
| *Predicate.Join P xq* $\Rightarrow$ *set-of-pred P* $\cup$ *set-of-seq xq*)
⟨*proof*⟩

lemma *of-seq-code* [*code*]:
set-of-seq Predicate.Empty = {}
set-of-seq (*Predicate.Insert x P*) = *insert x* (*set-of-pred P*)
set-of-seq (*Predicate.Join P xq*) = *set-of-pred P* $\cup$ *set-of-seq xq*
⟨*proof*⟩

Lazy Evaluation of an indexed function

function *iterate-upto* :: (*natural* $\Rightarrow$ $'a$) $\Rightarrow$ *natural* $\Rightarrow$ *natural* $\Rightarrow$ $'a$ *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*⟩

Misc

declare *Inf-set-fold* [**where** $'a$ = $'a$ *Predicate.pred*, *code*]
declare *Sup-set-fold* [**where** $'a$ = $'a$ *Predicate.pred*, *code*]

lemma *pred-of-set-fold-sup*:
assumes *finite A*
shows *pred-of-set A* = *Finite-Set.fold sup bot* (*Predicate.single* ‘ *A*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *pred-of-set-set-fold-sup*:
pred-of-set (*set xs*) = *fold sup* (*List.map Predicate.single xs*) *bot*
⟨*proof*⟩

lemma *pred-of-set-set-foldr-sup* [*code*]:
pred-of-set (*set xs*) = *foldr sup* (*List.map Predicate.single xs*) *bot*

⟨*proof*⟩

no-notation
bind (**infixl** $\gg=$ *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 the iterate-upto*
hide-fact (**open**) *null-def member-def*

end

75 Lazy sequences

theory *Lazy-Sequence*
imports *Predicate*
begin

75.1 Type of lazy sequences

datatype (*plugins only*: *code extraction*) (*dead 'a*) *lazy-sequence* =
lazy-sequence-of-list 'a list

primrec *list-of-lazy-sequence* :: *'a lazy-sequence* ⇒ *'a list*
where
list-of-lazy-sequence (*lazy-sequence-of-list xs*) = *xs*

lemma *lazy-sequence-of-list-of-lazy-sequence* [*simp*]:
lazy-sequence-of-list (*list-of-lazy-sequence xq*) = *xq*
⟨*proof*⟩

lemma *lazy-sequence-eqI*:
list-of-lazy-sequence xq = *list-of-lazy-sequence yq* ⟹ *xq* = *yq*
⟨*proof*⟩

lemma *lazy-sequence-eq-iff*:
xq = *yq* ⟷ *list-of-lazy-sequence xq* = *list-of-lazy-sequence yq*
⟨*proof*⟩

lemma *case-lazy-sequence* [*simp*]:
case-lazy-sequence f xq = *f* (*list-of-lazy-sequence xq*)
⟨*proof*⟩

lemma *rec-lazy-sequence* [*simp*]:
rec-lazy-sequence f xq = *f* (*list-of-lazy-sequence xq*)
⟨*proof*⟩

definition *Lazy-Sequence* :: (*unit* ⇒ (*'a* × *'a lazy-sequence*) *option*) ⇒ *'a lazy-sequence*

where
Lazy-Sequence f = *lazy-sequence-of-list* (*case f* () *of*
None ⇒ []
| *Some* (*x*, *xq*) ⇒ *x* # *list-of-lazy-sequence xq*)

code-datatype *Lazy-Sequence*

declare *list-of-lazy-sequence.simps* [*code del*]
declare *lazy-sequence.case* [*code del*]
declare *lazy-sequence.rec* [*code del*]

lemma *list-of-Lazy-Sequence* [*simp*]:
list-of-lazy-sequence (*Lazy-Sequence f*) = (*case f* () *of*
None ⇒ []
| *Some* (*x*, *xq*) ⇒ *x* # *list-of-lazy-sequence xq*)
⟨*proof*⟩

definition *yield* :: *'a lazy-sequence* ⇒ (*'a* × *'a lazy-sequence*) *option*
where
yield xq = (*case list-of-lazy-sequence xq of*
[] ⇒ *None*
| *x* # *xs* ⇒ *Some* (*x*, *lazy-sequence-of-list xs*))

lemma *yield-Seq* [*simp*, *code*]:
yield (*Lazy-Sequence f*) = *f* ()
⟨*proof*⟩

lemma *case-yield-eq* [*simp*]: *case-option g h* (*yield xq*) =
case-list g (λ*x*. *curry h x* ∘ *lazy-sequence-of-list*) (*list-of-lazy-sequence xq*)
⟨*proof*⟩

lemma *equal-lazy-sequence-code* [*code*]:
HOL.equal xq yq = (*case* (*yield xq*, *yield yq*) *of*
(*None*, *None*) ⇒ *True*
| (*Some* (*x*, *xq'*), *Some* (*y*, *yq'*)) ⇒ *HOL.equal x y* ∧ *HOL.equal xq yq*
| - ⇒ *False*)
⟨*proof*⟩

lemma [*code nbe*]:
HOL.equal (*x* :: *'a lazy-sequence*) *x* ⟷ *True*
⟨*proof*⟩

definition *empty* :: *'a lazy-sequence*
where
empty = *lazy-sequence-of-list* []

lemma *list-of-lazy-sequence-empty* [*simp*]:
list-of-lazy-sequence empty = []
⟨*proof*⟩

lemma *empty-code* [*code*]:
  empty = *Lazy-Sequence* (λ-. *None*)
  ⟨*proof*⟩

definition *single* :: $'a \Rightarrow 'a$ *lazy-sequence*
where
  single x = *lazy-sequence-of-list* [*x*]

lemma *list-of-lazy-sequence-single* [*simp*]:
  list-of-lazy-sequence (*single x*) = [*x*]
  ⟨*proof*⟩

lemma *single-code* [*code*]:
  single x = *Lazy-Sequence* (λ-. *Some* (*x*, *empty*))
  ⟨*proof*⟩

definition *append* :: $'a$ *lazy-sequence* $\Rightarrow$ $'a$ *lazy-sequence* $\Rightarrow$ $'a$ *lazy-sequence*
where
  append xq yq = *lazy-sequence-of-list* (*list-of-lazy-sequence xq* @ *list-of-lazy-sequence yq*)

lemma *list-of-lazy-sequence-append* [*simp*]:
  list-of-lazy-sequence (*append xq yq*) = *list-of-lazy-sequence xq* @ *list-of-lazy-sequence yq*
  ⟨*proof*⟩

lemma *append-code* [*code*]:
  append xq yq = *Lazy-Sequence* (λ-. *case yield xq of*
    None $\Rightarrow$ *yield yq*
  | *Some* (*x*, *xq'*) $\Rightarrow$ *Some* (*x*, *append xq' yq*))
  ⟨*proof*⟩

definition *map* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *lazy-sequence* $\Rightarrow$ $'b$ *lazy-sequence*
where
  map f xq = *lazy-sequence-of-list* (*List.map f* (*list-of-lazy-sequence xq*))

lemma *list-of-lazy-sequence-map* [*simp*]:
  list-of-lazy-sequence (*map f xq*) = *List.map f* (*list-of-lazy-sequence xq*)
  ⟨*proof*⟩

lemma *map-code* [*code*]:
  map f xq =
    Lazy-Sequence (λ-. *map-option* (λ(*x*, *xq'*). (*f x*, *map f xq'*)) (*yield xq*))
  ⟨*proof*⟩

definition *flat* :: $'a$ *lazy-sequence lazy-sequence* $\Rightarrow$ $'a$ *lazy-sequence*
where
  flat xqq = *lazy-sequence-of-list* (*concat* (*List.map list-of-lazy-sequence* (*list-of-lazy-sequence*

xqq)))

lemma *list-of-lazy-sequence-flat* [*simp*]:
list-of-lazy-sequence (*flat xqq*) = *concat* (*List.map list-of-lazy-sequence* (*list-of-lazy-sequence xqq*))
⟨*proof*⟩

lemma *flat-code* [*code*]:
flat xqq = *Lazy-Sequence* (λ-. *case yield xqq of*
None ⇒ *None*
| *Some* (*xq*, *xqq′*) ⇒ *yield* (*append xq* (*flat xqq′*)))
⟨*proof*⟩

definition *bind* :: ′*a lazy-sequence* ⇒ (′*a* ⇒ ′*b lazy-sequence*) ⇒ ′*b lazy-sequence*
where
bind xq f = *flat* (*map f xq*)

definition *if-seq* :: *bool* ⇒ *unit lazy-sequence*
where
if-seq b = (*if b then single* () *else empty*)

definition *those* :: ′*a option lazy-sequence* ⇒ ′*a lazy-sequence option*
where
those xq = *map-option lazy-sequence-of-list* (*List.those* (*list-of-lazy-sequence xq*))

function *iterate-upto* :: (*natural* ⇒ ′*a*) ⇒ *natural* ⇒ *natural* ⇒ ′*a lazy-sequence*
where
iterate-upto f n m =
Lazy-Sequence (λ-. *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*)

75.2 Code setup

code-reflect *Lazy-Sequence*
datatypes *lazy-sequence* = *Lazy-Sequence*

⟨*ML*⟩

75.3 Generator Sequences

75.3.1 General lazy sequence operation

definition *product* :: *'a lazy-sequence* ⇒ *'b lazy-sequence* ⇒ (*'a* × *'b*) *lazy-sequence*
where
product s1 s2 = *bind s1* (λa. *bind s2* (λb. *single* (*a*, *b*)))

75.3.2 Small lazy typeclasses

class *small-lazy* =
fixes *small-lazy* :: *natural* ⇒ *'a lazy-sequence*

instantiation *unit* :: *small-lazy*
begin

definition *small-lazy d* = *single* ()

instance ⟨*proof*⟩

end

instantiation *int* :: *small-lazy*
begin

maybe optimise this expression -¿ append (single x) xs == cons x xs Performance difference?

function *small-lazy'* :: *int* ⇒ *int* ⇒ *int lazy-sequence*
where
small-lazy' d i = (*if d* < *i then empty*
else append (*single i*) (*small-lazy' d* (*i* + *1*)))
⟨*proof*⟩

termination
⟨*proof*⟩

definition
small-lazy d = *small-lazy'* (*int* (*nat-of-natural d*)) (− (*int* (*nat-of-natural d*)))

instance ⟨*proof*⟩

end

instantiation *prod* :: (*small-lazy*, *small-lazy*) *small-lazy*
begin

definition
small-lazy d = *product* (*small-lazy d*) (*small-lazy d*)

instance ⟨*proof*⟩

end

instantiation *list* :: (*small-lazy*) *small-lazy*
begin

fun *small-lazy-list* :: *natural* ⇒ *'a list lazy-sequence*
where
small-lazy-list d = *append* (*single* [])
(*if d* > *0 then bind* (*product* (*small-lazy* (*d* − *1*))
(*small-lazy* (*d* − *1*))) (λ(*x*, *xs*). *single* (*x* # *xs*)) *else empty*)

instance ⟨*proof*⟩

end

75.4 With Hit Bound Value

assuming in negative context

type-synonym *'a hit-bound-lazy-sequence* = *'a option lazy-sequence*

definition *hit-bound* :: *'a hit-bound-lazy-sequence*
where
hit-bound = *Lazy-Sequence* (λ-. *Some* (*None*, *empty*))

lemma *list-of-lazy-sequence-hit-bound* [*simp*]:
list-of-lazy-sequence hit-bound = [*None*]
⟨*proof*⟩

definition *hb-single* :: *'a* ⇒ *'a hit-bound-lazy-sequence*
where
hb-single x = *Lazy-Sequence* (λ-. *Some* (*Some x*, *empty*))

definition *hb-map* :: (*'a* ⇒ *'b*) ⇒ *'a hit-bound-lazy-sequence* ⇒ *'b hit-bound-lazy-sequence*
where
hb-map f xq = *map* (*map-option f*) *xq*

lemma *hb-map-code* [*code*]:
hb-map f xq =
Lazy-Sequence (λ-. *map-option* (λ(*x*, *xq'*). (*map-option f x*, *hb-map f xq'*)) (*yield*
xq))
⟨*proof*⟩

definition *hb-flat* :: *'a hit-bound-lazy-sequence hit-bound-lazy-sequence* ⇒ *'a hit-bound-lazy-sequence*
where
hb-flat xqq = *lazy-sequence-of-list* (*concat*
(*List.map* ((λ*x*. *case x of None* ⇒ [*None*] | *Some xs* ⇒ *xs*) ∘ *map-option*
list-of-lazy-sequence) (*list-of-lazy-sequence xqq*)))

lemma *list-of-lazy-sequence-hb-flat* [*simp*]:
list-of-lazy-sequence (*hb-flat xqq*) =
concat (*List.map* ((λx. *case x of None* $\Rightarrow$ [*None*] | *Some xs* $\Rightarrow$ *xs*) $\circ$ *map-option list-of-lazy-sequence*) (*list-of-lazy-sequence xqq*))
⟨*proof*⟩

lemma *hb-flat-code* [*code*]:
hb-flat xqq = *Lazy-Sequence* (λ-. *case yield xqq of*
None $\Rightarrow$ *None*
| *Some* (*xq*, *xqq'*) $\Rightarrow$ *yield*
(*append* (*case xq of None* $\Rightarrow$ *hit-bound* | *Some xq* $\Rightarrow$ *xq*) (*hb-flat xqq'*)))
⟨*proof*⟩

definition *hb-bind* :: *'a hit-bound-lazy-sequence* $\Rightarrow$ (*'a* $\Rightarrow$ *'b hit-bound-lazy-sequence*) $\Rightarrow$ *'b hit-bound-lazy-sequence*
where
hb-bind xq f = *hb-flat* (*hb-map f xq*)

definition *hb-if-seq* :: *bool* $\Rightarrow$ *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* $\Rightarrow$ *unit lazy-sequence*
where
hb-not-seq xq = (*case yield xq of*
None $\Rightarrow$ *single* ()
| *Some* (*x*, *xq*) $\Rightarrow$ *empty*)

hide-const (**open**) *yield empty single append flat map bind if-seq those iterate-upto not-seq product*

hide-fact (**open**) *yield-def empty-def single-def append-def flat-def map-def bind-def if-seq-def those-def not-seq-def product-def*

end

76 Depth-Limited Sequences with failure element

theory *Limited-Sequence*
imports *Lazy-Sequence*
begin

76.1 Depth-Limited Sequence

type-synonym *'a dseq* = *natural* $\Rightarrow$ *bool* $\Rightarrow$ *'a lazy-sequence option*

definition *empty* :: *'a dseq*
where
empty = (λ- -. *Some Lazy-Sequence.empty*)

definition *single* :: $'a \Rightarrow 'a$ *dseq*
where
  single $x = (\lambda$- -. *Some* (*Lazy-Sequence.single* x))

definition *eval* :: $'a$ *dseq* $\Rightarrow$ *natural* $\Rightarrow$ *bool* $\Rightarrow$ $'a$ *lazy-sequence option*
where
  [*simp*]: *eval f i pol* = *f i pol*

definition *yield* :: $'a$ *dseq* $\Rightarrow$ *natural* $\Rightarrow$ *bool* $\Rightarrow$ ($'a \times 'a$ *dseq*) *option*
where
  yield f i pol = (*case eval f i pol of*
    None $\Rightarrow$ *None*
  | *Some s* $\Rightarrow$ (*map-option* $\circ$ *apsnd*) (λr - -. *Some r*) (*Lazy-Sequence.yield s*))

definition *map-seq* :: ($'a \Rightarrow 'b$ *dseq*) $\Rightarrow$ $'a$ *lazy-sequence* $\Rightarrow$ $'b$ *dseq*
where
  map-seq f xq i pol = *map-option Lazy-Sequence.flat*
    (*Lazy-Sequence.those* (*Lazy-Sequence.map* (λx. *f x i pol*) *xq*))

lemma *map-seq-code* [*code*]:
  map-seq f xq i pol = (*case Lazy-Sequence.yield xq of*
    None $\Rightarrow$ *Some Lazy-Sequence.empty*
  | *Some* (x, xq') $\Rightarrow$ (*case eval* (*f x*) *i pol of*
      None $\Rightarrow$ *None*
    | *Some yq* $\Rightarrow$ (*case map-seq f* xq' *i pol of*
        None $\Rightarrow$ *None*
      | *Some zq* $\Rightarrow$ *Some* (*Lazy-Sequence.append yq zq*))))
  ⟨*proof*⟩

definition *bind* :: $'a$ *dseq* $\Rightarrow$ ($'a \Rightarrow 'b$ *dseq*) $\Rightarrow$ $'b$ *dseq*
where
  bind x f = (λi *pol*.
     if $i = 0$ *then*
       (*if pol then Some Lazy-Sequence.empty else None*)
     else
       (*case x* ($i - 1$) *pol of*
         None $\Rightarrow$ *None*
       | *Some xq* $\Rightarrow$ *map-seq f xq i pol*))

definition *union* :: $'a$ *dseq* $\Rightarrow$ $'a$ *dseq* $\Rightarrow$ $'a$ *dseq*
where
  union x y = (λi *pol*. *case* (*x i pol*, *y i pol*) *of*
      (*Some xq*, *Some yq*) $\Rightarrow$ *Some* (*Lazy-Sequence.append xq yq*)
    | - $\Rightarrow$ *None*)

definition *if-seq* :: *bool* $\Rightarrow$ *unit dseq*
where
  if-seq b = (*if b then single* () *else empty*)

definition *not-seq* :: *unit dseq* ⇒ *unit dseq*
where
  not-seq x = (λ*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 = (λ*i pol. case g i pol of*
     None ⇒ *None*
   | *Some xq* ⇒ *Some* (*Lazy-Sequence.map f xq*))

76.2 Positive Depth-Limited Sequence

type-synonym '*a pos-dseq* = *natural* ⇒ '*a Lazy-Sequence.lazy-sequence*

definition *pos-empty* :: '*a pos-dseq*
where
  pos-empty = (λ*i. Lazy-Sequence.empty*)

definition *pos-single* :: '*a* ⇒ '*a pos-dseq*
where
  pos-single x = (λ*i. Lazy-Sequence.single x*)

definition *pos-bind* :: '*a pos-dseq* ⇒ ('*a* ⇒ '*b pos-dseq*) ⇒ '*b pos-dseq*
where
  pos-bind x f = (λ*i. Lazy-Sequence.bind* (*x i*) (λ*a. f a i*))

definition *pos-decr-bind* :: '*a pos-dseq* ⇒ ('*a* ⇒ '*b pos-dseq*) ⇒ '*b pos-dseq*
where
  pos-decr-bind x f = (λ*i.*
     if i = *0 then*
       Lazy-Sequence.empty
     else
       Lazy-Sequence.bind (*x* (*i* − *1*)) (λ*a. f a i*))

definition *pos-union* :: '*a pos-dseq* ⇒ '*a pos-dseq* ⇒ '*a pos-dseq*
where
  pos-union xq yq = (λ*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* :: (*natural* ⇒ '*a*) ⇒ *natural* ⇒ *natural* ⇒ '*a pos-dseq*
where

pos-iterate-upto f n m = (λi. *Lazy-Sequence.iterate-upto f n m*)

definition *pos-map* :: ($'a \Rightarrow 'b$) $\Rightarrow$ $'a$ *pos-dseq* $\Rightarrow$ $'b$ *pos-dseq*
where
pos-map f xq = (λi. *Lazy-Sequence.map f* (*xq i*))

76.3 Negative Depth-Limited Sequence

type-synonym $'a$ *neg-dseq* = *natural* $\Rightarrow$ $'a$ *Lazy-Sequence.hit-bound-lazy-sequence*

definition *neg-empty* :: $'a$ *neg-dseq*
where
neg-empty = (λi. *Lazy-Sequence.empty*)

definition *neg-single* :: $'a \Rightarrow 'a$ *neg-dseq*
where
neg-single x = (λi. *Lazy-Sequence.hb-single x*)

definition *neg-bind* :: $'a$ *neg-dseq* $\Rightarrow$ ($'a \Rightarrow 'b$ *neg-dseq*) $\Rightarrow$ $'b$ *neg-dseq*
where
neg-bind x f = (λi. *hb-bind* (*x i*) (λa. *f a i*))

definition *neg-decr-bind* :: $'a$ *neg-dseq* $\Rightarrow$ ($'a \Rightarrow 'b$ *neg-dseq*) $\Rightarrow$ $'b$ *neg-dseq*
where
neg-decr-bind x f = (λi.
  if i = *0 then*
    Lazy-Sequence.hit-bound
  else
    hb-bind (*x* (*i* − *1*)) (λa. *f a i*))

definition *neg-union* :: $'a$ *neg-dseq* $\Rightarrow$ $'a$ *neg-dseq* $\Rightarrow$ $'a$ *neg-dseq*
where
neg-union x y = (λi. *Lazy-Sequence.append* (*x i*) (*y i*))

definition *neg-if-seq* :: *bool* $\Rightarrow$ *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 = (λi. *Lazy-Sequence.iterate-upto* (λi. *Some* (*f i*)) *n m*)

definition *neg-map* :: ($'a \Rightarrow 'b$) $\Rightarrow$ $'a$ *neg-dseq* $\Rightarrow$ $'b$ *neg-dseq*
where
neg-map f xq = (λi. *Lazy-Sequence.hb-map f* (*xq i*))

76.4 Negation

definition *pos-not-seq* :: *unit neg-dseq* $\Rightarrow$ *unit pos-dseq*
where

pos-not-seq xq = (*λi. Lazy-Sequence.hb-not-seq* (*xq* (*3* ∗ *i*)))

definition *neg-not-seq* :: *unit pos-dseq* ⇒ *unit neg-dseq*
where
neg-not-seq x = (*λi. case Lazy-Sequence.yield* (*x i*) *of*
None => *Lazy-Sequence.hb-single* ()
| *Some* ((), *xq*) => *Lazy-Sequence.empty*)

⟨*ML*⟩

code-reserved *Eval Limited-Sequence*

hide-const (**open**) *yield empty single eval map-seq bind union if-seq not-seq map pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-seq pos-iterate-upto pos-not-seq pos-map neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-seq neg-iterate-upto neg-not-seq neg-map*

hide-fact (**open**) *yield-def empty-def single-def eval-def map-seq-def bind-def union-def if-seq-def not-seq-def map-def 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

77 Term evaluation using the generic code generator

theory *Code-Evaluation*
imports *Typerep Limited-Sequence*
keywords *value* :: *diag*
begin

77.1 Term representation

77.1.1 Terms and class *term-of*

datatype (*plugins only*: *extraction*) *term* = *dummy-term*

definition *Const* :: *String.literal* ⇒ *typerep* ⇒ *term* **where**
Const - - = *dummy-term*

definition *App* :: *term* ⇒ *term* ⇒ *term* **where**
App - - = *dummy-term*

definition *Abs* :: *String.literal* ⇒ *typerep* ⇒ *term* ⇒ *term* **where**
Abs - - - = *dummy-term*

definition *Free* :: *String.literal* ⇒ *typerep* ⇒ *term* **where**
Free - - = *dummy-term*

code-datatype *Const App Abs Free*

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*⟩

77.1.2 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*)

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*)

77.2 Tools setup and evaluation

context
begin

qualified definition *TERM-OF* :: *'a*::*term-of itself*
where

TERM-OF = *snd* (*Code-Evaluation.term-of* :: *'a* ⇒ -, *TYPE*(*'a*))

qualified definition *TERM-OF-EQUAL* :: *'a*::*term-of itself*
where
TERM-OF-EQUAL = *snd* (λ(*a*::*'a*). (*Code-Evaluation.term-of a*, *HOL.eq a*), *TYPE*(*'a*))

end

lemma *eq-eq-TrueD*:
fixes *x y* :: *'a*::{}
assumes (*x* ≡ *y*) ≡ *Trueprop True*
shows *x* ≡ *y*
⟨*proof*⟩

code-printing
type-constructor *term* ⇀ (*Eval*) *Term.term*
| **constant** *Const* ⇀ (*Eval*) *Term.Const*/ ((-), (-))
| **constant** *App* ⇀ (*Eval*) *Term.$*/ ((-), (-))
| **constant** *Abs* ⇀ (*Eval*) *Term.Abs*/ ((-), (-), (-))
| **constant** *Free* ⇀ (*Eval*) *Term.Free*/ ((-), (-))

⟨*ML*⟩

code-reserved *Eval Code-Evaluation*

⟨*ML*⟩

77.3 *term-of* instances

instantiation *fun* :: (*typerep*, *typerep*) *term-of*
begin

definition
term-of (*f* :: *'a* ⇒ *'b*) =
Const (*STR ''Pure.dummy-pattern''*)
(*Typerep.Typerep* (*STR ''fun''*) [*Typerep.typerep TYPE*(*'a*), *Typerep.typerep TYPE*(*'b*)])

instance ⟨*proof*⟩

end

declare [[*code drop*: *rec-term case-term*
term-of :: *typerep* ⇒ - *term-of* :: *term* ⇒ - *term-of* :: *String.literal* ⇒ -
term-of :: - *Predicate.pred* ⇒ *term term-of* :: - *Predicate.seq* ⇒ *term*]]

definition *case-char* :: *'a* ⇒ (*num* ⇒ *'a*) ⇒ *char* ⇒ *'a*
where *case-char f g c* = (*if c* = *0 then f else g* (*num-of-nat* (*nat-of-char c*)))

lemma *term-of-char* [*unfolded typerep-fun-def typerep-char-def typerep-num-def*, *code*]:
term-of =
case-char (*Const* (*STR* ''*Groups.zero-class.zero*'') (*TYPEREP*(*char*)))
(λk. *App* (*Const* (*STR* ''*String.Char*'') (*TYPEREP*(*num* ⇒ *char*))) (*term-of* *k*))
⟨*proof*⟩

lemma *term-of-string* [*code*]:
term-of s = *App* (*Const* (*STR* ''*STR*'')
(*Typerep.Typerep* (*STR* ''*fun*'') [*Typerep.Typerep* (*STR* ''*list*'') [*Typerep.Typerep* (*STR* ''*char*'') []],
Typerep.Typerep (*STR* ''*String.literal*'') []])) (*term-of* (*String.explode s*))
⟨*proof*⟩

code-printing
constant *term-of* :: *integer* ⇒ *term* ⇀ (*Eval*) *HOLogic.mk'-number*/ *HOLogic.code'-integerT*
| **constant** *term-of* :: *String.literal* ⇒ *term* ⇀ (*Eval*) *HOLogic.mk'-literal*

declare [[*code drop*: *term-of* :: *integer* ⇒ -]]

lemma *term-of-integer* [*unfolded typerep-fun-def typerep-num-def typerep-integer-def*, *code*]:
term-of (*i* :: *integer*) =
(*if i* > *0 then*
App (*Const* (*STR* ''*Num.numeral-class.numeral*'') (*TYPEREP*(*num* ⇒ *integer*)))
(*term-of* (*num-of-integer i*))
else if i = *0 then Const* (*STR* ''*Groups.zero-class.zero*'') *TYPEREP*(*integer*)
else
App (*Const* (*STR* ''*Groups.uminus-class.uminus*'') *TYPEREP*(*integer* ⇒ *integer*))
(*term-of* (− *i*)))
⟨*proof*⟩

code-reserved *Eval HOLogic*

77.4 Generic reification

⟨*ML*⟩

77.5 Diagnostic

definition *tracing* :: *String.literal* ⇒ '*a* ⇒ '*a* **where**
[*code del*]: *tracing s x* = *x*

code-printing
constant *tracing* :: *String.literal* => '*a* => '*a* ⇀ (*Eval*) *Code'-Evaluation.tracing*

hide-const *dummy-term valapp*
hide-const (**open**) *Const App Abs Free termify valtermify term-of tracing*

end

78 A simple counterexample generator performing random testing

theory *Quickcheck-Random*
imports *Random Code-Evaluation Enum*
begin

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

⟨*ML*⟩

78.1 Catching Match exceptions

axiomatization *catch-match* :: *'a => 'a => 'a*

code-printing
  constant *catch-match* ⇀ (*Quickcheck*) ((-) *handle Match =>* -)

78.2 The *random* class

class *random* = *typerep* +
  fixes *random* :: *natural* ⇒ *Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*

78.3 Fundamental and numeric types

instantiation *bool* :: *random*
begin

definition
  random i = *Random.range 2* ∘→
  (*λk. 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 :: *natural* ⇒ *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* (*Enum.enum* :: *char list*) ∘→ (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* :: *natural* ⇒ *Random.seed*
⇒ (*nat* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed*
where
random-nat i = *Random.range* (*i* + *1*) ∘→ (λ*k*. *Pair* (
let n = *nat-of-natural k*
in (*n*, λ-. *Code-Evaluation.term-of n*)))

instance ⟨*proof*⟩

end

instantiation *int* :: *random*
begin

definition
random i = *Random.range* (*2* ∗ *i* + *1*) ∘→ (λ*k*. *Pair* (
let j = (*if k* ≥ *i then int* (*nat-of-natural* (*k* − *i*)) *else* − (*int* (*nat-of-natural*

$(i - k))))$
  in $(j, \lambda$-. *Code-Evaluation.term-of* $j)))$

instance ⟨*proof*⟩

end

instantiation *natural* :: *random*
begin

definition *random-natural* :: *natural* ⇒ *Random.seed*
⇒ (*natural* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed*
where
random-natural i = *Random.range* $(i + 1)$ ∘→ (λn. *Pair* (n, λ-. *Code-Evaluation.term-of* n))

instance ⟨*proof*⟩

end

instantiation *integer* :: *random*
begin

definition *random-integer* :: *natural* ⇒ *Random.seed*
⇒ (*integer* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed*
where
random-integer i = *Random.range* $(2 * i + 1)$ ∘→ (λk. *Pair* (
let j = (*if* $k \geq i$ *then* *integer-of-natural* $(k - i)$ *else* $-$ (*integer-of-natural* $(i - k)$))
in $(j, \lambda$-. *Code-Evaluation.term-of* $j)))$

instance ⟨*proof*⟩

end

78.4 Complex generators

Towards $'a \Rightarrow 'b$

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* :: ({*equal*, *term-of*}, *random*) *random*
begin

definition
random-fun :: *natural* ⇒ *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* ∘→ *id*)

definition *beyond* :: *natural* ⇒ *natural* ⇒ *natural*
where *beyond k l* = (*if l* > *k then l else 0*)

lemma *beyond-zero*: *beyond k 0* = *0*
⟨*proof*⟩

definition (**in** *term-syntax*) [*code-unfold*]:
valterm-emptyset = *Code-Evaluation.valtermify* ({} :: ('*a* :: *typerep*) *set*)

definition (**in** *term-syntax*) [*code-unfold*]:
valtermify-insert x s = *Code-Evaluation.valtermify insert* {·} (*x* :: ('*a* :: *typerep* ∗ -)) {·} *s*

instantiation *set* :: (*random*) *random*
begin

fun *random-aux-set*
where
random-aux-set 0 j = *collapse* (*Random.select-weight* [(*1*, *Pair valterm-emptyset*)])
| *random-aux-set* (*Code-Numeral.Suc i*) *j* =
collapse (*Random.select-weight*
[(*1*, *Pair valterm-emptyset*),
(*Code-Numeral.Suc i*,
random j ∘→ (%*x*. *random-aux-set i j* ∘→ (%*s*. *Pair* (*valtermify-insert x s*))))])

lemma [*code*]:
random-aux-set i j =
collapse (*Random.select-weight* [(*1*, *Pair valterm-emptyset*),

(*i*, *random j* ∘→ (%*x*. *random-aux-set* (*i* − *1*) *j* ∘→ (%*s*. *Pair* (*valtermify-insert x s*))))])
⟨*proof*⟩

definition *random-set i* = *random-aux-set i i*

instance ⟨*proof*⟩

end

lemma *random-aux-rec*:
fixes *random-aux* :: *natural* ⇒ *'a*
assumes *random-aux 0* = *rhs 0*
and ⋀*k*. *random-aux* (*Code-Numeral.Suc k*) = *rhs* (*Code-Numeral.Suc k*)
shows *random-aux k* = *rhs k*
⟨*proof*⟩

78.5 Deriving random generators for datatypes

⟨*ML*⟩

78.6 Code setup

code-printing
constant *random-fun-aux* ⇀ (*Quickcheck*) *Random'-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 Random-Generators*

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

hide-const (**open**) *catch-match random collapse beyond random-fun-aux random-fun-lift*

hide-fact (**open**) *collapse-def beyond-def random-fun-lift-def*

end

79 The Random-Predicate Monad

theory *Random-Pred*
imports *Quickcheck-Random*
begin

fun *iter'* :: *'a itself* ⇒ *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ (*'a*::*random*) *Predicate.pred*
where

iter′ T nrandom sz seed = (*if nrandom* = *0 then bot-class.bot else*
  let ((*x*, -), *seed′*) = *Quickcheck-Random.random sz seed*
 in Predicate.Seq (%*u*. *Predicate.Insert x* (*iter′ T* (*nrandom* − *1*) *sz seed′*)))

definition *iter* :: *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ (′*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′*) = *Quickcheck-Random.random sz seed*
 in Predicate.Seq (%*u*. *Predicate.Insert x* (*iter* (*nrandom* − *1*) *sz seed′*)))
⟨*proof*⟩

type-synonym ′*a random-pred* = *Random.seed* ⇒ (′*a Predicate.pred* × *Random.seed*)

definition *empty* :: ′*a random-pred*
where *empty* = *Pair bot*

definition *single* :: ′*a* => ′*a random-pred*
where *single x* = *Pair* (*Predicate.single x*)

definition *bind* :: ′*a random-pred* ⇒ (′*a* ⇒ ′*b random-pred*) ⇒ ′*b random-pred*
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 random-pred* ⇒ ′*a random-pred* ⇒ ′*a random-pred*
where
union R1 R2 = (λ*s*. *let*
  (*P1*, *s′*) = *R1 s*; (*P2*, *s′′*) = *R2 s′*
 in (*sup-class.sup P1 P2*, *s′′*))

definition *if-randompred* :: *bool* ⇒ *unit random-pred*
where
if-randompred b = (*if b then single* () *else empty*)

definition *iterate-upto* :: (*natural* ⇒ ′*a*) => *natural* ⇒ *natural* ⇒ ′*a random-pred*
where
iterate-upto f n m = *Pair* (*Predicate.iterate-upto f n m*)

definition *not-randompred* :: *unit random-pred* ⇒ *unit random-pred*
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* :: $(Random.seed \Rightarrow ('a \times (unit \Rightarrow term)) \times Random.seed) \Rightarrow$ *'a random-pred*
where *Random g = scomp g (Pair o (Predicate.single o fst))*

definition *map* :: $('a \Rightarrow 'b) \Rightarrow$ *'a random-pred* $\Rightarrow$ *'b random-pred*
where *map f P = bind P (single o f)*

hide-const (**open**) *iter' iter empty single bind union if-randompred iterate-upto not-randompred Random map*

hide-fact *iter'.simps*

hide-fact (**open**) *iter-def empty-def single-def bind-def union-def if-randompred-def iterate-upto-def not-randompred-def Random-def map-def*

end

80 Various kind of sequences inside the random monad

theory *Random-Sequence*
imports *Random-Pred*
begin

type-synonym *'a random-dseq = natural* $\Rightarrow$ *natural* $\Rightarrow$ *Random.seed* $\Rightarrow$ *('a Limited-Sequence.dseq* $\times$ *Random.seed)*

definition *empty* :: *'a random-dseq*
where
empty = (%nrandom size. Pair (Limited-Sequence.empty))

definition *single* :: *'a => 'a random-dseq*
where
single x = (%nrandom size. Pair (Limited-Sequence.single x))

definition *bind* :: *'a random-dseq => ('a* $\Rightarrow$ *'b random-dseq)* $\Rightarrow$ *'b random-dseq*
where
bind R f = (λnrandom size s. let
(P, s') = R nrandom size s;
(s1, s2) = Random.split-seed s'
in (Limited-Sequence.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 (Limited-Sequence.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 = (λ*nrandom size s. let*
(S, s') = *R nrandom size s*
in (*Limited-Sequence.not-seq S*, s'))

definition *map* :: ('*a* => '*b*) => '*a random-dseq* => '*b random-dseq*
where
map f P = *bind P* (*single o f*)

fun *Random* :: (*natural* ⇒ *Random.seed* ⇒ (('*a* × (*unit* ⇒ *term*)) × *Random.seed*)) ⇒ '*a random-dseq*
where
Random g nrandom = (%*size. if nrandom* <= *0 then* (*Pair Limited-Sequence.empty*) *else*
(*scomp* (*g size*) (%*r. scomp* (*Random g* (*nrandom* − *1*) *size*) (%*rs. Pair* (*Limited-Sequence.union* (*Limited-Sequence.single* (*fst r*)) *rs*)))))

type-synonym '*a pos-random-dseq* = *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ '*a Limited-Sequence.pos-dseq*

definition *pos-empty* :: '*a pos-random-dseq*
where
pos-empty = (%*nrandom size seed. Limited-Sequence.pos-empty*)

definition *pos-single* :: '*a* => '*a pos-random-dseq*
where
pos-single x = (%*nrandom size seed. Limited-Sequence.pos-single x*)

definition *pos-bind* :: '*a pos-random-dseq* => ('*a* ⇒ '*b pos-random-dseq*) ⇒ '*b pos-random-dseq*
where
pos-bind R f = (λ*nrandom size seed. Limited-Sequence.pos-bind* (*R nrandom size seed*) (%*a. f a nrandom size seed*))

definition *pos-decr-bind* :: '*a pos-random-dseq* => ('*a* ⇒ '*b pos-random-dseq*) ⇒ '*b pos-random-dseq*
where
pos-decr-bind R f = (λ*nrandom size seed. Limited-Sequence.pos-decr-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. Limited-Sequence.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* :: (*natural* => *'a*) => *natural* => *natural* => *'a pos-random-dseq*
where
pos-iterate-upto f n m = (*λnrandom size seed. Limited-Sequence.pos-iterate-upto f n m*)

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*)

fun *iter* :: (*Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*)
⇒ *natural* ⇒ *Random.seed* ⇒ *'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 *pos-Random* :: (*natural* ⇒ *Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*)
⇒ *'a pos-random-dseq*
where
pos-Random g = (*%nrandom size seed depth. iter* (*g size*) *nrandom seed*)

type-synonym *'a neg-random-dseq* = *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ *'a Limited-Sequence.neg-dseq*

definition *neg-empty* :: *'a neg-random-dseq*
where
neg-empty = (*%nrandom size seed. Limited-Sequence.neg-empty*)

definition *neg-single* :: *'a* => *'a neg-random-dseq*
where
neg-single x = (*%nrandom size seed. Limited-Sequence.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. Limited-Sequence.neg-bind* (*R nrandom size seed*) (*%a. f a nrandom size seed*))

definition *neg-decr-bind* :: *'a neg-random-dseq* => (*'a* ⇒ *'b neg-random-dseq*) ⇒ *'b neg-random-dseq*
where
neg-decr-bind R f = (*λnrandom size seed. Limited-Sequence.neg-decr-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. Limited-Sequence.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* :: (*natural* => *'a*) => *natural* => *natural* => *'a neg-random-dseq*
where
neg-iterate-upto f n m = (*λnrandom size seed. Limited-Sequence.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. Limited-Sequence.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*)

definition *pos-not-random-dseq* :: *unit neg-random-dseq* => *unit pos-random-dseq*
where
pos-not-random-dseq R = (*λnrandom size seed. Limited-Sequence.pos-not-seq* (*R nrandom size seed*))

hide-const (**open**)
empty single bind union if-random-dseq not-random-dseq map Random
pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-random-dseq pos-iterate-upto
pos-not-random-dseq pos-map iter pos-Random
neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-random-dseq neg-iterate-upto
neg-not-random-dseq neg-map

hide-fact (**open**) *empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def*
map-def Random.simps
pos-empty-def pos-single-def pos-bind-def pos-decr-bind-def pos-union-def pos-if-random-dseq-def
pos-iterate-upto-def pos-not-random-dseq-def pos-map-def iter.simps pos-Random-def
neg-empty-def neg-single-def neg-bind-def neg-decr-bind-def neg-union-def neg-if-random-dseq-def
neg-iterate-upto-def neg-not-random-dseq-def neg-map-def

end

81 A simple counterexample generator performing exhaustive testing

theory *Quickcheck-Exhaustive*
imports *Quickcheck-Random*
keywords *quickcheck-generator* :: *thy-decl*
begin

81.1 Basic operations for exhaustive generators

definition *orelse* :: *'a option* ⇒ *'a option* ⇒ *'a option* (**infixr** *orelse 55*)
  where [*code-unfold*]: *x orelse y* = (*case x of Some x'* ⇒ *Some x'* | *None* ⇒ *y*)

81.2 Exhaustive generator type classes

class *exhaustive* = *term-of* +
  fixes *exhaustive* :: (*'a* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*

class *full-exhaustive* = *term-of* +
  fixes *full-exhaustive* ::
    (*'a* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*

instantiation *natural* :: *full-exhaustive*
begin

function *full-exhaustive-natural'* ::
    (*natural* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
      natural ⇒ *natural* ⇒ (*bool* × *term list*) *option*
  where *full-exhaustive-natural' f d i* =
    (*if d* < *i then None*
     else (*f* (*i*, λ-. *Code-Evaluation.term-of i*)) *orelse* (*full-exhaustive-natural' f d* (*i* + *1*)))
⟨*proof*⟩

termination
  ⟨*proof*⟩

definition *full-exhaustive f d* = *full-exhaustive-natural' f d 0*

instance ⟨*proof*⟩

end

instantiation *natural* :: *exhaustive*

begin

function *exhaustive-natural′* ::
$(natural \Rightarrow (bool \times term\ list)\ option) \Rightarrow natural \Rightarrow natural \Rightarrow (bool \times term\ list)\ option$
where *exhaustive-natural′ f d i* =
(*if* $d < i$ *then None*
else (*f i orelse exhaustive-natural′ f d* $(i + 1)$)))
⟨*proof*⟩

termination
⟨*proof*⟩

definition *exhaustive f d* = *exhaustive-natural′ f d 0*

instance ⟨*proof*⟩

end

instantiation *integer* :: *exhaustive*
begin

function *exhaustive-integer′* ::
$(integer \Rightarrow (bool \times term\ list)\ option) \Rightarrow integer \Rightarrow integer \Rightarrow (bool \times term\ list)\ option$
where *exhaustive-integer′ f d i* =
(*if* $d < i$ *then None else* (*f i orelse exhaustive-integer′ f d* $(i + 1)$)))
⟨*proof*⟩

termination
⟨*proof*⟩

definition *exhaustive f d* = *exhaustive-integer′ f* (*integer-of-natural d*) (− (*integer-of-natural d*))

instance ⟨*proof*⟩

end

instantiation *integer* :: *full-exhaustive*
begin

function *full-exhaustive-integer′* ::
$(integer \times (unit \Rightarrow term) \Rightarrow (bool \times term\ list)\ option) \Rightarrow$
$integer \Rightarrow integer \Rightarrow (bool \times term\ list)\ option$
where *full-exhaustive-integer′ f d i* =
(*if* $d < i$ *then None*
else
(*case f* (*i*, λ-. *Code-Evaluation.term-of i*) *of*

Some t ⇒ *Some t*
| *None* ⇒ *full-exhaustive-integer′ f d* (*i* + *1*)))
⟨*proof*⟩

termination
⟨*proof*⟩

definition *full-exhaustive f d* =
full-exhaustive-integer′ f (*integer-of-natural d*) (− (*integer-of-natural d*))

instance ⟨*proof*⟩

end

instantiation *nat* :: *exhaustive*
begin

definition *exhaustive f d* = *exhaustive* (λ*x*. *f* (*nat-of-natural x*)) *d*

instance ⟨*proof*⟩

end

instantiation *nat* :: *full-exhaustive*
begin

definition *full-exhaustive f d* =
full-exhaustive (λ(*x*, *xt*). *f* (*nat-of-natural x*, λ-. *Code-Evaluation.term-of* (*nat-of-natural x*))) *d*

instance ⟨*proof*⟩

end

instantiation *int* :: *exhaustive*
begin

function *exhaustive-int′* ::
(*int* ⇒ (*bool* × *term list*) *option*) ⇒ *int* ⇒ *int* ⇒ (*bool* × *term list*) *option*
where *exhaustive-int′ f d i* =
(*if d* < *i then None else* (*f i orelse exhaustive-int′ f d* (*i* + *1*)))
⟨*proof*⟩

termination
⟨*proof*⟩

definition *exhaustive f d* =
exhaustive-int′ f (*int-of-integer* (*integer-of-natural d*))
(− (*int-of-integer* (*integer-of-natural d*)))

instance ⟨*proof*⟩

end

instantiation *int* :: *full-exhaustive*
begin

function *full-exhaustive-int'* ::
  $(int \times (unit \Rightarrow term) \Rightarrow (bool \times term\ list)\ option) \Rightarrow$
    $int \Rightarrow int \Rightarrow (bool \times term\ list)\ option$
 where *full-exhaustive-int' f d i* =
  (*if d* < *i then None*
   else
    (*case f* (*i*, λ-. *Code-Evaluation.term-of i*) *of*
     Some t ⇒ *Some t*
    | *None* ⇒ *full-exhaustive-int' f d* (*i* + *1*)))
⟨*proof*⟩

termination
 ⟨*proof*⟩

definition *full-exhaustive f d* =
 full-exhaustive-int' f (*int-of-integer* (*integer-of-natural d*))
  (− (*int-of-integer* (*integer-of-natural d*)))

instance ⟨*proof*⟩

end

instantiation *prod* :: (*exhaustive*, *exhaustive*) *exhaustive*
begin

definition *exhaustive f d* = *exhaustive* (λ*x*. *exhaustive* (λ*y*. *f* ((*x*, *y*))) *d*) *d*

instance ⟨*proof*⟩

end

definition (**in** *term-syntax*)
 [*code-unfold*]: *valtermify-pair x y* =
  Code-Evaluation.valtermify (*Pair* :: *'a*::*typerep* ⇒ *'b*::*typerep* ⇒ *'a* × *'b*) {·} *x*
{·} *y*

instantiation *prod* :: (*full-exhaustive*, *full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive f d* =
 full-exhaustive (λ*x*. *full-exhaustive* (λ*y*. *f* (*valtermify-pair x y*)) *d*) *d*

instance ⟨*proof*⟩

end

instantiation *set* :: (*exhaustive*) *exhaustive*
begin

fun *exhaustive-set*
where
exhaustive-set f i =
(*if i* = *0 then None*
else
f {} *orelse*
exhaustive-set
(λA. *f A orelse exhaustive* (λx. *if x* ∈ *A then None else f* (*insert x A*)) (*i* − *1*)) (*i* − *1*))

instance ⟨*proof*⟩

end

instantiation *set* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-set*
where
full-exhaustive-set f i =
(*if i* = *0 then None*
else
f valterm-emptyset orelse
full-exhaustive-set
(λA. *f A orelse Quickcheck-Exhaustive.full-exhaustive*
(λx. *if fst x* ∈ *fst A then None else f* (*valtermify-insert x A*)) (*i* − *1*)) (*i* − *1*))

instance ⟨*proof*⟩

end

instantiation *fun* :: ({*equal*,*exhaustive*}, *exhaustive*) *exhaustive*
begin

fun *exhaustive-fun'* ::
(('*a* ⇒ '*b*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
exhaustive-fun' f i d =
(*exhaustive* (λb. *f* (λ-. *b*)) *d*) *orelse*

(*if* $i > 1$ *then*
exhaustive-fun'
(λg. *exhaustive* (λa. *exhaustive* (λb. *f* (*g*(*a* := *b*))) *d*) *d*) ($i - 1$) *d* *else* *None*)

definition *exhaustive-fun* ::
(('*a* ⇒ '*b*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where *exhaustive-fun f d* = *exhaustive-fun' f d d*

instance ⟨*proof*⟩

end

definition [*code-unfold*]:
valtermify-absdummy =
(λ(*v*, *t*).
(λ-::'*a*. *v*,
λ*u*::*unit*. *Code-Evaluation.Abs* (*STR ''x''*) (*Typerep.typerep TYPE*('*a*::*typerep*)) (*t* ())))

definition (**in** *term-syntax*)
[*code-unfold*]: *valtermify-fun-upd g a b* =
Code-Evaluation.valtermify
(*fun-upd* :: ('*a*::*typerep* ⇒ '*b*::*typerep*) ⇒ '*a* ⇒ '*b* ⇒ '*a* ⇒ '*b*) {·} *g* {·} *a* {·} *b*

instantiation *fun* :: ({*equal*,*full-exhaustive*}, *full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-fun'* ::
(('*a* ⇒ '*b*) × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
natural ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
full-exhaustive-fun' f i d =
full-exhaustive (λv. *f* (*valtermify-absdummy v*)) *d* *orelse*
(*if* $i > 1$ *then*
full-exhaustive-fun'
(λg. *full-exhaustive*
(λa. *full-exhaustive* (λb. *f* (*valtermify-fun-upd g a b*)) *d*) *d*) ($i - 1$) *d*
else None)

definition *full-exhaustive-fun* ::
(('*a* ⇒ '*b*) × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
natural ⇒ (*bool* × *term list*) *option*
where *full-exhaustive-fun f d* = *full-exhaustive-fun' f d d*

instance ⟨*proof*⟩

end

81.2.1 A smarter enumeration scheme for functions over finite datatypes

class *check-all* = *enum* + *term-of* +
fixes *check-all* :: ('a × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ (bool * term list) option
fixes *enum-term-of* :: 'a itself ⇒ unit ⇒ term list

fun *check-all-n-lists* :: ('a::check-all list × (unit ⇒ term list) ⇒
(bool × term list) option) ⇒ natural ⇒ (bool * term list) option
where
check-all-n-lists f n =
(if n = 0 then f ([], (λ-. []))
else check-all (λ(x, xt).
check-all-n-lists (λ(xs, xst). f ((x # xs), (λ-. (xt () # xst ())))) (n − 1)))

definition (**in** *term-syntax*)
[code-unfold]: *termify-fun-upd g a b* =
(Code-Evaluation.termify
(fun-upd :: ('a::typerep ⇒ 'b::typerep) ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'b) <·> g <·> a <·> b)

definition *mk-map-term* ::
(unit ⇒ typerep) ⇒ (unit ⇒ typerep) ⇒
(unit ⇒ term list) ⇒ (unit ⇒ term list) ⇒ unit ⇒ term
where *mk-map-term T1 T2 domm rng* =
(λ-.
let
T1 = T1 ();
T2 = T2 ();
update-term =
(λg (a, b).
Code-Evaluation.App (Code-Evaluation.App (Code-Evaluation.App
(Code-Evaluation.Const (STR ''Fun.fun-upd'')
(Typerep.Typerep (STR ''fun'') [Typerep.Typerep (STR ''fun'') [T1, T2],
Typerep.Typerep (STR ''fun'') [T1,
Typerep.Typerep (STR ''fun'') [T2, Typerep.Typerep (STR ''fun'') [T1, T2]]]]))
g) a) b)
in
List.foldl update-term
(Code-Evaluation.Abs (STR ''x'') T1
(Code-Evaluation.Const (STR ''HOL.undefined'') T2)) (zip (domm ()) (rng ())))

instantiation *fun* :: ({equal,check-all}, check-all) check-all
begin

definition

check-all f =
  (*let*
    mk-term =
      mk-map-term
        (λ-. *Typerep.typerep* (*TYPE*('*a*)))
        (λ-. *Typerep.typerep* (*TYPE*('*b*)))
        (*enum-term-of* (*TYPE*('*a*)));
    enum = (*Enum.enum* :: '*a list*)
  in
    check-all-n-lists
      (λ(*ys*, *yst*). *f* (*the o map-of* (*zip enum ys*), *mk-term yst*))
      (*natural-of-nat* (*length enum*)))

definition *enum-term-of-fun* :: ('*a* ⇒ '*b*) *itself* ⇒ *unit* ⇒ *term list*
  where *enum-term-of-fun* =
    (λ- -.
      let
        enum-term-of-a = *enum-term-of* (*TYPE*('*a*));
        mk-term =
          mk-map-term
            (λ-. *Typerep.typerep* (*TYPE*('*a*)))
            (λ-. *Typerep.typerep* (*TYPE*('*b*)))
            enum-term-of-a
      in
        map (λ*ys*. *mk-term* (λ-. *ys*) ())
         (*List.n-lists* (*length* (*enum-term-of-a* ())) (*enum-term-of* (*TYPE*('*b*)) ())))

instance ⟨*proof*⟩

end

fun (**in** *term-syntax*) *check-all-subsets* ::
  (('*a*::*typerep*) *set* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
    ('*a* × (*unit* ⇒ *term*)) *list* ⇒ (*bool* × *term list*) *option*
where
  check-all-subsets f [] = *f valterm-emptyset*
| *check-all-subsets f* (*x* # *xs*) =
   check-all-subsets (λ*s*. *case f s of Some ts* ⇒ *Some ts* | *None* ⇒ *f* (*valtermify-insert x s*)) *xs*

definition (**in** *term-syntax*)
  [*code-unfold*]: *term-emptyset* = *Code-Evaluation.termify* ({} :: ('*a*::*typerep*) *set*)

definition (**in** *term-syntax*)
  [*code-unfold*]: *termify-insert x s* =
    Code-Evaluation.termify (*insert* :: ('*a*::*typerep*) ⇒ '*a set* ⇒ '*a set*) <·> *x* <·> *s*

definition (**in** *term-syntax*) *setify* :: ($'a$::*typerep*) *itself* $\Rightarrow$ *term list* $\Rightarrow$ *term*
where
setify T ts = *foldr* (*termify-insert T*) *ts* (*term-emptyset T*)

instantiation *set* :: (*check-all*) *check-all*
begin

definition
check-all-set f =
check-all-subsets f
(*zip* (*Enum.enum* :: $'a$ *list*)
(*map* (λa. λu :: *unit*. *a*) (*Quickcheck-Exhaustive.enum-term-of* (*TYPE* ($'a$))
())))

definition *enum-term-of-set* :: $'a$ *set itself* $\Rightarrow$ *unit* $\Rightarrow$ *term list*
where *enum-term-of-set* - - =
map (*setify* (*TYPE*($'a$))) (*subseqs* (*Quickcheck-Exhaustive.enum-term-of* (*TYPE*($'a$))
()))

instance ⟨*proof*⟩

end

instantiation *unit* :: *check-all*
begin

definition *check-all f* = *f* (*Code-Evaluation.valtermify* ())

definition *enum-term-of-unit* :: *unit itself* $\Rightarrow$ *unit* $\Rightarrow$ *term list*
where *enum-term-of-unit* = (λ- -. [*Code-Evaluation.term-of* ()])

instance ⟨*proof*⟩

end

instantiation *bool* :: *check-all*
begin

definition
check-all f =
(*case f* (*Code-Evaluation.valtermify False*) *of*
Some x' $\Rightarrow$ *Some* x'
| *None* $\Rightarrow$ *f* (*Code-Evaluation.valtermify True*))

definition *enum-term-of-bool* :: *bool itself* $\Rightarrow$ *unit* $\Rightarrow$ *term list*
where *enum-term-of-bool* = (λ- -. *map Code-Evaluation.term-of* (*Enum.enum*
:: *bool list*))

instance ⟨*proof*⟩

end

definition (**in** *term-syntax*) [*code-unfold*]:
termify-pair x y =
Code-Evaluation.termify (*Pair* :: $'a$::*typerep* $\Rightarrow$ $'b$ :: *typerep* $\Rightarrow$ $'a * 'b$) <·> *x* <·> *y*

instantiation *prod* :: (*check-all*, *check-all*) *check-all*
begin

definition *check-all f* = *check-all* (λx. *check-all* (λy. *f* (*valtermify-pair x y*)))

definition *enum-term-of-prod* :: ($'a * 'b$) *itself* $\Rightarrow$ *unit* $\Rightarrow$ *term list*
where *enum-term-of-prod* =
(λ- -.
map (λ(*x*, *y*). *termify-pair TYPE*($'a$) *TYPE*($'b$) *x y*)
(*List.product* (*enum-term-of* (*TYPE*($'a$)) ()) (*enum-term-of* (*TYPE*($'b$)) ())))

instance ⟨*proof*⟩

end

definition (**in** *term-syntax*)
[*code-unfold*]: *valtermify-Inl x* =
Code-Evaluation.valtermify (*Inl* :: $'a$::*typerep* $\Rightarrow$ $'a + 'b$ :: *typerep*) {·} *x*

definition (**in** *term-syntax*)
[*code-unfold*]: *valtermify-Inr x* =
Code-Evaluation.valtermify (*Inr* :: $'b$::*typerep* $\Rightarrow$ $'a$::*typerep* + $'b$) {·} *x*

instantiation *sum* :: (*check-all*, *check-all*) *check-all*
begin

definition
check-all f = *check-all* (λa. *f* (*valtermify-Inl a*)) *orelse check-all* (λb. *f* (*valtermify-Inr b*))

definition *enum-term-of-sum* :: ($'a + 'b$) *itself* $\Rightarrow$ *unit* $\Rightarrow$ *term list*
where *enum-term-of-sum* =
(λ- -.
let
T1 = *Typerep.typerep* (*TYPE*($'a$));
T2 = *Typerep.typerep* (*TYPE*($'b$))
in
map
(*Code-Evaluation.App* (*Code-Evaluation.Const* (*STR* ''*Sum-Type.Inl*'')

(*Typerep.Typerep* (*STR ''fun''*) [*T1*, *Typerep.Typerep* (*STR ''Sum-Type.sum''*) [*T1*, *T2*]])))
(*enum-term-of* (*TYPE*('*a*)) ()) @
map
(*Code-Evaluation.App* (*Code-Evaluation.Const* (*STR ''Sum-Type.Inr''*)
(*Typerep.Typerep* (*STR ''fun''*) [*T2*, *Typerep.Typerep* (*STR ''Sum-Type.sum''*) [*T1*, *T2*]])))
(*enum-term-of* (*TYPE*('*b*)) ()))

instance ⟨*proof*⟩

end

instantiation *char* :: *check-all*
begin

primrec *check-all-char'* ::
(*char* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *char list* ⇒ (*bool* × *term list*) *option*
where *check-all-char' f* [] = *None*
| *check-all-char' f* (*c* # *cs*) = *f* (*c*, λ-. *Code-Evaluation.term-of c*)
orelse check-all-char' f cs

definition *check-all-char* ::
(*char* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ (*bool* × *term list*) *option*
where *check-all f* = *check-all-char' f Enum.enum*

definition *enum-term-of-char* :: *char itself* ⇒ *unit* ⇒ *term list*
where
enum-term-of-char = (λ- -. *map Code-Evaluation.term-of* (*Enum.enum* :: *char list*))

instance ⟨*proof*⟩

end

instantiation *option* :: (*check-all*) *check-all*
begin

definition
check-all f =
f (*Code-Evaluation.valtermify* (*None* :: '*a option*)) *orelse*
check-all
(λ(*x*, *t*).
f
(*Some x*,
λ-. *Code-Evaluation.App*
(*Code-Evaluation.Const* (*STR ''Option.option.Some''*)
(*Typerep.Typerep* (*STR ''fun''*)

[*Typerep.typerep TYPE('a)*,
Typerep.Typerep (STR ''Option.option'') [Typerep.typerep TYPE('a)]]]))
(t ())))

definition *enum-term-of-option* :: *'a option itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-option* =
(λ - -.
Code-Evaluation.term-of (None :: 'a option) #
(*map*
(*Code-Evaluation.App*
(*Code-Evaluation.Const (STR ''Option.option.Some'')*
(*Typerep.Typerep (STR ''fun'')*
[*Typerep.typerep TYPE('a)*,
Typerep.Typerep (STR ''Option.option'') [Typerep.typerep TYPE('a)]]]))
(*enum-term-of (TYPE('a)) ()*)))

instance ⟨*proof*⟩

end

instantiation *Enum.finite-1* :: *check-all*
begin

definition *check-all f* = *f (Code-Evaluation.valtermify Enum.finite-1.*a_1)

definition *enum-term-of-finite-1* :: *Enum.finite-1 itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-1* = (λ- -. [*Code-Evaluation.term-of Enum.finite-1.*a_1])

instance ⟨*proof*⟩

end

instantiation *Enum.finite-2* :: *check-all*
begin

definition
check-all f =
(*f (Code-Evaluation.valtermify Enum.finite-2.*a_1) *orelse*
f (Code-Evaluation.valtermify Enum.finite-2.a_2))

definition *enum-term-of-finite-2* :: *Enum.finite-2 itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-2* =
(λ- -. *map Code-Evaluation.term-of (Enum.enum :: Enum.finite-2 list)*)

instance ⟨*proof*⟩

end

instantiation *Enum.finite-3* :: *check-all*
begin

definition
check-all f =
(f (*Code-Evaluation.valtermify Enum.finite-3*.a_1) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-3*.a_2) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-3*.a_3))

definition *enum-term-of-finite-3* :: *Enum.finite-3 itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-3* =
(λ- -. *map Code-Evaluation.term-of* (*Enum.enum* :: *Enum.finite-3 list*))

instance ⟨*proof*⟩

end

instantiation *Enum.finite-4* :: *check-all*
begin

definition
check-all f =
f (*Code-Evaluation.valtermify Enum.finite-4*.a_1) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-4*.a_2) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-4*.a_3) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-4*.a_4)

definition *enum-term-of-finite-4* :: *Enum.finite-4 itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-4* =
(λ- -. *map Code-Evaluation.term-of* (*Enum.enum* :: *Enum.finite-4 list*))

instance ⟨*proof*⟩

end

81.3 Bounded universal quantifiers

class *bounded-forall* =
fixes *bounded-forall* :: (*'a* ⇒ *bool*) ⇒ *natural* ⇒ *bool*

81.4 Fast exhaustive combinators

class *fast-exhaustive* = *term-of* +
fixes *fast-exhaustive* :: (*'a* ⇒ *unit*) ⇒ *natural* ⇒ *unit*

axiomatization *throw-Counterexample* :: *term list* ⇒ *unit*
axiomatization *catch-Counterexample* :: *unit* ⇒ *term list option*

code-printing
constant *throw-Counterexample* ⇀

(*Quickcheck*) *raise* (*Exhaustive′-Generators.Counterexample* -)
| **constant** *catch-Counterexample* ⇀
(*Quickcheck*) (((-); *NONE*) *handle Exhaustive′-Generators.Counterexample ts* ⇒ *SOME ts*)

81.5 Continuation passing style functions as plus monad

type-synonym *′a cps* = (*′a* ⇒ *term list option*) ⇒ *term list option*

definition *cps-empty* :: *′a cps*
where *cps-empty* = (*λcont. None*)

definition *cps-single* :: *′a* ⇒ *′a cps*
where *cps-single v* = (*λcont. cont v*)

definition *cps-bind* :: *′a cps* ⇒ (*′a* ⇒ *′b cps*) ⇒ *′b cps*
where *cps-bind m f* = (*λcont. m* (*λa.* (*f a*) *cont*))

definition *cps-plus* :: *′a cps* ⇒ *′a cps* ⇒ *′a cps*
where *cps-plus a b* = (*λc. case a c of None* ⇒ *b c* | *Some x* ⇒ *Some x*)

definition *cps-if* :: *bool* ⇒ *unit cps*
where *cps-if b* = (*if b then cps-single* () *else cps-empty*)

definition *cps-not* :: *unit cps* ⇒ *unit cps*
where *cps-not n* = (*λc. case n* (*λu. Some* []) *of None* ⇒ *c* () | *Some* - ⇒ *None*)

type-synonym *′a pos-bound-cps* =
(*′a* ⇒ (*bool* ∗ *term list*) *option*) ⇒ *natural* ⇒ (*bool* ∗ *term list*) *option*

definition *pos-bound-cps-empty* :: *′a pos-bound-cps*
where *pos-bound-cps-empty* = (*λcont i. None*)

definition *pos-bound-cps-single* :: *′a* ⇒ *′a pos-bound-cps*
where *pos-bound-cps-single v* = (*λcont i. cont v*)

definition *pos-bound-cps-bind* :: *′a pos-bound-cps* ⇒ (*′a* ⇒ *′b pos-bound-cps*) ⇒ *′b pos-bound-cps*
where *pos-bound-cps-bind m f* = (*λcont i. if i* = *0 then None else* (*m* (*λa.* (*f a*) *cont i*) (*i* − *1*)))

definition *pos-bound-cps-plus* :: *′a pos-bound-cps* ⇒ *′a pos-bound-cps* ⇒ *′a pos-bound-cps*
where *pos-bound-cps-plus a b* = (*λc i. case a c i of None* ⇒ *b c i* | *Some x* ⇒ *Some x*)

definition *pos-bound-cps-if* :: *bool* ⇒ *unit pos-bound-cps*
where *pos-bound-cps-if b* = (*if b then pos-bound-cps-single* () *else pos-bound-cps-empty*)

datatype (*plugins only*: *code extraction*) (*dead ′a*) *unknown* =

Unknown | *Known* $'a$

datatype (*plugins only*: *code extraction*) (*dead* $'a$) *three-valued* =
Unknown-value | *Value* $'a$ | *No-value*

type-synonym $'a$ *neg-bound-cps* =
($'a$ *unknown* ⇒ *term list three-valued*) ⇒ *natural* ⇒ *term list three-valued*

definition *neg-bound-cps-empty* :: $'a$ *neg-bound-cps*
where *neg-bound-cps-empty* = (λ*cont i*. *No-value*)

definition *neg-bound-cps-single* :: $'a$ ⇒ $'a$ *neg-bound-cps*
where *neg-bound-cps-single v* = (λ*cont i*. *cont* (*Known v*))

definition *neg-bound-cps-bind* :: $'a$ *neg-bound-cps* ⇒ ($'a$ ⇒ $'b$ *neg-bound-cps*) ⇒
$'b$ *neg-bound-cps*
where *neg-bound-cps-bind m f* =
(λ*cont i*.
if i = *0 then cont Unknown*
else m (λ*a*. *case a of Unknown* ⇒ *cont Unknown* | *Known a'* ⇒ *f a' cont i*)
(*i* − *1*))

definition *neg-bound-cps-plus* :: $'a$ *neg-bound-cps* ⇒ $'a$ *neg-bound-cps* ⇒ $'a$ *neg-bound-cps*
where *neg-bound-cps-plus a b* =
(λ*c i*.
case a c i of
No-value ⇒ *b c i*
| *Value x* ⇒ *Value x*
| *Unknown-value* ⇒
(*case b c i of*
No-value ⇒ *Unknown-value*
| *Value x* ⇒ *Value x*
| *Unknown-value* ⇒ *Unknown-value*))

definition *neg-bound-cps-if* :: *bool* ⇒ *unit neg-bound-cps*
where *neg-bound-cps-if b* = (*if b then neg-bound-cps-single* () *else neg-bound-cps-empty*)

definition *neg-bound-cps-not* :: *unit pos-bound-cps* ⇒ *unit neg-bound-cps*
where *neg-bound-cps-not n* =
(λ*c i*. *case n* (λ*u*. *Some* (*True*, [])) *i of None* ⇒ *c* (*Known* ()) | *Some* - ⇒
No-value)

definition *pos-bound-cps-not* :: *unit neg-bound-cps* ⇒ *unit pos-bound-cps*
where *pos-bound-cps-not n* =
(λ*c i*. *case n* (λ*u*. *Value* []) *i of No-value* ⇒ *c* () | *Value* - ⇒ *None* |
Unknown-value ⇒ *None*)

81.6 Defining generators for any first-order data type

axiomatization *unknown* :: *'a*

notation (**output**) *unknown* (*?*)

⟨*ML*⟩

declare [[*quickcheck-batch-tester* = *exhaustive*]]

81.7 Defining generators for abstract types

⟨*ML*⟩

hide-fact (**open**) *orelse-def*
no-notation *orelse* (**infixr** *orelse 55*)

hide-const *valtermify-absdummy valtermify-fun-upd*
valterm-emptyset valtermify-insert
valtermify-pair valtermify-Inl valtermify-Inr
termify-fun-upd term-emptyset termify-insert termify-pair setify

hide-const (**open**)
exhaustive full-exhaustive
exhaustive-int' full-exhaustive-int'
exhaustive-integer' full-exhaustive-integer'
exhaustive-natural' full-exhaustive-natural'
throw-Counterexample catch-Counterexample
check-all enum-term-of
orelse unknown mk-map-term check-all-n-lists check-all-subsets

hide-type (**open**) *cps pos-bound-cps neg-bound-cps unknown three-valued*

hide-const (**open**) *cps-empty cps-single cps-bind cps-plus cps-if cps-not*
pos-bound-cps-empty pos-bound-cps-single pos-bound-cps-bind
pos-bound-cps-plus pos-bound-cps-if pos-bound-cps-not
neg-bound-cps-empty neg-bound-cps-single neg-bound-cps-bind
neg-bound-cps-plus neg-bound-cps-if neg-bound-cps-not
Unknown Known Unknown-value Value No-value

end

82 A compiler for predicates defined by introduction rules

theory *Predicate-Compile*
imports *Random-Sequence Quickcheck-Exhaustive*

keywords
code-pred :: *thy-goal* **and**
values :: *diag*
begin

⟨*ML*⟩

82.1 Set membership as a generator predicate

Introduce a new constant for membership to allow fine-grained control in code equations.

definition *contains* :: *'a set => 'a => bool*
where *contains A x* ⟷ *x* : *A*

definition *contains-pred* :: *'a set => 'a => unit Predicate.pred*
where *contains-pred A x* = (*if x* : *A then Predicate.single* () *else bot*)

lemma *pred-of-setE*:
assumes *Predicate.eval* (*pred-of-set A*) *x*
obtains *contains A x*
⟨*proof*⟩

lemma *pred-of-setI*: *contains A x ==> Predicate.eval* (*pred-of-set A*) *x*
⟨*proof*⟩

lemma *pred-of-set-eq*: *pred-of-set* ≡ λ*A*. *Predicate.Pred* (*contains A*)
⟨*proof*⟩

lemma *containsI*: *x* ∈ *A ==> contains A x*
⟨*proof*⟩

lemma *containsE*: **assumes** *contains A x*
obtains *A' x'* **where** *A* = *A' x* = *x' x* : *A*
⟨*proof*⟩

lemma *contains-predI*: *contains A x ==> Predicate.eval* (*contains-pred A x*) ()
⟨*proof*⟩

lemma *contains-predE*:
assumes *Predicate.eval* (*contains-pred A x*) *y*
obtains *contains A x*
⟨*proof*⟩

lemma *contains-pred-eq*: *contains-pred* ≡ λ*A x*. *Predicate.Pred* (λ*y*. *contains A x*)
⟨*proof*⟩

lemma *contains-pred-notI*:
¬ *contains A x ==> Predicate.eval* (*Predicate.not-pred* (*contains-pred A x*)) ()

⟨*proof*⟩

⟨*ML*⟩

hide-const (**open**) *contains contains-pred*
hide-fact (**open**) *pred-of-setE pred-of-setI pred-of-set-eq*
containsI containsE contains-predI contains-predE contains-pred-eq contains-pred-notI

end

83 Counterexample generator performing narrowing-based testing

theory *Quickcheck-Narrowing*
imports *Quickcheck-Random*
keywords *find-unused-assms* :: *diag*
begin

83.1 Counterexample generator

83.1.1 Code generation setup

⟨*ML*⟩

code-printing
code-module *Typerep* ⇀ (*Haskell-Quickcheck*) ‹
data Typerep = *Typerep String* [*Typerep*]
›
| **type-constructor** *typerep* ⇀ (*Haskell-Quickcheck*) *Typerep.Typerep*
| **constant** *Typerep.Typerep* ⇀ (*Haskell-Quickcheck*) *Typerep.Typerep*
| **type-constructor** *integer* ⇀ (*Haskell-Quickcheck*) *Prelude.Int*

code-reserved *Haskell-Quickcheck Typerep*

code-printing
constant *0*::*integer* ⇀
(*Haskell-Quickcheck*) !(*0*/ ::/ *Prelude.Int*)

⟨*ML*⟩

83.1.2 Narrowing's deep representation of types and terms

datatype (*plugins only*: *code extraction*) *narrowing-type* =
Narrowing-sum-of-products narrowing-type list list

datatype (*plugins only*: *code extraction*) *narrowing-term* =
Narrowing-variable integer list narrowing-type
| *Narrowing-constructor integer narrowing-term list*

datatype (*plugins only*: *code extraction*) (*dead* $'a$) *narrowing-cons* =
  Narrowing-cons narrowing-type (*narrowing-term list* $\Rightarrow$ $'a$) *list*

primrec *map-cons* :: ($'a$ => $'b$) => $'a$ *narrowing-cons* => $'b$ *narrowing-cons*
where
  map-cons f (*Narrowing-cons ty cs*) = *Narrowing-cons ty* (*map* ($\lambda c.$ *f o c*) *cs*)

83.1.3 From narrowing's deep representation of terms to *Code-Evaluation*'s terms

class *partial-term-of* = *typerep* +
  fixes *partial-term-of* :: $'a$ *itself* => *narrowing-term* => *Code-Evaluation.term*

lemma *partial-term-of-anything*: *partial-term-of x nt* $\equiv$ *t*
  ⟨*proof*⟩

83.1.4 Auxilary functions for Narrowing

consts *nth* :: $'a$ *list* => *integer* => $'a$

code-printing constant *nth* ⇀ (*Haskell-Quickcheck*) **infixl** *9* !!

consts *error* :: *char list* => $'a$

code-printing constant *error* ⇀ (*Haskell-Quickcheck*) *error*

consts *toEnum* :: *integer* => *char*

code-printing constant *toEnum* ⇀ (*Haskell-Quickcheck*) *Prelude.toEnum*

consts *marker* :: *char*

code-printing constant *marker* ⇀ (*Haskell-Quickcheck*) $'\backslash 0'$

83.1.5 Narrowing's basic operations

type-synonym $'a$ *narrowing* = *integer* => $'a$ *narrowing-cons*

definition *cons* :: $'a$ => $'a$ *narrowing*
where
  cons a d = (*Narrowing-cons* (*Narrowing-sum-of-products* [[]]) [(λ-. *a*)])

fun *conv* :: (*narrowing-term list* => $'a$) *list* => *narrowing-term* => $'a$
where
  conv cs (*Narrowing-variable p* -) = *error* (*marker* # *map toEnum p*)
| *conv cs* (*Narrowing-constructor i xs*) = (*nth cs i*) *xs*

fun *non-empty* :: *narrowing-type* => *bool*
where
  non-empty (*Narrowing-sum-of-products ps*) = ($\neg$ (*List.null ps*))

definition *apply* :: $('a => 'b)$ *narrowing* => *'a narrowing* => *'b narrowing*
where
apply f a d = (*if d* > *0 then*
(*case f d of Narrowing-cons* (*Narrowing-sum-of-products ps*) *cfs* ⇒
case a (*d* − *1*) *of Narrowing-cons ta cas* ⇒
let
shallow = *non-empty ta*;
cs = [(λ(*x* # *xs*) ⇒ *cf xs* (*conv cas x*)). *shallow*, *cf* ← *cfs*]
in Narrowing-cons (*Narrowing-sum-of-products* [*ta* # *p*. *shallow*, *p* ← *ps*]) *cs*)
else Narrowing-cons (*Narrowing-sum-of-products* []) [])

definition *sum* :: *'a narrowing* => *'a narrowing* => *'a narrowing*
where
sum a b d =
(*case a d of Narrowing-cons* (*Narrowing-sum-of-products ssa*) *ca* =>
case b d of Narrowing-cons (*Narrowing-sum-of-products ssb*) *cb* =>
Narrowing-cons (*Narrowing-sum-of-products* (*ssa* @ *ssb*)) (*ca* @ *cb*))

lemma [*fundef-cong*]:
assumes $a\ d = a'\ d\ b\ d = b'\ d\ d = d'$
shows $sum\ a\ b\ d = sum\ a'\ b'\ d'$
⟨*proof*⟩

lemma [*fundef-cong*]:
assumes $f\ d = f'\ d\ (\bigwedge d'.\ 0 \leq d' \wedge d' < d \Longrightarrow a\ d' = a'\ d')$
assumes $d = d'$
shows $apply\ f\ a\ d = apply\ f'\ a'\ d'$
⟨*proof*⟩

83.1.6 Narrowing generator type class

class *narrowing* =
fixes *narrowing* :: *integer* => *'a narrowing-cons*

datatype (*plugins only*: *code extraction*) *property* =
Universal narrowing-type (*narrowing-term* => *property*) *narrowing-term* => *Code-Evaluation.term*
| *Existential narrowing-type* (*narrowing-term* => *property*) *narrowing-term* => *Code-Evaluation.term*
| *Property bool*

definition *exists* :: (*'a* :: {*narrowing*, *partial-term-of*} => *property*) => *property*
where
exists f = (*case narrowing* (*100* :: *integer*) *of Narrowing-cons ty cs* => *Existential ty* (λ *t*. *f* (*conv cs t*)) (*partial-term-of* (*TYPE*(*'a*))))

definition *all* :: (*'a* :: {*narrowing*, *partial-term-of*} => *property*) => *property*
where
all f = (*case narrowing* (*100* :: *integer*) *of Narrowing-cons ty cs* => *Universal ty* (λ*t*. *f* (*conv cs t*)) (*partial-term-of* (*TYPE*(*'a*))))

83.1.7 class *is-testable*

The class *is-testable* ensures that all necessary type instances are generated.

class *is-testable*

instance *bool* :: *is-testable* ⟨*proof*⟩

instance *fun* :: ({*term-of*, *narrowing*, *partial-term-of*}, *is-testable*) *is-testable* ⟨*proof*⟩

definition *ensure-testable* :: *'a* :: *is-testable* => *'a* :: *is-testable*
where
ensure-testable f = *f*

83.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way

datatype (*plugins only*: *code quickcheck-narrowing extraction*) (*dead 'a*, *dead 'b*) *ffun* =
Constant 'b
| *Update 'a 'b* (*'a*, *'b*) *ffun*

primrec *eval-ffun* :: (*'a*, *'b*) *ffun* => *'a* => *'b*
where
eval-ffun (*Constant c*) *x* = *c*
| *eval-ffun* (*Update x' y f*) *x* = (*if x* = *x' then y else eval-ffun f x*)

hide-type (**open**) *ffun*
hide-const (**open**) *Constant Update eval-ffun*

datatype (*plugins only*: *code quickcheck-narrowing extraction*) (*dead 'b*) *cfun* = *Constant 'b*

primrec *eval-cfun* :: *'b cfun* => *'a* => *'b*
where
eval-cfun (*Constant c*) *y* = *c*

hide-type (**open**) *cfun*
hide-const (**open**) *Constant eval-cfun Abs-cfun Rep-cfun*

83.1.9 Setting up the counterexample generator

⟨*ML*⟩

definition *narrowing-dummy-partial-term-of* :: ('*a* :: *partial-term-of*) *itself* => *narrowing-term* => *term*
where
narrowing-dummy-partial-term-of = *partial-term-of*

definition *narrowing-dummy-narrowing* :: *integer* => ('*a* :: *narrowing*) *narrowing-cons*
where
narrowing-dummy-narrowing = *narrowing*

lemma [*code*]:
ensure-testable f =
(*let*
x = *narrowing-dummy-narrowing* :: *integer* => *bool narrowing-cons*;
y = *narrowing-dummy-partial-term-of* :: *bool itself* => *narrowing-term* => *term*;
z = (*conv* :: - => - => *unit*) *in f*)
⟨*proof*⟩

83.2 Narrowing for sets

instantiation *set* :: (*narrowing*) *narrowing*
begin

definition *narrowing-set* = *Quickcheck-Narrowing.apply* (*Quickcheck-Narrowing.cons set*) *narrowing*

instance ⟨*proof*⟩

end

83.3 Narrowing for integers

definition *drawn-from* :: '*a list* ⇒ '*a narrowing-cons*
where
drawn-from xs =
Narrowing-cons (*Narrowing-sum-of-products* (*map* (λ-. []) *xs*)) (*map* (λ*x* -. *x*) *xs*)

function *around-zero* :: *int* ⇒ *int list*
where
around-zero i = (*if i* < *0 then* [] *else* (*if i* = *0 then* [*0*] *else around-zero* (*i* − *1*) @ [*i*, −*i*]))
⟨*proof*⟩
termination ⟨*proof*⟩

declare *around-zero.simps* [*simp del*]

lemma *length-around-zero*:
assumes *i* >= *0*
shows *length* (*around-zero i*) = *2* ∗ *nat i* + *1*

⟨*proof*⟩

instantiation *int* :: *narrowing*
begin

definition
narrowing-int d = (*let* (*u* :: - ⇒ - ⇒ *unit*) = *conv*; *i* = *int-of-integer d*
in drawn-from (*around-zero i*))

instance ⟨*proof*⟩

end

declare [[*code drop*: *partial-term-of* :: *int itself* ⇒ -]]

lemma [*code*]:
partial-term-of (*ty* :: *int itself*) (*Narrowing-variable p t*) ≡
Code-Evaluation.Free (*STR ''-''*) (*Typerep.Typerep* (*STR ''Int.int''*) [])
partial-term-of (*ty* :: *int itself*) (*Narrowing-constructor i* []) ≡
(*if i mod 2* = *0*
then Code-Evaluation.term-of (− (*int-of-integer i*) *div 2*)
else Code-Evaluation.term-of ((*int-of-integer i* + *1*) *div 2*))
⟨*proof*⟩

instantiation *integer* :: *narrowing*
begin

definition
narrowing-integer d = (*let* (*u* :: - ⇒ - ⇒ *unit*) = *conv*; *i* = *int-of-integer d*
in drawn-from (*map integer-of-int* (*around-zero i*)))

instance ⟨*proof*⟩

end

declare [[*code drop*: *partial-term-of* :: *integer itself* ⇒ -]]

lemma [*code*]:
partial-term-of (*ty* :: *integer itself*) (*Narrowing-variable p t*) ≡
Code-Evaluation.Free (*STR ''-''*) (*Typerep.Typerep* (*STR ''Code-Numeral.integer''*)
[])
partial-term-of (*ty* :: *integer itself*) (*Narrowing-constructor i* []) ≡
(*if i mod 2* = *0*
then Code-Evaluation.term-of (− *i div 2*)
else Code-Evaluation.term-of ((*i* + *1*) *div 2*))
⟨*proof*⟩

code-printing constant *Code-Evaluation.term-of* :: *integer* ⇒ *term* ⇀ (*Haskell-Quickcheck*)

(*let* { t = *Typerep.Typerep Code'-Numeral.integer* [];
  mkFunT s t = *Typerep.Typerep fun* [s, t];
  numT = *Typerep.Typerep Num.num* [];
  mkBit 0 = *Generated'-Code.Const Num.num.Bit0* (*mkFunT numT numT*);
  mkBit 1 = *Generated'-Code.Const Num.num.Bit1* (*mkFunT numT numT*);
  mkNumeral 1 = *Generated'-Code.Const Num.num.One numT*;
  mkNumeral i = *let* { q = i '*Prelude.div*' *2*; r = i '*Prelude.mod*' *2* }
    in Generated'-Code.App (*mkBit r*) (*mkNumeral q*);
  mkNumber 0 = *Generated'-Code.Const Groups.zero'-class.zero t*;
  mkNumber 1 = *Generated'-Code.Const Groups.one'-class.one t*;
  mkNumber i = *if* $i > 0$ *then*
      Generated'-Code.App
        (*Generated'-Code.Const Num.numeral'-class.numeral*
          (*mkFunT numT t*))
        (*mkNumeral i*)
    else
      Generated'-Code.App
        (*Generated'-Code.Const Groups.uminus'-class.uminus* (*mkFunT t t*))
        (*mkNumber* (− *i*)); } *in mkNumber*)

83.4 The *find-unused-assms* command

⟨*ML*⟩

83.5 Closing up

hide-type *narrowing-type narrowing-term narrowing-cons property*
hide-const *map-cons nth error toEnum marker empty Narrowing-cons conv non-empty ensure-testable all exists drawn-from around-zero*
hide-const (**open**) *Narrowing-variable Narrowing-constructor apply sum cons*
hide-fact *empty-def cons-def conv.simps non-empty.simps apply-def sum-def ensure-testable-def all-def exists-def*

end

84 Program extraction for HOL

theory *Extraction*
imports *Option*
begin

⟨*ML*⟩

84.1 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 implies-True-equals TrueE
False-implies-equals implies-False-swap

lemmas [*extraction-expand-def*] =
HOL.induct-forall-def HOL.induct-implies-def HOL.induct-equal-def HOL.induct-conj-def
HOL.induct-true-def HOL.induct-false-def

datatype (*plugins only*: *code extraction*) *sumbool* = *Left* | *Right*

84.2 Type of extracted program

extract-type

$typeof\ (Trueprop\ P) \equiv typeof\ P$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('Q))$

$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE(Null))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('P \Rightarrow 'Q))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$\quad typeof\ (\forall x.\ P\ x) \equiv Type\ (TYPE(Null))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$\quad typeof\ (\forall x::'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$\quad typeof\ (\exists x::'a.\ P\ x) \equiv Type\ (TYPE('a))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$\quad typeof\ (\exists x::'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$

$typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

84.3 Realizability

realizability

$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv (\forall\, x::'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P::bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

84.4 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'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q$ ⟨*proof*⟩

realizers
$impI\ (P,\ Q)$: $\lambda pq.\ pq$
$\boldsymbol{\lambda}(c\text{: -})\ (d\text{: -})\ P\ Q\ pq\ (h\text{: -}).\ allI \cdot \cdot \bullet\ c \bullet (\boldsymbol{\lambda} x.\ impI \cdot \cdot \cdot \cdot \bullet (h \cdot x))$

$impI\ (P)$: $Null$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ (h\text{: -}).\ allI \cdot \cdot \bullet\ c \bullet (\boldsymbol{\lambda} x.\ impI \cdot \cdot \cdot \cdot \bullet (h \cdot x))$

$impI\ (Q)$: $\lambda q.\ q\ \boldsymbol{\lambda}(c\text{: -})\ P\ Q\ q.\ impI \cdot \cdot \cdot \cdot$

$impI$: $Null\ impI$

$mp\ (P,\ Q)$: $\lambda pq.\ pq$
$\boldsymbol{\lambda}(c\text{: -})\ (d\text{: -})\ P\ Q\ pq\ (h\text{: -})\ p.\ mp \cdot \cdot \cdot \cdot \bullet (spec \cdot \cdot \cdot\ p \bullet c \bullet h)$

$mp\ (P)$: $Null$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ (h\text{: -})\ p.\ mp \cdot \cdot \cdot \cdot \bullet (spec \cdot \cdot \cdot\ p \bullet c \bullet h)$

$mp\ (Q)$: $\lambda q.\ q\ \boldsymbol{\lambda}(c\text{: -})\ P\ Q\ q.\ mp \cdot \cdot \cdot \cdot$

mp: $Null\ mp$

$allI\ (P)$: $\lambda p.\ p\ \boldsymbol{\lambda}(c\text{: -})\ P\ (d\text{: -})\ p.\ allI \cdot \cdot \bullet\ d$

$allI$: $Null\ allI$

$spec\ (P)$: $\lambda x\ p.\ p\ x\ \boldsymbol{\lambda}(c\text{: -})\ P\ x\ (d\text{: -})\ p.\ spec \cdot \cdot \cdot\ x \bullet d$

$spec$: $Null\ spec$

$exI\ (P)$: $\lambda x\ p.\ (x,\ p)\ \boldsymbol{\lambda}(c\text{: -})\ P\ x\ (d\text{: -})\ p.\ exI\text{-}realizer \cdot P \cdot p \cdot x \bullet c \bullet d$

exI: $\lambda x.\ x\ \boldsymbol{\lambda} P\ x\ (c\text{: -})\ (h\text{: -}).\ h$

$exE\ (P,\ Q)$: $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
$\boldsymbol{\lambda}(c\text{: -})\ (d\text{: -})\ P\ Q\ (e\text{: -})\ p\ (h\text{: -})\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \bullet c \bullet e \bullet d \bullet h$

$exE\ (P)$: $Null$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ (d\text{: -})\ p.\ exE\text{-}realizer' \cdot \cdot \cdot \cdot \cdot \cdot \bullet\ c \bullet d$

exE (Q): $\lambda x\ pq.\ pq\ x$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ (d\text{: -})\ x\ (h1\text{: -})\ pq\ (h2\text{: -}).\ h2 \cdot x \bullet h1$

exE: *Null*
$\boldsymbol{\lambda}P\ Q\ (c\text{: -})\ x\ (h1\text{: -})\ (h2\text{: -}).\ h2 \cdot x \bullet h1$

conjI $(P,\ Q)$: *Pair*
$\boldsymbol{\lambda}(c\text{: -})\ (d\text{: -})\ P\ Q\ p\ (h\text{: -})\ q.\ conjI\text{-}realizer \cdot P \cdot p \cdot Q \cdot q \bullet c \bullet d \bullet h$

conjI (P): $\lambda p.\ p$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ p.\ conjI \cdot \text{-} \cdot \text{-}$

conjI (Q): $\lambda q.\ q$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ (h\text{: -})\ q.\ conjI \cdot \text{-} \cdot \text{-} \bullet h$

conjI: *Null conjI*

conjunct1 $(P,\ Q)$: *fst*
$\boldsymbol{\lambda}(c\text{: -})\ (d\text{: -})\ P\ Q\ pq.\ conjunct1 \cdot \text{-} \cdot \text{-}$

conjunct1 (P): $\lambda p.\ p$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ p.\ conjunct1 \cdot \text{-} \cdot \text{-}$

conjunct1 (Q): *Null*
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ q.\ conjunct1 \cdot \text{-} \cdot \text{-}$

conjunct1: *Null conjunct1*

conjunct2 $(P,\ Q)$: *snd*
$\boldsymbol{\lambda}(c\text{: -})\ (d\text{: -})\ P\ Q\ pq.\ conjunct2 \cdot \text{-} \cdot \text{-}$

conjunct2 (P): *Null*
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ p.\ conjunct2 \cdot \text{-} \cdot \text{-}$

conjunct2 (Q): $\lambda p.\ p$
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ p.\ conjunct2 \cdot \text{-} \cdot \text{-}$

conjunct2: *Null conjunct2*

disjI1 $(P,\ Q)$: *Inl*
$\boldsymbol{\lambda}(c\text{: -})\ (d\text{: -})\ P\ Q\ p.\ iffD2 \cdot \text{-} \cdot \text{-} \bullet (sum.case\text{-}1 \cdot P \cdot \text{-} \cdot p \bullet arity\text{-}type\text{-}bool \bullet c \bullet d)$

disjI1 (P): *Some*
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q\ p.\ iffD2 \cdot \text{-} \cdot \text{-} \bullet (option.case\text{-}2 \cdot \text{-} \cdot P \cdot p \bullet arity\text{-}type\text{-}bool \bullet c)$

disjI1 (Q): *None*
$\boldsymbol{\lambda}(c\text{: -})\ P\ Q.\ iffD2 \cdot \text{-} \cdot \text{-} \bullet (option.case\text{-}1 \cdot \text{-} \cdot \text{-} \bullet arity\text{-}type\text{-}bool \bullet c)$

disjI1: *Left*
$\boldsymbol{\lambda}P\ Q.\ iffD2 \cdot \cdot \cdot \cdot \bullet (sumbool.case\text{-}1 \cdot \cdot \cdot \cdot \bullet arity\text{-}type\text{-}bool)$

disjI2 (P, Q): *Inr*
$\boldsymbol{\lambda}(d{:}\ \text{-})\ (c{:}\ \text{-})\ Q\ P\ q.\ iffD2 \cdot \cdot \cdot \cdot \bullet (sum.case\text{-}2 \cdot \cdot \cdot Q \cdot q \bullet arity\text{-}type\text{-}bool \bullet c \bullet d)$

disjI2 (P): *None*
$\boldsymbol{\lambda}(c{:}\ \text{-})\ Q\ P.\ iffD2 \cdot \cdot \cdot \cdot \bullet (option.case\text{-}1 \cdot \cdot \cdot \cdot \bullet arity\text{-}type\text{-}bool \bullet c)$

disjI2 (Q): *Some*
$\boldsymbol{\lambda}(c{:}\ \text{-})\ Q\ P\ q.\ iffD2 \cdot \cdot \cdot \cdot \bullet (option.case\text{-}2 \cdot \cdot \cdot Q \cdot q \bullet arity\text{-}type\text{-}bool \bullet c)$

disjI2: *Right*
$\boldsymbol{\lambda}Q\ P.\ iffD2 \cdot \cdot \cdot \cdot \bullet (sumbool.case\text{-}2 \cdot \cdot \cdot \cdot \bullet arity\text{-}type\text{-}bool)$

disjE (P, Q, R): $\lambda pq\ pr\ qr.$
$(case\ pq\ of\ Inl\ p \Rightarrow pr\ p \mid Inr\ q \Rightarrow qr\ q)$
$\boldsymbol{\lambda}(c{:}\ \text{-})\ (d{:}\ \text{-})\ (e{:}\ \text{-})\ P\ Q\ R\ pq\ (h1{:}\ \text{-})\ pr\ (h2{:}\ \text{-})\ qr.$
$disjE\text{-}realizer \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \bullet c \bullet d \bullet e \bullet h1 \bullet h2$

disjE (Q, R): $\lambda pq\ pr\ qr.$
$(case\ pq\ of\ None \Rightarrow pr \mid Some\ q \Rightarrow qr\ q)$
$\boldsymbol{\lambda}(c{:}\ \text{-})\ (d{:}\ \text{-})\ P\ Q\ R\ pq\ (h1{:}\ \text{-})\ pr\ (h2{:}\ \text{-})\ qr.$
$disjE\text{-}realizer2 \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \bullet c \bullet d \bullet h1 \bullet h2$

disjE (P, R): $\lambda pq\ pr\ qr.$
$(case\ pq\ of\ None \Rightarrow qr \mid Some\ p \Rightarrow pr\ p)$
$\boldsymbol{\lambda}(c{:}\ \text{-})\ (d{:}\ \text{-})\ P\ Q\ R\ pq\ (h1{:}\ \text{-})\ pr\ (h2{:}\ \text{-})\ qr\ (h3{:}\ \text{-}).$
$disjE\text{-}realizer2 \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot qr \cdot pr \bullet c \bullet d \bullet h1 \bullet h3 \bullet h2$

disjE (R): $\lambda pq\ pr\ qr.$
$(case\ pq\ of\ Left \Rightarrow pr \mid Right \Rightarrow qr)$
$\boldsymbol{\lambda}(c{:}\ \text{-})\ P\ Q\ R\ pq\ (h1{:}\ \text{-})\ pr\ (h2{:}\ \text{-})\ qr.$
$disjE\text{-}realizer3 \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \bullet c \bullet h1 \bullet h2$

disjE (P, Q): *Null*
$\boldsymbol{\lambda}(c{:}\ \text{-})\ (d{:}\ \text{-})\ P\ Q\ R\ pq.\ disjE\text{-}realizer \cdot \cdot \cdot \cdot \cdot pq \cdot (\lambda x.\ R) \cdot \cdot \cdot \cdot \bullet c \bullet d \bullet arity\text{-}type\text{-}bool$

disjE (Q): *Null*
$\boldsymbol{\lambda}(c{:}\ \text{-})\ P\ Q\ R\ pq.\ disjE\text{-}realizer2 \cdot \cdot \cdot \cdot \cdot pq \cdot (\lambda x.\ R) \cdot \cdot \cdot \cdot \bullet c \bullet arity\text{-}type\text{-}bool$

disjE (P): *Null*
$\boldsymbol{\lambda}(c{:}\ \text{-})\ P\ Q\ R\ pq\ (h1{:}\ \text{-})\ (h2{:}\ \text{-})\ (h3{:}\ \text{-}).$
$disjE\text{-}realizer2 \cdot \cdot \cdot \cdot \cdot pq \cdot (\lambda x.\ R) \cdot \cdot \cdot \cdot \bullet c \bullet arity\text{-}type\text{-}bool \bullet h1 \bullet h3 \bullet h2$

disjE: *Null*

$\boldsymbol{\lambda}P\ Q\ R\ pq.\ disjE\text{-}realizer3 \cdot - \cdot - \cdot - \bullet\ pq \cdot (\lambda x.\ R) \cdot - \cdot - \cdot - \bullet\ arity\text{-}type\text{-}bool$

FalseE (P): *default*
$\boldsymbol{\lambda}(c\!: -)\ P.\ FalseE \cdot -$

FalseE: *Null FalseE*

notI (P): *Null*
$\boldsymbol{\lambda}(c\!: -)\ P\ (h\!: -).\ allI \cdot - \bullet\ c \bullet (\boldsymbol{\lambda}x.\ notI \cdot - \bullet (h \cdot x))$

notI: *Null notI*

notE (P, R): $\lambda p.$ *default*
$\boldsymbol{\lambda}(c\!: -)\ (d\!: -)\ P\ R\ (h\!: -)\ p.\ notE \cdot - \cdot - \bullet (spec \cdot - \cdot p \bullet c \bullet h)$

notE (P): *Null*
$\boldsymbol{\lambda}(c\!: -)\ P\ R\ (h\!: -)\ p.\ notE \cdot - \cdot - \bullet (spec \cdot - \cdot p \bullet c \bullet h)$

notE (R): *default*
$\boldsymbol{\lambda}(c\!: -)\ P\ R.\ notE \cdot - \cdot -$

notE: *Null notE*

subst (P): $\lambda s\ t\ ps.\ ps$
$\boldsymbol{\lambda}(c\!: -)\ s\ t\ P\ (d\!: -)\ (h\!: -)\ ps.\ subst \cdot s \cdot t \cdot P\ ps \bullet d \bullet h$

subst: *Null subst*

iffD1 (P, Q): *fst*
$\boldsymbol{\lambda}(d\!: -)\ (c\!: -)\ Q\ P\ pq\ (h\!: -)\ p.$
$\quad mp \cdot - \cdot - \bullet (spec \cdot - \cdot p \bullet d \bullet (conjunct1 \cdot - \cdot - \bullet h))$

iffD1 (P): $\lambda p.\ p$
$\boldsymbol{\lambda}(c\!: -)\ Q\ P\ p\ (h\!: -).\ mp \cdot - \cdot - \bullet (conjunct1 \cdot - \cdot - \bullet h)$

iffD1 (Q): *Null*
$\boldsymbol{\lambda}(c\!: -)\ Q\ P\ q1\ (h\!: -)\ q2.$
$\quad mp \cdot - \cdot - \bullet (spec \cdot - \cdot q2 \bullet c \bullet (conjunct1 \cdot - \cdot - \bullet h))$

iffD1: *Null iffD1*

iffD2 (P, Q): *snd*
$\boldsymbol{\lambda}(c\!: -)\ (d\!: -)\ P\ Q\ pq\ (h\!: -)\ q.$
$\quad mp \cdot - \cdot - \bullet (spec \cdot - \cdot q \bullet d \bullet (conjunct2 \cdot - \cdot - \bullet h))$

iffD2 (P): $\lambda p.\ p$
$\boldsymbol{\lambda}(c\!: -)\ P\ Q\ p\ (h\!: -).\ mp \cdot - \cdot - \bullet (conjunct2 \cdot - \cdot - \bullet h)$

iffD2 (Q): *Null*

$\boldsymbol{\lambda}$(*c*: -) *P Q q1* (*h*: -) *q2*.
  mp · - - - • (*spec* · - · *q2* • *c* • (*conjunct2* · - - - • *h*))

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
$\boldsymbol{\lambda}$(*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* • ($\boldsymbol{\lambda}$*x*. *impI* · - - - • (*h1* · *x*))) •
  (*allI* · - • *d* • ($\boldsymbol{\lambda}$*x*. *impI* · - - - • (*h2* · *x*)))

iffI (*P*): λp. *p*
$\boldsymbol{\lambda}$(*c*: -) *P Q* (*h1* : -) *p* (*h2* : -). *conjI* · - - - •
  (*allI* · - • *c* • ($\boldsymbol{\lambda}$*x*. *impI* · - - - • (*h1* · *x*))) •
  (*impI* · - - - • *h2*)

iffI (*Q*): λq. *q*
$\boldsymbol{\lambda}$(*c*: -) *P Q q* (*h1* : -) (*h2* : -). *conjI* · - - - •
  (*impI* · - - - • *h1*) •
  (*allI* · - • *c* • ($\boldsymbol{\lambda}$*x*. *impI* · - - - • (*h2* · *x*)))

iffI: *Null iffI*

end

85 Extensible records with structural subtyping

theory *Record*
imports *Quickcheck-Exhaustive*
keywords
  record :: *thy-decl* **and**
  print-record :: *diag*
begin

85.1 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 analogous 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 analogous 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.

85.2 Operators and lemmas for types isomorphic to tuples

datatype $(dead\ 'a,\ dead\ 'b,\ dead\ 'c)\ tuple\text{-}isomorphism =$
$\quad Tuple\text{-}Isomorphism\ 'a \Rightarrow 'b \times 'c\ 'b \times 'c \Rightarrow 'a$

primrec
$\quad repr :: ('a,\ 'b,\ 'c)\ tuple\text{-}isomorphism \Rightarrow 'a \Rightarrow 'b \times 'c$ **where**
$\quad repr\ (Tuple\text{-}Isomorphism\ r\ a) = r$

primrec
$abst :: ('a, 'b, 'c)\ tuple\text{-}isomorphism \Rightarrow 'b \times 'c \Rightarrow 'a$ **where**
$abst\ (Tuple\text{-}Isomorphism\ r\ a) = a$

definition
$iso\text{-}tuple\text{-}fst :: ('a, 'b, 'c)\ tuple\text{-}isomorphism \Rightarrow 'a \Rightarrow 'b$ **where**
$iso\text{-}tuple\text{-}fst\ isom = fst \circ repr\ isom$

definition
$iso\text{-}tuple\text{-}snd :: ('a, 'b, 'c)\ tuple\text{-}isomorphism \Rightarrow 'a \Rightarrow 'c$ **where**
$iso\text{-}tuple\text{-}snd\ isom = snd \circ repr\ isom$

definition
$iso\text{-}tuple\text{-}fst\text{-}update ::$
$('a, 'b, 'c)\ tuple\text{-}isomorphism \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)$ **where**
$iso\text{-}tuple\text{-}fst\text{-}update\ isom\ f = abst\ isom \circ apfst\ f \circ repr\ isom$

definition
$iso\text{-}tuple\text{-}snd\text{-}update ::$
$('a, 'b, 'c)\ tuple\text{-}isomorphism \Rightarrow ('c \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'a)$ **where**
$iso\text{-}tuple\text{-}snd\text{-}update\ isom\ f = abst\ isom \circ apsnd\ f \circ repr\ isom$

definition
$iso\text{-}tuple\text{-}cons ::$
$('a, 'b, 'c)\ tuple\text{-}isomorphism \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a$ **where**
$iso\text{-}tuple\text{-}cons\ isom = curry\ (abst\ isom)$

85.3 Logical infrastructure for records

definition
$iso\text{-}tuple\text{-}surjective\text{-}proof\text{-}assist :: 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$ **where**
$iso\text{-}tuple\text{-}surjective\text{-}proof\text{-}assist\ x\ y\ f \longleftrightarrow f\ x = y$

definition
$iso\text{-}tuple\text{-}update\text{-}accessor\text{-}cong\text{-}assist ::$
$(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$ **where**
$iso\text{-}tuple\text{-}update\text{-}accessor\text{-}cong\text{-}assist\ upd\ ac \longleftrightarrow$
$(\forall f\ v.\ upd\ (\lambda x.\ f\ (ac\ v))\ v = upd\ f\ v) \wedge (\forall v.\ upd\ id\ v = v)$

definition
$iso\text{-}tuple\text{-}update\text{-}accessor\text{-}eq\text{-}assist ::$
$(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where
$iso\text{-}tuple\text{-}update\text{-}accessor\text{-}eq\text{-}assist\ upd\ ac\ v\ f\ v'\ x \longleftrightarrow$
$upd\ f\ v = v' \wedge ac\ v = x \wedge iso\text{-}tuple\text{-}update\text{-}accessor\text{-}cong\text{-}assist\ upd\ ac$

lemma *update-accessor-congruence-foldE*:
assumes *uac*: $iso\text{-}tuple\text{-}update\text{-}accessor\text{-}cong\text{-}assist\ upd\ ac$

and r: $r = r'$ **and** v: $ac\ r' = v'$
and f: $\bigwedge v.\ v' = v \Longrightarrow f\ v = f'\ v$
shows $upd\ f\ r = upd\ f'\ r'$
$\langle proof \rangle$

lemma *update-accessor-congruence-unfoldE*:
iso-tuple-update-accessor-cong-assist upd ac $\Longrightarrow$
$r = r' \Longrightarrow ac\ r' = v' \Longrightarrow (\bigwedge v.\ v = v' \Longrightarrow f\ v = f'\ v) \Longrightarrow$
$upd\ f\ r = upd\ f'\ r'$
$\langle proof \rangle$

lemma *iso-tuple-update-accessor-cong-assist-id*:
iso-tuple-update-accessor-cong-assist upd ac $\Longrightarrow upd\ id = id$
$\langle proof \rangle$

lemma *update-accessor-noopE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd ac*
and ac: $f\ (ac\ x) = ac\ x$
shows $upd\ f\ x = x$
$\langle proof \rangle$

lemma *update-accessor-noop-compE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd ac*
and ac: $f\ (ac\ x) = ac\ x$
shows $upd\ (g \circ f)\ x = upd\ g\ x$
$\langle proof \rangle$

lemma *update-accessor-cong-assist-idI*:
iso-tuple-update-accessor-cong-assist id id
$\langle proof \rangle$

lemma *update-accessor-cong-assist-triv*:
iso-tuple-update-accessor-cong-assist upd ac $\Longrightarrow$
iso-tuple-update-accessor-cong-assist upd ac
$\langle proof \rangle$

lemma *update-accessor-accessor-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\Longrightarrow ac\ v = x$
$\langle proof \rangle$

lemma *update-accessor-updator-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\Longrightarrow upd\ f\ v = v'$
$\langle proof \rangle$

lemma *iso-tuple-update-accessor-eq-assist-idI*:
$v' = f\ v \Longrightarrow$ *iso-tuple-update-accessor-eq-assist id id v f v' v*
$\langle proof \rangle$

lemma *iso-tuple-update-accessor-eq-assist-triv*:

iso-tuple-update-accessor-eq-assist upd ac v f v′ x $\Longrightarrow$
iso-tuple-update-accessor-eq-assist upd ac v f v′ x
⟨*proof*⟩

lemma *iso-tuple-update-accessor-cong-from-eq*:
iso-tuple-update-accessor-eq-assist upd ac v f v′ x $\Longrightarrow$
iso-tuple-update-accessor-cong-assist upd ac
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assistI*:
f x = *y* $\Longrightarrow$ *iso-tuple-surjective-proof-assist x y f*
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assist-idE*:
iso-tuple-surjective-proof-assist x y id $\Longrightarrow$ *x* = *y*
⟨*proof*⟩

locale *isomorphic-tuple* =
fixes *isom* :: (′*a*, ′*b*, ′*c*) *tuple-isomorphism*
assumes *repr-inv*: $\bigwedge$*x*. *abst isom* (*repr isom x*) = *x*
and *abst-inv*: $\bigwedge$*y*. *repr isom* (*abst isom y*) = *y*
begin

lemma *repr-inj*: *repr isom x* = *repr isom y* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

lemma *abst-inj*: *abst isom x* = *abst isom y* $\longleftrightarrow$ *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* $\Longrightarrow$
(*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* $\Longrightarrow$
(*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*:

$(f\ o\ \textit{iso-tuple-snd isom})\ o\ (\textit{iso-tuple-fst-update isom}\ o\ h)\ g =$
$\quad id\ o\ (f\ o\ \textit{iso-tuple-snd isom})$
⟨*proof*⟩

lemma *iso-tuple-update-swap-fst-fst*:
$h\ f\ o\ j\ g = j\ g\ o\ h\ f \Longrightarrow$
$\quad (\textit{iso-tuple-fst-update isom}\ o\ h)\ f\ o\ (\textit{iso-tuple-fst-update isom}\ o\ j)\ g =$
$\quad\quad (\textit{iso-tuple-fst-update isom}\ o\ j)\ g\ o\ (\textit{iso-tuple-fst-update isom}\ o\ h)\ f$
⟨*proof*⟩

lemma *iso-tuple-update-swap-snd-snd*:
$h\ f\ o\ j\ g = j\ g\ o\ h\ f \Longrightarrow$
$\quad (\textit{iso-tuple-snd-update isom}\ o\ h)\ f\ o\ (\textit{iso-tuple-snd-update isom}\ o\ j)\ g =$
$\quad\quad (\textit{iso-tuple-snd-update isom}\ o\ j)\ g\ o\ (\textit{iso-tuple-snd-update isom}\ o\ h)\ f$
⟨*proof*⟩

lemma *iso-tuple-update-swap-fst-snd*:
$(\textit{iso-tuple-snd-update isom}\ o\ h)\ f\ o\ (\textit{iso-tuple-fst-update isom}\ o\ j)\ g =$
$\quad (\textit{iso-tuple-fst-update isom}\ o\ j)\ g\ o\ (\textit{iso-tuple-snd-update isom}\ o\ h)\ f$
⟨*proof*⟩

lemma *iso-tuple-update-swap-snd-fst*:
$(\textit{iso-tuple-fst-update isom}\ o\ h)\ f\ o\ (\textit{iso-tuple-snd-update isom}\ o\ j)\ g =$
$\quad (\textit{iso-tuple-snd-update isom}\ o\ j)\ g\ o\ (\textit{iso-tuple-fst-update isom}\ o\ h)\ f$
⟨*proof*⟩

lemma *iso-tuple-update-compose-fst-fst*:
$h\ f\ o\ j\ g = k\ (f\ o\ g) \Longrightarrow$
$\quad (\textit{iso-tuple-fst-update isom}\ o\ h)\ f\ o\ (\textit{iso-tuple-fst-update isom}\ o\ j)\ g =$
$\quad\quad (\textit{iso-tuple-fst-update isom}\ o\ k)\ (f\ o\ g)$
⟨*proof*⟩

lemma *iso-tuple-update-compose-snd-snd*:
$h\ f\ o\ j\ g = k\ (f\ o\ g) \Longrightarrow$
$\quad (\textit{iso-tuple-snd-update isom}\ o\ h)\ f\ o\ (\textit{iso-tuple-snd-update isom}\ o\ j)\ g =$
$\quad\quad (\textit{iso-tuple-snd-update isom}\ o\ k)\ (f\ o\ g)$
⟨*proof*⟩

lemma *iso-tuple-surjective-proof-assist-step*:
$\textit{iso-tuple-surjective-proof-assist}\ v\ a\ (\textit{iso-tuple-fst isom}\ o\ f) \Longrightarrow$
$\quad \textit{iso-tuple-surjective-proof-assist}\ v\ b\ (\textit{iso-tuple-snd isom}\ o\ f) \Longrightarrow$
$\quad \textit{iso-tuple-surjective-proof-assist}\ v\ (\textit{iso-tuple-cons isom}\ a\ b)\ f$
⟨*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*
$\quad (\textit{iso-tuple-fst-update isom}\ o\ f)\ (g\ o\ \textit{iso-tuple-fst isom})$
⟨*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*
$(iso\text{-}tuple\text{-}snd\text{-}update\ isom\ o\ f)\ (g\ o\ iso\text{-}tuple\text{-}snd\ isom)$
⟨*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*
$(iso\text{-}tuple\text{-}fst\text{-}update\ isom\ o\ f)\ (g\ o\ iso\text{-}tuple\text{-}fst\ isom)$
$(iso\text{-}tuple\text{-}cons\ isom\ a\ b)\ u\ (iso\text{-}tuple\text{-}cons\ isom\ a'\ b)\ v$
⟨*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*
$(iso\text{-}tuple\text{-}snd\text{-}update\ isom\ o\ f)\ (g\ o\ iso\text{-}tuple\text{-}snd\ isom)$
$(iso\text{-}tuple\text{-}cons\ isom\ a\ b)\ u\ (iso\text{-}tuple\text{-}cons\ isom\ a\ b')\ v$
⟨*proof*⟩

lemma *iso-tuple-cons-conj-eqI*:
$a = c \wedge b = d \wedge P \longleftrightarrow Q \Longrightarrow$
$iso\text{-}tuple\text{-}cons\ isom\ a\ b = iso\text{-}tuple\text{-}cons\ isom\ c\ d \wedge P \longleftrightarrow Q$
⟨*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*: $\bigwedge x\ y.\ repr\ x = repr\ y \longleftrightarrow x = y$

  and *abst-inv*: $\bigwedge z.\ repr\ (abst\ z) = z$
  and *v*: $v \equiv$ *Tuple-Isomorphism repr abst*
 shows *isomorphic-tuple v*
⟨*proof*⟩

definition
 tuple-iso-tuple $\equiv$ *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*⟩

85.4 Concrete record syntax

nonterminal
 ident **and**
 field-type **and**
 field-types **and**
 field **and**
 fields **and**
 field-update **and**
 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 (*ASCII*)

```
  -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)
```

85.5 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

86 Greatest common divisor and least common multiple

theory *GCD*
 imports *Groups-List*
begin

86.1 Abstract bounded quasi semilattices as common foundation

locale *bounded-quasi-semilattice* = *abel-semigroup* +
 fixes *top* :: $'a$ ($\top$) **and** *bot* :: $'a$ ($\bot$)
  and *normalize* :: $'a \Rightarrow 'a$
 assumes *idem-normalize* [*simp*]: $a * a = normalize\ a$
  and *normalize-left-idem* [*simp*]: $normalize\ a * b = a * b$
  and *normalize-idem* [*simp*]: $normalize\ (a * b) = a * b$
  and *normalize-top* [*simp*]: $normalize\ \top = \top$
  and *normalize-bottom* [*simp*]: $normalize\ \bot = \bot$
  and *top-left-normalize* [*simp*]: $\top * a = normalize\ a$
  and *bottom-left-bottom* [*simp*]: $\bot * a = \bot$

begin

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

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

lemma *comp-fun-idem*: *comp-fun-idem f*
⟨*proof*⟩

interpretation *comp-fun-idem f*
⟨*proof*⟩

lemma *top-right-normalize* [*simp*]:
$a * \top = normalize\ a$
⟨*proof*⟩

lemma *bottom-right-bottom* [*simp*]:
$a * \bot = \bot$
⟨*proof*⟩

lemma *normalize-right-idem* [*simp*]:
$a * normalize\ b = a * b$
⟨*proof*⟩

end

locale *bounded-quasi-semilattice-set* = *bounded-quasi-semilattice*
begin

interpretation *comp-fun-idem f*
⟨*proof*⟩

definition F :: *'a set* ⇒ *'a*
where
eq-fold: $F\ A = (if\ finite\ A\ then\ Finite\text{-}Set.fold\ f\ \top\ A\ else\ \bot)$

lemma *infinite* [*simp*]:
$infinite\ A \Longrightarrow F\ A = \bot$
⟨*proof*⟩

lemma *set-eq-fold* [*code*]:
$F\ (set\ xs) = fold\ f\ xs\ \top$
⟨*proof*⟩

lemma *empty* [*simp*]:

$F \{\} = \top$
⟨*proof*⟩

lemma *insert* [*simp*]:
F (*insert* a A) $= a * F\ A$
⟨*proof*⟩

lemma *normalize* [*simp*]:
normalize $(F\ A) = F\ A$
⟨*proof*⟩

lemma *in-idem*:
assumes $a \in A$
shows $a * F\ A = F\ A$
⟨*proof*⟩

lemma *union*:
$F\ (A \cup B) = F\ A * F\ B$
⟨*proof*⟩

lemma *remove*:
assumes $a \in A$
shows $F\ A = a * F\ (A - \{a\})$
⟨*proof*⟩

lemma *insert-remove*:
F (*insert* a A) $= a * F\ (A - \{a\})$
⟨*proof*⟩

lemma *subset*:
assumes $B \subseteq A$
shows $F\ B * F\ A = F\ A$
⟨*proof*⟩

end

86.2 Abstract GCD and LCM

class *gcd* = *zero* + *one* + *dvd* +
fixes *gcd* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *lcm* :: $'a \Rightarrow 'a \Rightarrow 'a$
begin

abbreviation *coprime* :: $'a \Rightarrow 'a \Rightarrow$ *bool*
where *coprime* $x\ y \equiv$ *gcd* $x\ y = 1$

end

class *Gcd* = *gcd* +

fixes $Gcd :: 'a\ set \Rightarrow 'a$
and $Lcm :: 'a\ set \Rightarrow 'a$
begin

abbreviation $GREATEST\text{-}COMMON\text{-}DIVISOR :: 'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$
where $GREATEST\text{-}COMMON\text{-}DIVISOR\ A\ f \equiv Gcd\ (f\ `\ A)$

abbreviation $LEAST\text{-}COMMON\text{-}MULTIPLE :: 'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$
where $LEAST\text{-}COMMON\text{-}MULTIPLE\ A\ f \equiv Lcm\ (f\ `\ A)$

end

syntax
-GCD1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3GCD -./ -*) [*0*, *10*] *10*)
-GCD :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3GCD -∈-./ -*) [*0*, *0*, *10*] *10*)
-LCM1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3LCM -./ -*) [*0*, *10*] *10*)
-LCM :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3LCM -∈-./ -*) [*0*, *0*, *10*] *10*)
translations
GCD x y. B ⇌ *GCD x. GCD y. B*
GCD x. B ⇌ *CONST GREATEST-COMMON-DIVISOR CONST UNIV* (*λx. B*)
GCD x. B ⇌ *GCD x ∈ CONST UNIV. B*
GCD x∈A. B ⇌ *CONST GREATEST-COMMON-DIVISOR A* (*λx. B*)
LCM x y. B ⇌ *LCM x. LCM y. B*
LCM x. B ⇌ *CONST LEAST-COMMON-MULTIPLE CONST UNIV* (*λx. B*)
LCM x. B ⇌ *LCM x ∈ CONST UNIV. B*
LCM x∈A. B ⇌ *CONST LEAST-COMMON-MULTIPLE A* (*λx. B*)

⟨*ML*⟩

class *semiring-gcd* = *normalization-semidom* + *gcd* +
assumes *gcd-dvd1* [*iff*]: *gcd a b dvd a*
and *gcd-dvd2* [*iff*]: *gcd a b dvd b*
and *gcd-greatest*: *c dvd a* ⟹ *c dvd b* ⟹ *c dvd gcd a b*
and *normalize-gcd* [*simp*]: *normalize* (*gcd a b*) = *gcd a b*
and *lcm-gcd*: *lcm a b* = *normalize* (*a* ∗ *b*) *div gcd a b*
begin

lemma *gcd-greatest-iff* [*simp*]: *a dvd gcd b c* ⟷ *a dvd b* ∧ *a dvd c*
⟨*proof*⟩

lemma *gcd-dvdI1*: *a dvd c* ⟹ *gcd a b dvd c*
⟨*proof*⟩

lemma *gcd-dvdI2*: *b dvd c* ⟹ *gcd a b dvd c*
⟨*proof*⟩

lemma *dvd-gcdD1*: *a dvd gcd b c* ⟹ *a dvd b*

⟨*proof*⟩

lemma *dvd-gcdD2*: $a\ dvd\ gcd\ b\ c \Longrightarrow a\ dvd\ c$
⟨*proof*⟩

lemma *gcd-0-left* [*simp*]: $gcd\ 0\ a = normalize\ a$
⟨*proof*⟩

lemma *gcd-0-right* [*simp*]: $gcd\ a\ 0 = normalize\ a$
⟨*proof*⟩

lemma *gcd-eq-0-iff* [*simp*]: $gcd\ a\ b = 0 \longleftrightarrow a = 0 \wedge b = 0$
(**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *unit-factor-gcd*: $unit\text{-}factor\ (gcd\ a\ b) = (if\ a = 0 \wedge b = 0\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *is-unit-gcd* [*simp*]: $is\text{-}unit\ (gcd\ a\ b) \longleftrightarrow coprime\ a\ b$
⟨*proof*⟩

sublocale *gcd*: *abel-semigroup gcd*
⟨*proof*⟩

sublocale *gcd*: *bounded-quasi-semilattice gcd 0 1 normalize*
⟨*proof*⟩

lemma *gcd-self*: $gcd\ a\ a = normalize\ a$
⟨*proof*⟩

lemma *gcd-left-idem*: $gcd\ a\ (gcd\ a\ b) = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-right-idem*: $gcd\ (gcd\ a\ b)\ b = gcd\ a\ b$
⟨*proof*⟩

lemma *coprime-1-left*: $coprime\ 1\ a$
⟨*proof*⟩

lemma *coprime-1-right*: $coprime\ a\ 1$
⟨*proof*⟩

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

lemma *gcd-mult-right*: $gcd\ (a * c)\ (b * c) = gcd\ b\ a * normalize\ c$
⟨*proof*⟩

lemma *mult-gcd-left*: $c * gcd\ a\ b = unit\text{-}factor\ c * gcd\ (c * a)\ (c * b)$

⟨proof⟩

lemma *mult-gcd-right*: *gcd a b* ∗ *c* = *gcd* (*a* ∗ *c*) (*b* ∗ *c*) ∗ *unit-factor c*
⟨proof⟩

lemma *dvd-lcm1* [*iff*]: *a dvd lcm a b*
⟨proof⟩

lemma *dvd-lcm2* [*iff*]: *b dvd lcm a b*
⟨proof⟩

lemma *dvd-lcmI1*: *a dvd b* ⟹ *a dvd lcm b c*
⟨proof⟩

lemma *dvd-lcmI2*: *a dvd c* ⟹ *a dvd lcm b c*
⟨proof⟩

lemma *lcm-dvdD1*: *lcm a b dvd c* ⟹ *a dvd c*
⟨proof⟩

lemma *lcm-dvdD2*: *lcm a b dvd c* ⟹ *b dvd c*
⟨proof⟩

lemma *lcm-least*:
assumes *a dvd c* **and** *b dvd c*
shows *lcm a b dvd c*
⟨proof⟩

lemma *lcm-least-iff* [*simp*]: *lcm a b dvd c* ⟷ *a dvd c* ∧ *b dvd c*
⟨proof⟩

lemma *normalize-lcm* [*simp*]: *normalize* (*lcm a b*) = *lcm a b*
⟨proof⟩

lemma *lcm-0-left* [*simp*]: *lcm 0 a* = *0*
⟨proof⟩

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

lemma *lcm-eq-0-iff*: *lcm a b* = *0* ⟷ *a* = *0* ∨ *b* = *0*
(**is** *?P* ⟷ *?Q*)
⟨proof⟩

lemma *lcm-eq-1-iff* [*simp*]: *lcm a b* = *1* ⟷ *is-unit a* ∧ *is-unit b*
⟨proof⟩

lemma *unit-factor-lcm*: *unit-factor* (*lcm a b*) = (*if a* = *0* ∨ *b* = *0 then 0 else 1*)
⟨proof⟩

sublocale *lcm*: *abel-semigroup lcm*
⟨*proof*⟩

sublocale *lcm*: *bounded-quasi-semilattice lcm 1 0 normalize*
⟨*proof*⟩

lemma *lcm-self*: *lcm a a* = *normalize a*
⟨*proof*⟩

lemma *lcm-left-idem*: *lcm a* (*lcm a b*) = *lcm a b*
⟨*proof*⟩

lemma *lcm-right-idem*: *lcm* (*lcm a b*) *b* = *lcm a b*
⟨*proof*⟩

lemma *gcd-mult-lcm* [*simp*]: *gcd a b* ∗ *lcm a b* = *normalize a* ∗ *normalize b*
⟨*proof*⟩

lemma *lcm-mult-gcd* [*simp*]: *lcm a b* ∗ *gcd a b* = *normalize a* ∗ *normalize b*
⟨*proof*⟩

lemma *gcd-lcm*:
assumes *a* ≠ *0* **and** *b* ≠ *0*
shows *gcd a b* = *normalize* (*a* ∗ *b*) *div lcm a b*
⟨*proof*⟩

lemma *lcm-1-left*: *lcm 1 a* = *normalize a*
⟨*proof*⟩

lemma *lcm-1-right*: *lcm a 1* = *normalize a*
⟨*proof*⟩

lemma *lcm-mult-left*: *lcm* (*c* ∗ *a*) (*c* ∗ *b*) = *normalize c* ∗ *lcm a b*
⟨*proof*⟩

lemma *lcm-mult-right*: *lcm* (*a* ∗ *c*) (*b* ∗ *c*) = *lcm b a* ∗ *normalize c*
⟨*proof*⟩

lemma *mult-lcm-left*: *c* ∗ *lcm a b* = *unit-factor c* ∗ *lcm* (*c* ∗ *a*) (*c* ∗ *b*)
⟨*proof*⟩

lemma *mult-lcm-right*: *lcm a b* ∗ *c* = *lcm* (*a* ∗ *c*) (*b* ∗ *c*) ∗ *unit-factor c*
⟨*proof*⟩

lemma *gcdI*:
assumes *c dvd a* **and** *c dvd b*
and *greatest*: ⋀*d*. *d dvd a* ⟹ *d dvd b* ⟹ *d dvd c*
and *normalize c* = *c*

shows $c = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-unique*:
$d\ dvd\ a \wedge d\ dvd\ b \wedge normalize\ d = d \wedge (\forall e.\ e\ dvd\ a \wedge e\ dvd\ b \longrightarrow e\ dvd\ d)$
$\longleftrightarrow d = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-dvd-prod*: $gcd\ a\ b\ dvd\ k * b$
⟨*proof*⟩

lemma *gcd-proj2-if-dvd*: $b\ dvd\ a \Longrightarrow gcd\ a\ b = normalize\ b$
⟨*proof*⟩

lemma *gcd-proj1-if-dvd*: $a\ dvd\ b \Longrightarrow gcd\ a\ b = normalize\ a$
⟨*proof*⟩

lemma *gcd-proj1-iff*: $gcd\ m\ n = normalize\ m \longleftrightarrow m\ dvd\ n$
⟨*proof*⟩

lemma *gcd-proj2-iff*: $gcd\ m\ n = normalize\ n \longleftrightarrow n\ dvd\ m$
⟨*proof*⟩

lemma *gcd-mult-distrib'*: $normalize\ c * gcd\ a\ b = gcd\ (c * a)\ (c * b)$
⟨*proof*⟩

lemma *gcd-mult-distrib*: $k * gcd\ a\ b = gcd\ (k * a)\ (k * b) * unit\text{-}factor\ k$
⟨*proof*⟩

lemma *lcm-mult-unit1*: $is\text{-}unit\ a \Longrightarrow lcm\ (b * a)\ c = lcm\ b\ c$
⟨*proof*⟩

lemma *lcm-mult-unit2*: $is\text{-}unit\ a \Longrightarrow lcm\ b\ (c * a) = lcm\ b\ c$
⟨*proof*⟩

lemma *lcm-div-unit1*:
$is\text{-}unit\ a \Longrightarrow lcm\ (b\ div\ a)\ c = lcm\ b\ c$
⟨*proof*⟩

lemma *lcm-div-unit2*: $is\text{-}unit\ a \Longrightarrow lcm\ b\ (c\ div\ a) = lcm\ b\ c$
⟨*proof*⟩

lemma *normalize-lcm-left*: $lcm\ (normalize\ a)\ b = lcm\ a\ b$
⟨*proof*⟩

lemma *normalize-lcm-right*: $lcm\ a\ (normalize\ b) = lcm\ a\ b$
⟨*proof*⟩

lemma *gcd-mult-unit1*: $is\text{-}unit\ a \Longrightarrow gcd\ (b * a)\ c = gcd\ b\ c$

⟨proof⟩

lemma *gcd-mult-unit2*: *is-unit a* $\Longrightarrow$ *gcd b* $(c * a)$ = *gcd b c*
⟨proof⟩

lemma *gcd-div-unit1*: *is-unit a* $\Longrightarrow$ *gcd* $(b$ *div* $a)$ c = *gcd b c*
⟨proof⟩

lemma *gcd-div-unit2*: *is-unit a* $\Longrightarrow$ *gcd b* $(c$ *div* $a)$ = *gcd b c*
⟨proof⟩

lemma *normalize-gcd-left*: *gcd* (*normalize a*) *b* = *gcd a b*
⟨proof⟩

lemma *normalize-gcd-right*: *gcd a* (*normalize b*) = *gcd a b*
⟨proof⟩

lemma *comp-fun-idem-gcd*: *comp-fun-idem gcd*
⟨proof⟩

lemma *comp-fun-idem-lcm*: *comp-fun-idem lcm*
⟨proof⟩

lemma *gcd-dvd-antisym*: *gcd a b dvd gcd c d* $\Longrightarrow$ *gcd c d dvd gcd a b* $\Longrightarrow$ *gcd a b* = *gcd c d*
⟨proof⟩

lemma *coprime-dvd-mult*:
assumes *coprime a b* **and** *a dvd c* $*$ *b*
shows *a dvd c*
⟨proof⟩

lemma *coprime-dvd-mult-iff*: *coprime a c* $\Longrightarrow$ *a dvd b* $*$ *c* $\longleftrightarrow$ *a dvd b*
⟨proof⟩

lemma *gcd-mult-cancel*: *coprime c b* $\Longrightarrow$ *gcd* $(c * a)$ b = *gcd a b*
⟨proof⟩

lemma *coprime-crossproduct*:
fixes *a b c d* :: *'a*
assumes *coprime a d* **and** *coprime b c*
shows *normalize a* $*$ *normalize c* = *normalize b* $*$ *normalize d* $\longleftrightarrow$
normalize a = *normalize b* $\wedge$ *normalize c* = *normalize d*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨proof⟩

lemma *gcd-add1* [*simp*]: *gcd* $(m + n)$ n = *gcd m n*
⟨proof⟩

lemma *gcd-add2* [*simp*]: $gcd\ m\ (m + n) = gcd\ m\ n$
⟨*proof*⟩

lemma *gcd-add-mult*: $gcd\ m\ (k * m + n) = gcd\ m\ n$
⟨*proof*⟩

lemma *coprimeI*: $(\bigwedge l.\ l\ dvd\ a \Longrightarrow l\ dvd\ b \Longrightarrow l\ dvd\ 1) \Longrightarrow gcd\ a\ b = 1$
⟨*proof*⟩

lemma *coprime*: $gcd\ a\ b = 1 \longleftrightarrow (\forall\, d.\ d\ dvd\ a \wedge d\ dvd\ b \longleftrightarrow is\text{-}unit\ d)$
⟨*proof*⟩

lemma *div-gcd-coprime*:
assumes *nz*: $a \neq 0 \vee b \neq 0$
shows $coprime\ (a\ div\ gcd\ a\ b)\ (b\ div\ gcd\ a\ b)$
⟨*proof*⟩

lemma *divides-mult*:
assumes $a\ dvd\ c$ **and** *nr*: $b\ dvd\ c$ **and** $coprime\ a\ b$
shows $a * b\ dvd\ c$
⟨*proof*⟩

lemma *coprime-lmult*:
assumes *dab*: $gcd\ d\ (a * b) = 1$
shows $gcd\ d\ a = 1$
⟨*proof*⟩

lemma *coprime-rmult*:
assumes *dab*: $gcd\ d\ (a * b) = 1$
shows $gcd\ d\ b = 1$
⟨*proof*⟩

lemma *coprime-mult*:
assumes $coprime\ d\ a$
and $coprime\ d\ b$
shows $coprime\ d\ (a * b)$
⟨*proof*⟩

lemma *coprime-mul-eq*: $gcd\ d\ (a * b) = 1 \longleftrightarrow gcd\ d\ a = 1 \wedge gcd\ d\ b = 1$
⟨*proof*⟩

lemma *coprime-mul-eq'*:
$coprime\ (a * b)\ d \longleftrightarrow coprime\ a\ d \wedge coprime\ b\ d$
⟨*proof*⟩

lemma *gcd-coprime*:
assumes *c*: $gcd\ a\ b \neq 0$
and *a*: $a = a' * gcd\ a\ b$
and *b*: $b = b' * gcd\ a\ b$

shows $gcd\ a'\ b' = 1$
$\langle proof \rangle$

lemma *coprime-power*:
assumes $0 < n$
shows $gcd\ a\ (b\ \hat{}\ n) = 1 \longleftrightarrow gcd\ a\ b = 1$
$\langle proof \rangle$

lemma *gcd-coprime-exists*:
assumes $gcd\ a\ b \neq 0$
shows $\exists\, a'\ b'.\ a = a' * gcd\ a\ b \wedge b = b' * gcd\ a\ b \wedge gcd\ a'\ b' = 1$
$\langle proof \rangle$

lemma *coprime-exp*: $gcd\ d\ a = 1 \Longrightarrow gcd\ d\ (a\hat{}n) = 1$
$\langle proof \rangle$

lemma *coprime-exp-left*: $coprime\ a\ b \Longrightarrow coprime\ (a\ \hat{}\ n)\ b$
$\langle proof \rangle$

lemma *coprime-exp2*:
assumes $coprime\ a\ b$
shows $coprime\ (a\ \hat{}\ n)\ (b\ \hat{}\ m)$
$\langle proof \rangle$

lemma *gcd-exp*: $gcd\ (a\ \hat{}\ n)\ (b\ \hat{}\ n) = gcd\ a\ b\ \hat{}\ n$
$\langle proof \rangle$

lemma *coprime-common-divisor*: $gcd\ a\ b = 1 \Longrightarrow a\ dvd\ a \Longrightarrow a\ dvd\ b \Longrightarrow is\text{-}unit\ a$
$\langle proof \rangle$

lemma *division-decomp*:
assumes $a\ dvd\ b * c$
shows $\exists\, b'\ c'.\ a = b' * c' \wedge b'\ dvd\ b \wedge c'\ dvd\ c$
$\langle proof \rangle$

lemma *pow-divs-pow*:
assumes *ab*: $a\ \hat{}\ n\ dvd\ b\ \hat{}\ n$ **and** *n*: $n \neq 0$
shows $a\ dvd\ b$
$\langle proof \rangle$

lemma *pow-divs-eq* [*simp*]: $n \neq 0 \Longrightarrow a\ \hat{}\ n\ dvd\ b\ \hat{}\ n \longleftrightarrow a\ dvd\ b$
$\langle proof \rangle$

lemma *coprime-plus-one* [*simp*]: $gcd\ (n + 1)\ n = 1$
$\langle proof \rangle$

lemma *prod-coprime* [*rule-format*]: $(\forall\, i{\in}A.\ gcd\ (f\ i)\ a = 1) \longrightarrow gcd\ (\prod i{\in}A.\ f\ i)\ a = 1$

⟨*proof*⟩

lemma *prod-list-coprime*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow coprime\ x\ y) \Longrightarrow coprime\ (prod\text{-}list\ xs)\ y$
⟨*proof*⟩

lemma *coprime-divisors*:
assumes $d\ dvd\ a\ e\ dvd\ b\ gcd\ a\ b = 1$
shows $gcd\ d\ e = 1$
⟨*proof*⟩

lemma *lcm-gcd-prod*: $lcm\ a\ b * gcd\ a\ b = normalize\ (a * b)$
⟨*proof*⟩

declare *unit-factor-lcm* [*simp*]

lemma *lcmI*:
assumes $a\ dvd\ c$ **and** $b\ dvd\ c$ **and** $\bigwedge d.\ a\ dvd\ d \Longrightarrow b\ dvd\ d \Longrightarrow c\ dvd\ d$
and $normalize\ c = c$
shows $c = lcm\ a\ b$
⟨*proof*⟩

lemma *gcd-dvd-lcm* [*simp*]: $gcd\ a\ b\ dvd\ lcm\ a\ b$
⟨*proof*⟩

lemmas $lcm\text{-}0 = lcm\text{-}0\text{-}right$

lemma *lcm-unique*:
$a\ dvd\ d \wedge b\ dvd\ d \wedge normalize\ d = d \wedge (\forall e.\ a\ dvd\ e \wedge b\ dvd\ e \longrightarrow d\ dvd\ e) \longleftrightarrow d = lcm\ a\ b$
⟨*proof*⟩

lemma *lcm-coprime*: $gcd\ a\ b = 1 \Longrightarrow lcm\ a\ b = normalize\ (a * b)$
⟨*proof*⟩

lemma *lcm-proj1-if-dvd*: $b\ dvd\ a \Longrightarrow lcm\ a\ b = normalize\ a$
⟨*proof*⟩

lemma *lcm-proj2-if-dvd*: $a\ dvd\ b \Longrightarrow lcm\ a\ b = normalize\ b$
⟨*proof*⟩

lemma *lcm-proj1-iff*: $lcm\ m\ n = normalize\ m \longleftrightarrow n\ dvd\ m$
⟨*proof*⟩

lemma *lcm-proj2-iff*: $lcm\ m\ n = normalize\ n \longleftrightarrow m\ dvd\ n$
⟨*proof*⟩

lemma *lcm-mult-distrib'*: $normalize\ c * lcm\ a\ b = lcm\ (c * a)\ (c * b)$
⟨*proof*⟩

lemma *lcm-mult-distrib*: $k * lcm\ a\ b = lcm\ (k * a)\ (k * b) * unit\text{-}factor\ k$
⟨*proof*⟩

lemma *dvd-productE*:
assumes $p\ dvd\ (a * b)$
obtains $x\ y$ **where** $p = x * y\ x\ dvd\ a\ y\ dvd\ b$
⟨*proof*⟩

lemma *coprime-crossproduct'*:
fixes $a\ b\ c\ d$
assumes $b \neq 0$
assumes *unit-factors*: $unit\text{-}factor\ b = unit\text{-}factor\ d$
assumes *coprime*: $coprime\ a\ b\ coprime\ c\ d$
shows $a * d = b * c \longleftrightarrow a = c \wedge b = d$
⟨*proof*⟩

end

class $ring\text{-}gcd = comm\text{-}ring\text{-}1 + semiring\text{-}gcd$
begin

lemma *coprime-minus-one*: $coprime\ (n - 1)\ n$
⟨*proof*⟩

lemma *gcd-neg1* [*simp*]: $gcd\ (-a)\ b = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-neg2* [*simp*]: $gcd\ a\ (-b) = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-neg-numeral-1* [*simp*]: $gcd\ (-\ numeral\ n)\ a = gcd\ (numeral\ n)\ a$
⟨*proof*⟩

lemma *gcd-neg-numeral-2* [*simp*]: $gcd\ a\ (-\ numeral\ n) = gcd\ a\ (numeral\ n)$
⟨*proof*⟩

lemma *gcd-diff1*: $gcd\ (m - n)\ n = gcd\ m\ n$
⟨*proof*⟩

lemma *gcd-diff2*: $gcd\ (n - m)\ n = gcd\ m\ n$
⟨*proof*⟩

lemma *lcm-neg1* [*simp*]: $lcm\ (-a)\ b = lcm\ a\ b$
⟨*proof*⟩

lemma *lcm-neg2* [*simp*]: $lcm\ a\ (-b) = lcm\ a\ b$
⟨*proof*⟩

lemma *lcm-neg-numeral-1* [*simp*]: $lcm\ (-\ numeral\ n)\ a = lcm\ (numeral\ n)\ a$
⟨*proof*⟩

lemma *lcm-neg-numeral-2* [*simp*]: $lcm\ a\ (-\ numeral\ n) = lcm\ a\ (numeral\ n)$
⟨*proof*⟩

end

class *semiring-Gcd* = *semiring-gcd* + *Gcd* +
assumes *Gcd-dvd*: $a \in A \Longrightarrow Gcd\ A\ dvd\ a$
and *Gcd-greatest*: $(\bigwedge b.\ b \in A \Longrightarrow a\ dvd\ b) \Longrightarrow a\ dvd\ Gcd\ A$
and *normalize-Gcd* [*simp*]: $normalize\ (Gcd\ A) = Gcd\ A$
assumes *dvd-Lcm*: $a \in A \Longrightarrow a\ dvd\ Lcm\ A$
and *Lcm-least*: $(\bigwedge b.\ b \in A \Longrightarrow b\ dvd\ a) \Longrightarrow Lcm\ A\ dvd\ a$
and *normalize-Lcm* [*simp*]: $normalize\ (Lcm\ A) = Lcm\ A$
begin

lemma *Lcm-Gcd*: $Lcm\ A = Gcd\ \{b.\ \forall\, a{\in}A.\ a\ dvd\ b\}$
⟨*proof*⟩

lemma *Gcd-Lcm*: $Gcd\ A = Lcm\ \{b.\ \forall\, a{\in}A.\ b\ dvd\ a\}$
⟨*proof*⟩

lemma *Gcd-empty* [*simp*]: $Gcd\ \{\} = 0$
⟨*proof*⟩

lemma *Lcm-empty* [*simp*]: $Lcm\ \{\} = 1$
⟨*proof*⟩

lemma *Gcd-insert* [*simp*]: $Gcd\ (insert\ a\ A) = gcd\ a\ (Gcd\ A)$
⟨*proof*⟩

lemma *Lcm-insert* [*simp*]: $Lcm\ (insert\ a\ A) = lcm\ a\ (Lcm\ A)$
⟨*proof*⟩

lemma *LcmI*:
assumes $\bigwedge a.\ a \in A \Longrightarrow a\ dvd\ b$
and $\bigwedge c.\ (\bigwedge a.\ a \in A \Longrightarrow a\ dvd\ c) \Longrightarrow b\ dvd\ c$
and $normalize\ b = b$
shows $b = Lcm\ A$
⟨*proof*⟩

lemma *Lcm-subset*: $A \subseteq B \Longrightarrow Lcm\ A\ dvd\ Lcm\ B$
⟨*proof*⟩

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

lemma *Gcd-0-iff* [*simp*]: $Gcd\ A = 0 \longleftrightarrow A \subseteq \{0\}$

(**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *Lcm-1-iff* [*simp*]: *Lcm A* = *1* ⟷ (∀ *a*∈*A*. *is-unit a*)
(**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *unit-factor-Lcm*: *unit-factor* (*Lcm A*) = (*if Lcm A* = *0 then 0 else 1*)
⟨*proof*⟩

lemma *unit-factor-Gcd*: *unit-factor* (*Gcd A*) = (*if Gcd A* = *0 then 0 else 1*)
⟨*proof*⟩

lemma *GcdI*:
assumes ⋀*a*. *a* ∈ *A* ⟹ *b dvd a*
and ⋀*c*. (⋀*a*. *a* ∈ *A* ⟹ *c dvd a*) ⟹ *c dvd b*
and *normalize b* = *b*
shows *b* = *Gcd A*
⟨*proof*⟩

lemma *Gcd-eq-1-I*:
assumes *is-unit a* **and** *a* ∈ *A*
shows *Gcd A* = *1*
⟨*proof*⟩

lemma *Lcm-eq-0-I*:
assumes *0* ∈ *A*
shows *Lcm A* = *0*
⟨*proof*⟩

lemma *Gcd-UNIV* [*simp*]: *Gcd UNIV* = *1*
⟨*proof*⟩

lemma *Lcm-UNIV* [*simp*]: *Lcm UNIV* = *0*
⟨*proof*⟩

lemma *Lcm-0-iff*:
assumes *finite A*
shows *Lcm A* = *0* ⟷ *0* ∈ *A*
⟨*proof*⟩

lemma *Gcd-image-normalize* [*simp*]: *Gcd* (*normalize* ' *A*) = *Gcd A*
⟨*proof*⟩

lemma *Gcd-eqI*:
assumes *normalize a* = *a*
assumes ⋀*b*. *b* ∈ *A* ⟹ *a dvd b*
and ⋀*c*. (⋀*b*. *b* ∈ *A* ⟹ *c dvd b*) ⟹ *c dvd a*
shows *Gcd A* = *a*

⟨proof⟩

lemma *dvd-GcdD*: $x\ dvd\ Gcd\ A \Longrightarrow y \in A \Longrightarrow x\ dvd\ y$
⟨proof⟩

lemma *dvd-Gcd-iff*: $x\ dvd\ Gcd\ A \longleftrightarrow (\forall\, y{\in}A.\ x\ dvd\ y)$
⟨proof⟩

lemma *Gcd-mult*: $Gcd\ (op * c\ `\ A) = normalize\ c * Gcd\ A$
⟨proof⟩

lemma *Lcm-eqI*:
assumes $normalize\ a = a$
and $\bigwedge b.\ b \in A \Longrightarrow b\ dvd\ a$
and $\bigwedge c.\ (\bigwedge b.\ b \in A \Longrightarrow b\ dvd\ c) \Longrightarrow a\ dvd\ c$
shows $Lcm\ A = a$
⟨proof⟩

lemma *Lcm-dvdD*: $Lcm\ A\ dvd\ x \Longrightarrow y \in A \Longrightarrow y\ dvd\ x$
⟨proof⟩

lemma *Lcm-dvd-iff*: $Lcm\ A\ dvd\ x \longleftrightarrow (\forall\, y{\in}A.\ y\ dvd\ x)$
⟨proof⟩

lemma *Lcm-mult*:
assumes $A \neq \{\}$
shows $Lcm\ (op * c\ `\ A) = normalize\ c * Lcm\ A$
⟨proof⟩

lemma *Lcm-no-units*: $Lcm\ A = Lcm\ (A - \{a.\ is\text{-}unit\ a\})$
⟨proof⟩

lemma *Lcm-0-iff'*: $Lcm\ A = 0 \longleftrightarrow (\nexists\, l.\ l \neq 0 \wedge (\forall\, a{\in}A.\ a\ dvd\ l))$
⟨proof⟩

lemma *Lcm-no-multiple*: $(\forall\, m.\ m \neq 0 \longrightarrow (\exists\, a{\in}A.\ \neg\ a\ dvd\ m)) \Longrightarrow Lcm\ A = 0$
⟨proof⟩

lemma *Lcm-singleton* [*simp*]: $Lcm\ \{a\} = normalize\ a$
⟨proof⟩

lemma *Lcm-2* [*simp*]: $Lcm\ \{a,\ b\} = lcm\ a\ b$
⟨proof⟩

lemma *Lcm-coprime*:
assumes *finite A*
and $A \neq \{\}$
and $\bigwedge a\ b.\ a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow gcd\ a\ b = 1$
shows $Lcm\ A = normalize\ (\prod A)$

⟨*proof*⟩

lemma *Lcm-coprime'*:
card A ≠ 0 ⟹
(⋀a b. a ∈ A ⟹ b ∈ A ⟹ a ≠ b ⟹ gcd a b = 1) ⟹
Lcm A = normalize (∏ A)
⟨*proof*⟩

lemma *Gcd-1*: *1 ∈ A ⟹ Gcd A = 1*
⟨*proof*⟩

lemma *Gcd-singleton* [*simp*]: *Gcd {a} = normalize a*
⟨*proof*⟩

lemma *Gcd-2* [*simp*]: *Gcd {a, b} = gcd a b*
⟨*proof*⟩

definition *pairwise-coprime*
where *pairwise-coprime A = (∀ x y. x ∈ A ∧ y ∈ A ∧ x ≠ y ⟶ coprime x y)*

lemma *pairwise-coprimeI* [*intro?*]:
(⋀x y. x ∈ A ⟹ y ∈ A ⟹ x ≠ y ⟹ coprime x y) ⟹ pairwise-coprime A
⟨*proof*⟩

lemma *pairwise-coprimeD*:
pairwise-coprime A ⟹ x ∈ A ⟹ y ∈ A ⟹ x ≠ y ⟹ coprime x y
⟨*proof*⟩

lemma *pairwise-coprime-subset*: *pairwise-coprime A ⟹ B ⊆ A ⟹ pairwise-coprime B*
⟨*proof*⟩

end

86.3 An aside: GCD and LCM on finite sets for incomplete gcd rings

context *semiring-gcd*
begin

sublocale *Gcd-fin*: *bounded-quasi-semilattice-set gcd 0 1 normalize*
defines
Gcd-fin (Gcd_{fin} - [*900*] *900*) = *Gcd-fin.F* :: *'a set ⇒ 'a* ⟨*proof*⟩

abbreviation *gcd-list* :: *'a list ⇒ 'a*
where *gcd-list xs ≡* Gcd_{fin} *(set xs)*

sublocale *Lcm-fin*: *bounded-quasi-semilattice-set lcm 1 0 normalize*

defines
Lcm-fin (Lcm_{fin} - [900] 900) = *Lcm-fin.F* ⟨*proof*⟩

abbreviation *lcm-list* :: *'a list* ⇒ *'a*
where *lcm-list xs* ≡ Lcm_{fin} (*set xs*)

lemma *Gcd-fin-dvd*:
$a \in A \Longrightarrow Gcd_{fin}\ A\ dvd\ a$
⟨*proof*⟩

lemma *dvd-Lcm-fin*:
$a \in A \Longrightarrow a\ dvd\ Lcm_{fin}\ A$
⟨*proof*⟩

lemma *Gcd-fin-greatest*:
a dvd Gcd_{fin} *A* **if** *finite A* **and** $\bigwedge b.\ b \in A \Longrightarrow a\ dvd\ b$
⟨*proof*⟩

lemma *Lcm-fin-least*:
Lcm_{fin} *A dvd a* **if** *finite A* **and** $\bigwedge b.\ b \in A \Longrightarrow b\ dvd\ a$
⟨*proof*⟩

lemma *gcd-list-greatest*:
a dvd gcd-list bs **if** $\bigwedge b.\ b \in set\ bs \Longrightarrow a\ dvd\ b$
⟨*proof*⟩

lemma *lcm-list-least*:
lcm-list bs dvd a **if** $\bigwedge b.\ b \in set\ bs \Longrightarrow b\ dvd\ a$
⟨*proof*⟩

lemma *dvd-Gcd-fin-iff*:
b dvd Gcd_{fin} *A* ⟷ (∀ *a*∈*A*. *b dvd a*) **if** *finite A*
⟨*proof*⟩

lemma *dvd-gcd-list-iff*:
b dvd gcd-list xs ⟷ (∀ *a*∈*set xs*. *b dvd a*)
⟨*proof*⟩

lemma *Lcm-fin-dvd-iff*:
Lcm_{fin} *A dvd b* ⟷ (∀ *a*∈*A*. *a dvd b*) **if** *finite A*
⟨*proof*⟩

lemma *lcm-list-dvd-iff*:
lcm-list xs dvd b ⟷ (∀ *a*∈*set xs*. *a dvd b*)
⟨*proof*⟩

lemma *Gcd-fin-mult*:
Gcd_{fin} (*image* (*times b*) *A*) = *normalize b* ∗ Gcd_{fin} *A* **if** *finite A*
⟨*proof*⟩

lemma *Lcm-fin-mult*:
Lcm_{fin} (*image* (*times* *b*) *A*) = *normalize* *b* ∗ Lcm_{fin} *A* **if** $A \neq \{\}$
⟨*proof*⟩

lemma *unit-factor-Gcd-fin*:
unit-factor (Gcd_{fin} *A*) = *of-bool* (Gcd_{fin} $A \neq 0$)
⟨*proof*⟩

lemma *unit-factor-Lcm-fin*:
unit-factor (Lcm_{fin} *A*) = *of-bool* (Lcm_{fin} $A \neq 0$)
⟨*proof*⟩

lemma *is-unit-Gcd-fin-iff* [*simp*]:
is-unit (Gcd_{fin} *A*) ⟷ Gcd_{fin} $A = 1$
⟨*proof*⟩

lemma *is-unit-Lcm-fin-iff* [*simp*]:
is-unit (Lcm_{fin} *A*) ⟷ Lcm_{fin} $A = 1$
⟨*proof*⟩

lemma *Gcd-fin-0-iff*:
Gcd_{fin} $A = 0 \longleftrightarrow A \subseteq \{0\}$ ∧ *finite* *A*
⟨*proof*⟩

lemma *Lcm-fin-0-iff*:
Lcm_{fin} $A = 0 \longleftrightarrow 0 \in A$ **if** *finite* *A*
⟨*proof*⟩

lemma *Lcm-fin-1-iff*:
Lcm_{fin} $A = 1 \longleftrightarrow (\forall a \in A.$ *is-unit* *a*) ∧ *finite* *A*
⟨*proof*⟩

end

context *semiring-Gcd*
begin

lemma *Gcd-fin-eq-Gcd* [*simp*]:
Gcd_{fin} *A* = *Gcd* *A* **if** *finite* *A* **for** *A* :: *'a set*
⟨*proof*⟩

lemma *Gcd-set-eq-fold* [*code-unfold*]:
Gcd (*set* *xs*) = *fold* *gcd* *xs* *0*
⟨*proof*⟩

lemma *Lcm-fin-eq-Lcm* [*simp*]:
Lcm_{fin} *A* = *Lcm* *A* **if** *finite* *A* **for** *A* :: *'a set*
⟨*proof*⟩

lemma *Lcm-set-eq-fold* [*code-unfold*]:
  $Lcm\ (set\ xs) = fold\ lcm\ xs\ 1$
  ⟨*proof*⟩

end

86.4 GCD and LCM on *nat* and *int*

instantiation *nat* :: *gcd*
begin

fun *gcd-nat* :: $nat \Rightarrow nat \Rightarrow nat$
  where $gcd\text{-}nat\ x\ y = (if\ y = 0\ then\ x\ else\ gcd\ y\ (x\ mod\ y))$

definition *lcm-nat* :: $nat \Rightarrow nat \Rightarrow nat$
  where $lcm\text{-}nat\ x\ y = x * y\ div\ (gcd\ x\ y)$

instance ⟨*proof*⟩

end

instantiation *int* :: *gcd*
begin

definition *gcd-int* :: $int \Rightarrow int \Rightarrow int$
  where $gcd\text{-}int\ x\ y = int\ (gcd\ (nat\ |x|)\ (nat\ |y|))$

definition *lcm-int* :: $int \Rightarrow int \Rightarrow int$
  where $lcm\text{-}int\ x\ y = int\ (lcm\ (nat\ |x|)\ (nat\ |y|))$

instance ⟨*proof*⟩

end

Transfer setup

lemma *transfer-nat-int-gcd*:
  $x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow gcd\ (nat\ x)\ (nat\ y) = nat\ (gcd\ x\ y)$
  $x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow lcm\ (nat\ x)\ (nat\ y) = nat\ (lcm\ x\ y)$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *transfer-nat-int-gcd-closures*:
  $x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow gcd\ x\ y \geq 0$
  $x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow lcm\ x\ y \geq 0$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

declare *transfer-morphism-nat-int*

[*transfer add return*: *transfer-nat-int-gcd transfer-nat-int-gcd-closures*]

lemma *transfer-int-nat-gcd*:
$gcd\ (int\ x)\ (int\ y) = int\ (gcd\ x\ y)$
$lcm\ (int\ x)\ (int\ y) = int\ (lcm\ x\ y)$
⟨*proof*⟩

lemma *transfer-int-nat-gcd-closures*:
$is\text{-}nat\ x \Longrightarrow is\text{-}nat\ y \Longrightarrow gcd\ x\ y >= 0$
$is\text{-}nat\ x \Longrightarrow is\text{-}nat\ y \Longrightarrow lcm\ x\ y >= 0$
⟨*proof*⟩

declare *transfer-morphism-int-nat*
[*transfer add return*: *transfer-int-nat-gcd transfer-int-nat-gcd-closures*]

lemma *gcd-nat-induct*:
fixes $m\ n :: nat$
assumes $\bigwedge m.\ P\ m\ 0$
and $\bigwedge m\ n.\ 0 < n \Longrightarrow P\ n\ (m\ mod\ n) \Longrightarrow P\ m\ n$
shows $P\ m\ n$
⟨*proof*⟩

Specific to *int*.

lemma *gcd-eq-int-iff*: $gcd\ k\ l = int\ n \longleftrightarrow gcd\ (nat\ |k|)\ (nat\ |l|) = n$
⟨*proof*⟩

lemma *lcm-eq-int-iff*: $lcm\ k\ l = int\ n \longleftrightarrow lcm\ (nat\ |k|)\ (nat\ |l|) = n$
⟨*proof*⟩

lemma *gcd-neg1-int* [*simp*]: $gcd\ (-\ x)\ y = gcd\ x\ y$
for $x\ y :: int$
⟨*proof*⟩

lemma *gcd-neg2-int* [*simp*]: $gcd\ x\ (-\ y) = gcd\ x\ y$
for $x\ y :: int$
⟨*proof*⟩

lemma *abs-gcd-int* [*simp*]: $|gcd\ x\ y| = gcd\ x\ y$
for $x\ y :: int$
⟨*proof*⟩

lemma *gcd-abs-int*: $gcd\ x\ y = gcd\ |x|\ |y|$
for $x\ y :: int$
⟨*proof*⟩

lemma *gcd-abs1-int* [*simp*]: $gcd\ |x|\ y = gcd\ x\ y$
for $x\ y :: int$
⟨*proof*⟩

lemma *gcd-abs2-int* [*simp*]: $gcd\ x\ |y| = gcd\ x\ y$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *gcd-cases-int*:
  fixes $x\ y$ :: *int*
  assumes $x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow P\ (gcd\ x\ y)$
    and $x \geq 0 \Longrightarrow y \leq 0 \Longrightarrow P\ (gcd\ x\ (-\ y))$
    and $x \leq 0 \Longrightarrow y \geq 0 \Longrightarrow P\ (gcd\ (-\ x)\ y)$
    and $x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow P\ (gcd\ (-\ x)\ (-\ y))$
  shows $P\ (gcd\ x\ y)$
  ⟨*proof*⟩

lemma *gcd-ge-0-int* [*simp*]: $gcd\ (x::int)\ y >= 0$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *lcm-neg1-int*: $lcm\ (-\ x)\ y = lcm\ x\ y$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *lcm-neg2-int*: $lcm\ x\ (-\ y) = lcm\ x\ y$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *lcm-abs-int*: $lcm\ x\ y = lcm\ |x|\ |y|$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *abs-lcm-int* [*simp*]: $|lcm\ i\ j| = lcm\ i\ j$
  for $i\ j$ :: *int*
  ⟨*proof*⟩

lemma *lcm-abs1-int* [*simp*]: $lcm\ |x|\ y = lcm\ x\ y$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *lcm-abs2-int* [*simp*]: $lcm\ x\ |y| = lcm\ x\ y$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *lcm-cases-int*:
  fixes $x\ y$ :: *int*
  assumes $x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow P\ (lcm\ x\ y)$
    and $x \geq 0 \Longrightarrow y \leq 0 \Longrightarrow P\ (lcm\ x\ (-\ y))$
    and $x \leq 0 \Longrightarrow y \geq 0 \Longrightarrow P\ (lcm\ (-\ x)\ y)$
    and $x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow P\ (lcm\ (-\ x)\ (-\ y))$
  shows $P\ (lcm\ x\ y)$
  ⟨*proof*⟩

lemma *lcm-ge-0-int* [*simp*]: *lcm x y* $\geq$ *0*
  for *x y* :: *int*
  ⟨*proof*⟩

lemma *gcd-0-nat*: *gcd x 0* = *x*
  for *x* :: *nat*
  ⟨*proof*⟩

lemma *gcd-0-int* [*simp*]: *gcd x 0* = $|x|$
  for *x* :: *int*
  ⟨*proof*⟩

lemma *gcd-0-left-nat*: *gcd 0 x* = *x*
  for *x* :: *nat*
  ⟨*proof*⟩

lemma *gcd-0-left-int* [*simp*]: *gcd 0 x* = $|x|$
  for *x* :: *int*
  ⟨*proof*⟩

lemma *gcd-red-nat*: *gcd x y* = *gcd y* (*x mod y*)
  for *x y* :: *nat*
  ⟨*proof*⟩

Weaker, but useful for the simplifier.

lemma *gcd-non-0-nat*: *y* $\neq$ *0* $\Longrightarrow$ *gcd x y* = *gcd y* (*x mod y*)
  for *x y* :: *nat*
  ⟨*proof*⟩

lemma *gcd-1-nat* [*simp*]: *gcd m 1* = *1*
  for *m* :: *nat*
  ⟨*proof*⟩

lemma *gcd-Suc-0* [*simp*]: *gcd m* (*Suc 0*) = *Suc 0*
  for *m* :: *nat*
  ⟨*proof*⟩

lemma *gcd-1-int* [*simp*]: *gcd m 1* = *1*
  for *m* :: *int*
  ⟨*proof*⟩

lemma *gcd-idem-nat*: *gcd x x* = *x*
  for *x* :: *nat*
  ⟨*proof*⟩

lemma *gcd-idem-int*: *gcd x x* = $|x|$
  for *x* :: *int*
  ⟨*proof*⟩

declare *gcd-nat.simps* [*simp del*]

gcd m n divides *m* and *n*. The conjunctions don't seem provable separately.

instance *nat* :: *semiring-gcd*
⟨*proof*⟩

instance *int* :: *ring-gcd*
⟨*proof*⟩

lemma *gcd-le1-nat* [*simp*]: $a \neq 0 \Longrightarrow gcd\ a\ b \leq a$
for *a b* :: *nat*
⟨*proof*⟩

lemma *gcd-le2-nat* [*simp*]: $b \neq 0 \Longrightarrow gcd\ a\ b \leq b$
for *a b* :: *nat*
⟨*proof*⟩

lemma *gcd-le1-int* [*simp*]: $a > 0 \Longrightarrow gcd\ a\ b \leq a$
for *a b* :: *int*
⟨*proof*⟩

lemma *gcd-le2-int* [*simp*]: $b > 0 \Longrightarrow gcd\ a\ b \leq b$
for *a b* :: *int*
⟨*proof*⟩

lemma *gcd-pos-nat* [*simp*]: $gcd\ m\ n > 0 \longleftrightarrow m \neq 0 \lor n \neq 0$
for *m n* :: *nat*
⟨*proof*⟩

lemma *gcd-pos-int* [*simp*]: $gcd\ m\ n > 0 \longleftrightarrow m \neq 0 \lor n \neq 0$
for *m n* :: *int*
⟨*proof*⟩

lemma *gcd-unique-nat*: $d\ dvd\ a \land d\ dvd\ b \land (\forall e.\ e\ dvd\ a \land e\ dvd\ b \longrightarrow e\ dvd\ d) \longleftrightarrow d = gcd\ a\ b$
for *d a* :: *nat*
⟨*proof*⟩

lemma *gcd-unique-int*:
$d \geq 0 \land d\ dvd\ a \land d\ dvd\ b \land (\forall e.\ e\ dvd\ a \land e\ dvd\ b \longrightarrow e\ dvd\ d) \longleftrightarrow d = gcd\ a\ b$
for *d a* :: *int*
⟨*proof*⟩

interpretation *gcd-nat*:
semilattice-neutr-order gcd 0::nat Rings.dvd $\lambda m\ n.\ m\ dvd\ n \land m \neq n$
⟨*proof*⟩

lemma *gcd-proj1-if-dvd-int* [*simp*]: x *dvd* $y \Longrightarrow gcd\ x\ y = |x|$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *gcd-proj2-if-dvd-int* [*simp*]: y *dvd* $x \Longrightarrow gcd\ x\ y = |y|$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

Multiplication laws.

lemma *gcd-mult-distrib-nat*: $k * gcd\ m\ n = gcd\ (k * m)\ (k * n)$
  for $k\ m\ n$ :: *nat*
  — [1, page 27]
  ⟨*proof*⟩

lemma *gcd-mult-distrib-int*: $|k| * gcd\ m\ n = gcd\ (k * m)\ (k * n)$
  for $k\ m\ n$ :: *int*
  ⟨*proof*⟩

lemma *coprime-crossproduct-nat*:
  fixes $a\ b\ c\ d$ :: *nat*
  assumes *coprime* $a\ d$ **and** *coprime* $b\ c$
  shows $a * c = b * d \longleftrightarrow a = b \land c = d$
  ⟨*proof*⟩

lemma *coprime-crossproduct-int*:
  fixes $a\ b\ c\ d$ :: *int*
  assumes *coprime* $a\ d$ **and** *coprime* $b\ c$
  shows $|a| * |c| = |b| * |d| \longleftrightarrow |a| = |b| \land |c| = |d|$
  ⟨*proof*⟩

Addition laws.

lemma *gcd-diff1-nat*: $m \geq n \Longrightarrow gcd\ (m - n)\ n = gcd\ m\ n$
  for $m\ n$ :: *nat*
  ⟨*proof*⟩

lemma *gcd-diff2-nat*: $n \geq m \Longrightarrow gcd\ (n - m)\ n = gcd\ m\ n$
  for $m\ n$ :: *nat*
  ⟨*proof*⟩

lemma *gcd-non-0-int*: $y > 0 \Longrightarrow gcd\ x\ y = gcd\ y\ (x\ mod\ y)$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *gcd-red-int*: $gcd\ x\ y = gcd\ y\ (x\ mod\ y)$
  for $x\ y$ :: *int*
  ⟨*proof*⟩

lemma *finite-divisors-nat* [*simp*]:
  fixes m :: *nat*
  assumes $m > 0$
  shows *finite* $\{d.\ d\ dvd\ m\}$
⟨*proof*⟩

lemma *finite-divisors-int* [*simp*]:
  fixes i :: *int*
  assumes $i \neq 0$
  shows *finite* $\{d.\ d\ dvd\ i\}$
⟨*proof*⟩

lemma *Max-divisors-self-nat* [*simp*]: $n \neq 0 \Longrightarrow Max\ \{d::nat.\ d\ dvd\ n\} = n$
  ⟨*proof*⟩

lemma *Max-divisors-self-int* [*simp*]: $n \neq 0 \Longrightarrow Max\ \{d::int.\ d\ dvd\ n\} = |n|$
  ⟨*proof*⟩

lemma *gcd-is-Max-divisors-nat*: $m > 0 \Longrightarrow n > 0 \Longrightarrow gcd\ m\ n = Max\ \{d.\ d\ dvd\ m \wedge d\ dvd\ n\}$
  for $m\ n$ :: *nat*
  ⟨*proof*⟩

lemma *gcd-is-Max-divisors-int*: $m \neq 0 \Longrightarrow n \neq 0 \Longrightarrow gcd\ m\ n = Max\ \{d.\ d\ dvd\ m \wedge d\ dvd\ n\}$
  for $m\ n$ :: *int*
  ⟨*proof*⟩

lemma *gcd-code-int* [*code*]: $gcd\ k\ l = |if\ l = 0\ then\ k\ else\ gcd\ l\ (|k|\ mod\ |l|)|$
  for $k\ l$ :: *int*
  ⟨*proof*⟩

86.5 Coprimality

lemma *coprime-nat*: $coprime\ a\ b \longleftrightarrow (\forall\ d.\ d\ dvd\ a \wedge d\ dvd\ b \longleftrightarrow d = 1)$
  for $a\ b$ :: *nat*
  ⟨*proof*⟩

lemma *coprime-Suc-0-nat*: $coprime\ a\ b \longleftrightarrow (\forall\ d.\ d\ dvd\ a \wedge d\ dvd\ b \longleftrightarrow d = Suc\ 0)$
  for $a\ b$ :: *nat*
  ⟨*proof*⟩

lemma *coprime-int*: $coprime\ a\ b \longleftrightarrow (\forall\ d.\ d \geq 0 \wedge d\ dvd\ a \wedge d\ dvd\ b \longleftrightarrow d = 1)$
  for $a\ b$ :: *int*

⟨*proof*⟩

lemma *pow-divides-eq-nat* [*simp*]: $n > 0 \Longrightarrow a\hat{\ }n\ dvd\ b\hat{\ }n \longleftrightarrow a\ dvd\ b$
for $a\ b\ n :: nat$
⟨*proof*⟩

lemma *coprime-Suc-nat* [*simp*]: *coprime* (*Suc n*) *n*
⟨*proof*⟩

lemma *coprime-minus-one-nat*: $n \neq 0 \Longrightarrow coprime\ (n - 1)\ n$
for $n :: nat$
⟨*proof*⟩

lemma *coprime-common-divisor-nat*: $coprime\ a\ b \Longrightarrow x\ dvd\ a \Longrightarrow x\ dvd\ b \Longrightarrow x = 1$
for $a\ b :: nat$
⟨*proof*⟩

lemma *coprime-common-divisor-int*: $coprime\ a\ b \Longrightarrow x\ dvd\ a \Longrightarrow x\ dvd\ b \Longrightarrow |x| = 1$
for $a\ b :: int$
⟨*proof*⟩

lemma *invertible-coprime-nat*: $x * y\ mod\ m = 1 \Longrightarrow coprime\ x\ m$
for $m\ x\ y :: nat$
⟨*proof*⟩

lemma *invertible-coprime-int*: $x * y\ mod\ m = 1 \Longrightarrow coprime\ x\ m$
for $m\ x\ y :: int$
⟨*proof*⟩

86.6 Bezout's theorem

Function *bezw* returns a pair of witnesses to Bezout's theorem – see the theorems that follow the definition.

fun $bezw :: nat \Rightarrow nat \Rightarrow int * int$
where $bezw\ x\ y =$
$(if\ y = 0\ then\ (1,\ 0)$
$else$
$(snd\ (bezw\ y\ (x\ mod\ y)),$
$fst\ (bezw\ y\ (x\ mod\ y)) - snd\ (bezw\ y\ (x\ mod\ y)) * int(x\ div\ y)))$

lemma *bezw-0* [*simp*]: $bezw\ x\ 0 = (1,\ 0)$
⟨*proof*⟩

lemma *bezw-non-0*:
$y > 0 \Longrightarrow bezw\ x\ y =$
$(snd\ (bezw\ y\ (x\ mod\ y)), fst\ (bezw\ y\ (x\ mod\ y)) - snd\ (bezw\ y\ (x\ mod\ y)) * int(x\ div\ y))$

⟨*proof*⟩

declare *bezw.simps* [*simp del*]

lemma *bezw-aux*: $fst\ (bezw\ x\ y) * int\ x + snd\ (bezw\ x\ y) * int\ y = int\ (gcd\ x\ y)$
⟨*proof*⟩

lemma *bezout-int*: $\exists\, u\ v.\ u * x + v * y = gcd\ x\ y$
for $x\ y :: int$
⟨*proof*⟩

Versions of Bezout for *nat*, by Amine Chaieb.

lemma *ind-euclid*:
fixes $P :: nat \Rightarrow nat \Rightarrow bool$
assumes *c*: $\forall\, a\ b.\ P\ a\ b \longleftrightarrow P\ b\ a$
and *z*: $\forall\, a.\ P\ a\ 0$
and *add*: $\forall\, a\ b.\ P\ a\ b \longrightarrow P\ a\ (a + b)$
shows $P\ a\ b$
⟨*proof*⟩

lemma *bezout-lemma-nat*:
assumes *ex*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge$
$(a * x = b * y + d \vee b * x = a * y + d)$
shows $\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ a + b \wedge$
$(a * x = (a + b) * y + d \vee (a + b) * x = a * y + d)$
⟨*proof*⟩

lemma *bezout-add-nat*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge$
$(a * x = b * y + d \vee b * x = a * y + d)$
⟨*proof*⟩

lemma *bezout1-nat*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge$
$(a * x - b * y = d \vee b * x - a * y = d)$
⟨*proof*⟩

lemma *bezout-add-strong-nat*:
fixes $a\ b :: nat$
assumes *a*: $a \neq 0$
shows $\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge a * x = b * y + d$
⟨*proof*⟩

lemma *bezout-nat*:
fixes $a :: nat$
assumes *a*: $a \neq 0$
shows $\exists\, x\ y.\ a * x = b * y + gcd\ a\ b$
⟨*proof*⟩

86.7 LCM properties on *nat* and *int*

lemma *lcm-altdef-int* [*code*]: $lcm\ a\ b = |a| * |b|\ div\ gcd\ a\ b$
for $a\ b$ :: *int*
⟨*proof*⟩

lemma *prod-gcd-lcm-nat*: $m * n = gcd\ m\ n * lcm\ m\ n$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *prod-gcd-lcm-int*: $|m| * |n| = gcd\ m\ n * lcm\ m\ n$
for $m\ n$ :: *int*
⟨*proof*⟩

lemma *lcm-pos-nat*: $m > 0 \Longrightarrow n > 0 \Longrightarrow lcm\ m\ n > 0$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *lcm-pos-int*: $m \neq 0 \Longrightarrow n \neq 0 \Longrightarrow lcm\ m\ n > 0$
for $m\ n$ :: *int*
⟨*proof*⟩

lemma *dvd-pos-nat*: $n > 0 \Longrightarrow m\ dvd\ n \Longrightarrow m > 0$
for $m\ n$ :: *nat*
⟨*proof*⟩

lemma *lcm-unique-nat*:
$a\ dvd\ d \wedge b\ dvd\ d \wedge (\forall e.\ a\ dvd\ e \wedge b\ dvd\ e \longrightarrow d\ dvd\ e) \longleftrightarrow d = lcm\ a\ b$
for $a\ b\ d$ :: *nat*
⟨*proof*⟩

lemma *lcm-unique-int*:
$d \geq 0 \wedge a\ dvd\ d \wedge b\ dvd\ d \wedge (\forall e.\ a\ dvd\ e \wedge b\ dvd\ e \longrightarrow d\ dvd\ e) \longleftrightarrow d = lcm\ a\ b$
for $a\ b\ d$ :: *int*
⟨*proof*⟩

lemma *lcm-proj2-if-dvd-nat* [*simp*]: $x\ dvd\ y \Longrightarrow lcm\ x\ y = y$
for $x\ y$ :: *nat*
⟨*proof*⟩

lemma *lcm-proj2-if-dvd-int* [*simp*]: $x\ dvd\ y \Longrightarrow lcm\ x\ y = |y|$
for $x\ y$ :: *int*
⟨*proof*⟩

lemma *lcm-proj1-if-dvd-nat* [*simp*]: $x\ dvd\ y \Longrightarrow lcm\ y\ x = y$
for $x\ y$ :: *nat*
⟨*proof*⟩

lemma *lcm-proj1-if-dvd-int* [*simp*]: $x\ dvd\ y \Longrightarrow lcm\ y\ x = |y|$

for $x\ y :: int$
⟨*proof*⟩

lemma *lcm-proj1-iff-nat* [*simp*]: $lcm\ m\ n = m \longleftrightarrow n\ dvd\ m$
for $m\ n :: nat$
⟨*proof*⟩

lemma *lcm-proj2-iff-nat* [*simp*]: $lcm\ m\ n = n \longleftrightarrow m\ dvd\ n$
for $m\ n :: nat$
⟨*proof*⟩

lemma *lcm-proj1-iff-int* [*simp*]: $lcm\ m\ n = |m| \longleftrightarrow n\ dvd\ m$
for $m\ n :: int$
⟨*proof*⟩

lemma *lcm-proj2-iff-int* [*simp*]: $lcm\ m\ n = |n| \longleftrightarrow m\ dvd\ n$
for $m\ n :: int$
⟨*proof*⟩

lemma *lcm-1-iff-nat* [*simp*]: $lcm\ m\ n = Suc\ 0 \longleftrightarrow m = Suc\ 0 \wedge n = Suc\ 0$
for $m\ n :: nat$
⟨*proof*⟩

lemma *lcm-1-iff-int* [*simp*]: $lcm\ m\ n = 1 \longleftrightarrow (m = 1 \vee m = -1) \wedge (n = 1 \vee n = -1)$
for $m\ n :: int$
⟨*proof*⟩

86.8 The complete divisibility lattice on *nat* and *int*

Lifting *gcd* and *lcm* to sets (*Gcd* / *Lcm*). *Gcd* is defined via *Lcm* to facilitate the proof that we have a complete lattice.

instantiation *nat* :: *semiring-Gcd*
begin

interpretation *semilattice-neutr-set lcm 1::nat*
⟨*proof*⟩

definition $Lcm\ M = (\mathit{if}\ finite\ M\ then\ F\ M\ else\ 0)$ **for** $M :: nat\ set$

lemma *Lcm-nat-empty*: $Lcm\ \{\} = (1::nat)$
⟨*proof*⟩

lemma *Lcm-nat-insert*: $Lcm\ (insert\ n\ M) = lcm\ n\ (Lcm\ M)$ **for** $n :: nat$
⟨*proof*⟩

lemma *Lcm-nat-infinite*: $infinite\ M \Longrightarrow Lcm\ M = 0$ **for** $M :: nat\ set$
⟨*proof*⟩

lemma *dvd-Lcm-nat* [*simp*]:
  fixes M :: *nat set*
  assumes $m \in M$
  shows *m dvd Lcm M*
⟨*proof*⟩

lemma *Lcm-dvd-nat* [*simp*]:
  fixes M :: *nat set*
  assumes $\forall\, m\in M.\ m$ *dvd n*
  shows *Lcm M dvd n*
⟨*proof*⟩

definition *Gcd M* = *Lcm* $\{d.\ \forall\, m\in M.\ d$ *dvd* $m\}$ **for** M :: *nat set*

instance
⟨*proof*⟩

end

lemma *Gcd-nat-eq-one*: $1 \in N \Longrightarrow$ *Gcd* $N = 1$
  for N :: *nat set*
  ⟨*proof*⟩

Alternative characterizations of Gcd:

lemma *Gcd-eq-Max*:
  fixes M :: *nat set*
  assumes *finite* (M::*nat set*) **and** $M \neq \{\}$ **and** $0 \notin M$
  shows *Gcd M* = *Max* ($\bigcap\, m\in M.\ \{d.\ d$ *dvd* $m\}$)
⟨*proof*⟩

lemma *Gcd-remove0-nat*: *finite M* $\Longrightarrow$ *Gcd M* = *Gcd* ($M - \{0\}$)
  for M :: *nat set*
  ⟨*proof*⟩

lemma *Lcm-in-lcm-closed-set-nat*:
  finite M $\Longrightarrow M \neq \{\} \Longrightarrow \forall\, m\ n.\ m \in M \longrightarrow n \in M \longrightarrow$ *lcm m n* $\in M \Longrightarrow$ *Lcm M* $\in M$
  for M :: *nat set*
  ⟨*proof*⟩

lemma *Lcm-eq-Max-nat*:
  finite M $\Longrightarrow M \neq \{\} \Longrightarrow 0 \notin M \Longrightarrow \forall\, m\ n.\ m \in M \longrightarrow n \in M \longrightarrow$ *lcm m n* $\in M \Longrightarrow$ *Lcm M* = *Max M*
  for M :: *nat set*
  ⟨*proof*⟩

lemma *mult-inj-if-coprime-nat*:
  inj-on f A $\Longrightarrow$ *inj-on g B* $\Longrightarrow \forall\, a\in A.\ \forall\, b\in B.$ *coprime* ($f\ a$) ($g\ b$) $\Longrightarrow$
    inj-on ($\lambda(a,\ b).\ f\ a * g\ b$) ($A \times B$)

for f :: $'a \Rightarrow nat$ **and** g :: $'b \Rightarrow nat$
$\langle proof \rangle$

86.8.1 Setwise GCD and LCM for integers

instantiation *int* :: *semiring-Gcd*
begin

definition $Lcm\ M = int\ (LCM\ m{\in}M.\ (nat \circ abs)\ m)$

definition $Gcd\ M = int\ (GCD\ m{\in}M.\ (nat \circ abs)\ m)$

instance
$\langle proof \rangle$

end

lemma *abs-Gcd* [*simp*]: $|Gcd\ K| = Gcd\ K$
for K :: *int set*
$\langle proof \rangle$

lemma *abs-Lcm* [*simp*]: $|Lcm\ K| = Lcm\ K$
for K :: *int set*
$\langle proof \rangle$

lemma *Gcm-eq-int-iff*: $Gcd\ K = int\ n \longleftrightarrow Gcd\ ((nat \circ abs)\ `\ K) = n$
$\langle proof \rangle$

lemma *Lcm-eq-int-iff*: $Lcm\ K = int\ n \longleftrightarrow Lcm\ ((nat \circ abs)\ `\ K) = n$
$\langle proof \rangle$

86.9 GCD and LCM on *integer*

instantiation *integer* :: *gcd*
begin

context
includes *integer.lifting*
begin

lift-definition *gcd-integer* :: *integer* ⇒ *integer* ⇒ *integer* **is** *gcd* $\langle proof \rangle$

lift-definition *lcm-integer* :: *integer* ⇒ *integer* ⇒ *integer* **is** *lcm* $\langle proof \rangle$

end

instance $\langle proof \rangle$

end

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

context
includes *integer.lifting*
begin

lemma *gcd-code-integer* [*code*]: *gcd k l* = |*if l* = (*0*::*integer*) *then k else gcd l* (|*k*| *mod* |*l*|)|
⟨*proof*⟩

lemma *lcm-code-integer* [*code*]: *lcm a b* = |*a*| ∗ |*b*| *div gcd a b*
for *a b* :: *integer*
⟨*proof*⟩

end

code-printing
constant *gcd* :: *integer* ⇒ - ⇀
(*OCaml*) *Big′-int.gcd′-big′-int*
and (*Haskell*) *Prelude.gcd*
and (*Scala*) *-.gcd′*((-)′)
— There is no gcd operation in the SML standard library, so no code setup for SML

Some code equations

lemmas *Gcd-nat-set-eq-fold* [*code*] = *Gcd-set-eq-fold* [**where** *?′a* = *nat*]
lemmas *Lcm-nat-set-eq-fold* [*code*] = *Lcm-set-eq-fold* [**where** *?′a* = *nat*]
lemmas *Gcd-int-set-eq-fold* [*code*] = *Gcd-set-eq-fold* [**where** *?′a* = *int*]
lemmas *Lcm-int-set-eq-fold* [*code*] = *Lcm-set-eq-fold* [**where** *?′a* = *int*]

Fact aliases.

lemma *lcm-0-iff-nat* [*simp*]: *lcm m n* = *0* ⟷ *m* = *0* ∨ *n* = *0*
for *m n* :: *nat*
⟨*proof*⟩

lemma *lcm-0-iff-int* [*simp*]: *lcm m n* = *0* ⟷ *m* = *0* ∨ *n* = *0*
for *m n* :: *int*
⟨*proof*⟩

lemma *dvd-lcm-I1-nat* [*simp*]: *k dvd m* ⟹ *k dvd lcm m n*
for *k m n* :: *nat*
⟨*proof*⟩

lemma *dvd-lcm-I2-nat* [*simp*]: *k dvd n* ⟹ *k dvd lcm m n*
for *k m n* :: *nat*
⟨*proof*⟩

lemma *dvd-lcm-I1-int* [*simp*]: *i dvd m* ⟹ *i dvd lcm m n*

for *i m n* :: *int*
⟨*proof*⟩

lemma *dvd-lcm-I2-int* [*simp*]: *i dvd n* $\Longrightarrow$ *i dvd lcm m n*
for *i m n* :: *int*
⟨*proof*⟩

lemma *coprime-exp2-nat* [*intro*]: *coprime a b* $\Longrightarrow$ *coprime* (*a* ^*n*) (*b* ^*m*)
for *a b* :: *nat*
⟨*proof*⟩

lemma *coprime-exp2-int* [*intro*]: *coprime a b* $\Longrightarrow$ *coprime* (*a* ^*n*) (*b* ^*m*)
for *a b* :: *int*
⟨*proof*⟩

lemmas *Gcd-dvd-nat* [*simp*] = *Gcd-dvd* [**where** *?'a* = *nat*]
lemmas *Gcd-dvd-int* [*simp*] = *Gcd-dvd* [**where** *?'a* = *int*]
lemmas *Gcd-greatest-nat* [*simp*] = *Gcd-greatest* [**where** *?'a* = *nat*]
lemmas *Gcd-greatest-int* [*simp*] = *Gcd-greatest* [**where** *?'a* = *int*]

lemma *dvd-Lcm-int* [*simp*]: *m* ∈ *M* $\Longrightarrow$ *m dvd Lcm M*
for *M* :: *int set*
⟨*proof*⟩

lemma *gcd-neg-numeral-1-int* [*simp*]: *gcd* (− *numeral n* :: *int*) *x* = *gcd* (*numeral n*) *x*
⟨*proof*⟩

lemma *gcd-neg-numeral-2-int* [*simp*]: *gcd x* (− *numeral n* :: *int*) = *gcd x* (*numeral n*)
⟨*proof*⟩

lemma *gcd-proj1-if-dvd-nat* [*simp*]: *x dvd y* $\Longrightarrow$ *gcd x y* = *x*
for *x y* :: *nat*
⟨*proof*⟩

lemma *gcd-proj2-if-dvd-nat* [*simp*]: *y dvd x* $\Longrightarrow$ *gcd x y* = *y*
for *x y* :: *nat*
⟨*proof*⟩

lemmas *Lcm-eq-0-I-nat* [*simp*] = *Lcm-eq-0-I* [**where** *?'a* = *nat*]
lemmas *Lcm-0-iff-nat* [*simp*] = *Lcm-0-iff* [**where** *?'a* = *nat*]
lemmas *Lcm-least-int* [*simp*] = *Lcm-least* [**where** *?'a* = *int*]

end

87 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

theory *Nitpick*
imports *Record GCD*
keywords
  nitpick :: *diag* **and**
  nitpick-params :: *thy-decl*
begin

datatype (*plugins only*: *extraction*) (*dead 'a*, *dead 'b*) *fun-box* = *FunBox 'a* ⇒ *'b*
datatype (*plugins only*: *extraction*) (*dead 'a*, *dead 'b*) *pair-box* = *PairBox 'a 'b*
datatype (*plugins only*: *extraction*) (*dead 'a*) *word* = *Word 'a set*

typedecl *bisim-iterator*
typedecl *unsigned-bit*
typedecl *signed-bit*

consts
  unknown :: *'a*
  is-unknown :: *'a* ⇒ *bool*
  bisim :: *bisim-iterator* ⇒ *'a* ⇒ *'a* ⇒ *bool*
  bisim-iterator-max :: *bisim-iterator*
  Quot :: *'a* ⇒ *'b*
  safe-The :: (*'a* ⇒ *bool*) ⇒ *'a*

Alternative definitions.

lemma *Ex1-unfold*[*nitpick-unfold*]: *Ex1 P* ≡ ∃ *x*. {*x*. *P x*} = {*x*}
  ⟨*proof*⟩

lemma *rtrancl-unfold*[*nitpick-unfold*]: $r^* \equiv (r^+)^=$
  ⟨*proof*⟩

lemma *rtranclp-unfold*[*nitpick-unfold*]: *rtranclp r a b* ≡ (*a* = *b* ∨ *tranclp r a b*)
  ⟨*proof*⟩

lemma *tranclp-unfold*[*nitpick-unfold*]:
  tranclp r a b ≡ (*a*, *b*) ∈ *trancl* {(*x*, *y*). *r x y*}
  ⟨*proof*⟩

lemma [*nitpick-simp*]:
  of-nat n = (*if n* = *0 then 0 else 1* + *of-nat* (*n* − *1*))
  ⟨*proof*⟩

definition *prod* :: *'a set* ⇒ *'b set* ⇒ (*'a* × *'b*) *set* **where**
  prod A B = {(*a*, *b*). *a* ∈ *A* ∧ *b* ∈ *B*}

definition *refl'* :: (*'a* × *'a*) *set* ⇒ *bool* **where**
  refl' r ≡ ∀ *x*. (*x*, *x*) ∈ *r*

definition wf' :: $('a \times 'a)$ *set* $\Rightarrow$ *bool* **where**
$wf'\ r \equiv acyclic\ r \wedge (finite\ r \vee unknown)$

definition $card'$ :: $'a$ *set* $\Rightarrow$ *nat* **where**
$card'\ A \equiv if\ finite\ A\ then\ length\ (SOME\ xs.\ set\ xs = A \wedge distinct\ xs)\ else\ 0$

definition sum' :: $('a \Rightarrow 'b{::}comm\text{-}monoid\text{-}add) \Rightarrow 'a\ set \Rightarrow 'b$ **where**
$sum'\ f\ A \equiv if\ finite\ A\ then\ sum\text{-}list\ (map\ f\ (SOME\ xs.\ set\ xs = A \wedge distinct\ xs))\ else\ 0$

inductive $fold\text{-}graph'$ :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$ **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 *specialize* optimization.

lemma *The-psimp*[*nitpick-psimp*]: $P = (op =)\ x \Longrightarrow The\ P = x$
⟨*proof*⟩

lemma *Eps-psimp*[*nitpick-psimp*]:
$[\![P\ x;\ \neg\ P\ y;\ Eps\ P = y]\!] \Longrightarrow Eps\ P = x$
⟨*proof*⟩

lemma *case-unit-unfold*[*nitpick-unfold*]:
$case\text{-}unit\ x\ u \equiv x$
⟨*proof*⟩

declare *unit.case*[*nitpick-simp del*]

lemma *case-nat-unfold*[*nitpick-unfold*]:
$case\text{-}nat\ x\ f\ n \equiv if\ n = 0\ then\ x\ else\ f\ (n - 1)$
⟨*proof*⟩

declare *nat.case*[*nitpick-simp del*]

lemma *size-list-simp*[*nitpick-simp*]:
$size\text{-}list\ f\ xs = (if\ xs = []\ then\ 0\ else\ Suc\ (f\ (hd\ xs) + size\text{-}list\ f\ (tl\ xs)))$
$size\ xs = (if\ xs = []\ then\ 0\ else\ Suc\ (size\ (tl\ xs)))$
⟨*proof*⟩

Auxiliary definitions used to provide an alternative representation for *rat* and *real*.

fun *nat-gcd* :: $nat \Rightarrow nat \Rightarrow nat$ **where**
$nat\text{-}gcd\ x\ y = (if\ y = 0\ then\ x\ else\ nat\text{-}gcd\ y\ (x\ mod\ y))$

declare *nat-gcd.simps* [*simp del*]

definition *nat-lcm* :: $nat \Rightarrow nat \Rightarrow nat$ **where**

nat-lcm $x\ y = x * y$ *div* (*nat-gcd* $x\ y$)

lemma *gcd-eq-nitpick-gcd* [*nitpick-unfold*]:
gcd $x\ y$ = *Nitpick.nat-gcd* $x\ y$
⟨*proof*⟩

lemma *lcm-eq-nitpick-lcm* [*nitpick-unfold*]:
lcm $x\ y$ = *Nitpick.nat-lcm* $x\ y$
⟨*proof*⟩

definition *Frac* :: *int* × *int* ⇒ *bool* **where**
Frac ≡ λ(*a*, *b*). *b* > *0* ∧ *gcd a b* = *1*

consts
Abs-Frac :: *int* × *int* ⇒ *'a*
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**
norm-frac a b =
(*if b* < *0 then norm-frac* (− *a*) (− *b*)
else if a = *0* ∨ *b* = *0 then* (*0*, *1*)
else let c = *gcd a b in* (*a div c*, *b div c*))
⟨*proof*⟩
termination ⟨*proof*⟩

declare *norm-frac.simps*[*simp del*]

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* = *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* :: $'a \Rightarrow 'a$ **where**
uminus-frac $q \equiv$ *Abs-Frac* $(-$ *num* q, *denom* $q)$

definition *number-of-frac* :: *int* $\Rightarrow 'a$ **where**
number-of-frac $n \equiv$ *Abs-Frac* $(n,\ 1)$

definition *inverse-frac* :: $'a \Rightarrow 'a$ **where**
inverse-frac $q \equiv$ *frac* (*denom* q) (*num* q)

definition *less-frac* :: $'a \Rightarrow 'a \Rightarrow$ *bool* **where**
[*nitpick-simp*]: *less-frac* $q\ r \longleftrightarrow$ *num* (*plus-frac* q (*uminus-frac* r)) < 0

definition *less-eq-frac* :: $'a \Rightarrow 'a \Rightarrow$ *bool* **where**
[*nitpick-simp*]: *less-eq-frac* $q\ r \longleftrightarrow$ *num* (*plus-frac* q (*uminus-frac* r)) ≤ 0

definition *of-frac* :: $'a \Rightarrow 'b$::{*inverse*,*ring-1*} **where**
of-frac $q \equiv$ *of-int* (*num* q) / *of-int* (*denom* q)

axiomatization *wf-wfrec* :: $('a \times 'a)$ *set* $\Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

definition *wf-wfrec'* :: $('a \times 'a)$ *set* $\Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
[*nitpick-simp*]: *wf-wfrec'* $R\ F\ x = F$ (*cut* (*wf-wfrec* $R\ F$) $R\ x$) x

definition *wfrec'* :: $('a \times 'a)$ *set* $\Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
wfrec' $R\ F\ x \equiv$ *if wf* R *then wf-wfrec'* $R\ F\ x$ *else THE* y. *wfrec-rel* R ($\lambda f\ x.\ F$ (*cut* $f\ R\ x$) x) $x\ y$

⟨*ML*⟩

hide-const (**open**) *unknown is-unknown bisim bisim-iterator-max Quot safe-The FunBox PairBox Word prod*
refl' wf' card' sum' fold-graph' nat-gcd nat-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-frac less-eq-frac of-frac wf-wfrec wf-wfrec wfrec'

hide-type (**open**) *bisim-iterator fun-box pair-box unsigned-bit signed-bit word*

hide-fact (**open**) *Ex1-unfold rtrancl-unfold rtranclp-unfold tranclp-unfold prod-def refl'-def wf'-def*
card'-def sum'-def The-psimp Eps-psimp case-unit-unfold case-nat-unfold
size-list-simp nat-lcm-def Frac-def zero-frac-def one-frac-def
num-def denom-def frac-def plus-frac-def times-frac-def uminus-frac-def
number-of-frac-def inverse-frac-def less-frac-def less-eq-frac-def of-frac-def wf-wfrec'-def
wfrec'-def

end

theory *Nunchaku*
imports *Nitpick*
keywords
nunchaku :: *diag* **and**
nunchaku-params :: *thy-decl*
begin

consts *unreachable* :: $'a$

definition *The-unsafe* :: $('a \Rightarrow bool) \Rightarrow 'a$ **where**
The-unsafe = *The*

definition *rmember* :: $'a\ set \Rightarrow 'a \Rightarrow bool$ **where**
$rmember\ A\ x \longleftrightarrow x \in A$

⟨*ML*⟩

hide-const (**open**) *unreachable The-unsafe rmember*

end

88 Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors

theory *BNF-Greatest-Fixpoint*
imports *BNF-Fixpoint-Base String*
keywords
codatatype :: *thy-decl* **and**
primcorecursive :: *thy-goal* **and**
primcorec :: *thy-decl*
begin

alias *proj* = *Equiv-Relations.proj*

lemma *one-pointE*: $[\![\bigwedge x.\ s = x \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *obj-sumE*: $[\![\forall x.\ s = Inl\ x \longrightarrow P; \forall x.\ s = Inr\ x \longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *not-TrueE*: $\neg\ True \Longrightarrow P$
⟨*proof*⟩

lemma *neq-eq-eq-contradict*: $[\![t \neq u;\ s = t;\ s = u]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *converse-Times*: $(A \times B)\ \hat{}{-1} = B \times A$
⟨*proof*⟩

lemma *equiv-proj*:
  assumes e: *equiv* A R **and** m: $z \in R$
  shows (*proj* R *o fst*) z = (*proj* R *o snd*) z
⟨*proof*⟩

definition *image2* **where** *image2* A f g = $\{(f\ a,\ g\ a) \mid a.\ a \in A\}$

lemma *Id-on-Gr*: *Id-on* A = *Gr* A *id*
  ⟨*proof*⟩

lemma *image2-eqI*: $[\![b = f\ x;\ c = g\ x;\ x \in A]\!] \Longrightarrow (b,\ c) \in$ *image2* $A\ f\ g$
  ⟨*proof*⟩

lemma *IdD*: $(a,\ b) \in Id \Longrightarrow a = b$
  ⟨*proof*⟩

lemma *image2-Gr*: *image2* A f g = (*Gr* A f) ^−*1* *O* (*Gr* A g)
  ⟨*proof*⟩

lemma *GrD1*: $(x,\ fx) \in$ *Gr* $A\ f \Longrightarrow x \in A$
  ⟨*proof*⟩

lemma *GrD2*: $(x,\ fx) \in$ *Gr* $A\ f \Longrightarrow f\ x = fx$
  ⟨*proof*⟩

lemma *Gr-incl*: *Gr* $A\ f \subseteq A \times B \longleftrightarrow f$ ` $A \subseteq B$
  ⟨*proof*⟩

lemma *subset-Collect-iff*: $B \subseteq A \Longrightarrow (B \subseteq \{x \in A.\ P\ x\}) = (\forall\, x \in B.\ P\ x)$
  ⟨*proof*⟩

lemma *subset-CollectI*: $B \subseteq A \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow Q\ x \Longrightarrow P\ x) \Longrightarrow (\{x \in B.\ Q\ x\} \subseteq \{x \in A.\ P\ x\})$
  ⟨*proof*⟩

lemma *in-rel-Collect-case-prod-eq*: *in-rel* (*Collect* (*case-prod* X)) = X
  ⟨*proof*⟩

lemma *Collect-case-prod-in-rel-leI*: $X \subseteq Y \Longrightarrow X \subseteq$ *Collect* (*case-prod* (*in-rel* Y))
  ⟨*proof*⟩

lemma *Collect-case-prod-in-rel-leE*: $X \subseteq$ *Collect* (*case-prod* (*in-rel* Y)) $\Longrightarrow (X \subseteq Y \Longrightarrow R) \Longrightarrow R$
  ⟨*proof*⟩

lemma *conversep-in-rel*: (*in-rel* $R)^{-1-1}$ = *in-rel* (R^{-1})

⟨*proof*⟩

lemma *relcompp-in-rel*: *in-rel R OO in-rel S* = *in-rel* (*R O S*)
⟨*proof*⟩

lemma *in-rel-Gr*: *in-rel* (*Gr A f*) = *Grp A f*
⟨*proof*⟩

definition *relImage* **where**
relImage R f ≡ {(*f a1*, *f a2*) | *a1 a2*. (*a1*,*a2*) ∈ *R*}

definition *relInvImage* **where**
relInvImage A R f ≡ {(*a1*, *a2*) | *a1 a2*. *a1* ∈ *A* ∧ *a2* ∈ *A* ∧ (*f a1*, *f a2*) ∈ *R*}

lemma *relImage-Gr*:
⟦*R* ⊆ *A* × *A*⟧ ⟹ *relImage R f* = (*Gr A f*)ˆ−*1 O R O Gr A f*
⟨*proof*⟩

lemma *relInvImage-Gr*: ⟦*R* ⊆ *B* × *B*⟧ ⟹ *relInvImage A R f* = *Gr A f O R O* (*Gr A f*)ˆ−*1*
⟨*proof*⟩

lemma *relImage-mono*:
R1 ⊆ *R2* ⟹ *relImage R1 f* ⊆ *relImage R2 f*
⟨*proof*⟩

lemma *relInvImage-mono*:
R1 ⊆ *R2* ⟹ *relInvImage A R1 f* ⊆ *relInvImage A R2 f*
⟨*proof*⟩

lemma *relInvImage-Id-on*:
(⋀*a1 a2*. *f a1* = *f a2* ⟷ *a1* = *a2*) ⟹ *relInvImage A* (*Id-on B*) *f* ⊆ *Id*
⟨*proof*⟩

lemma *relInvImage-UNIV-relImage*:
R ⊆ *relInvImage UNIV* (*relImage R f*) *f*
⟨*proof*⟩

lemma *relImage-proj*:
assumes *equiv A R*
shows *relImage R* (*proj R*) ⊆ *Id-on* (*A*//*R*)
⟨*proof*⟩

lemma *relImage-relInvImage*:
assumes *R* ⊆ *f* ‘ *A* × *f* ‘ *A*
shows *relImage* (*relInvImage A R f*) *f* = *R*
⟨*proof*⟩

lemma *subst-Pair*: *P x y* ⟹ *a* = (*x*, *y*) ⟹ *P* (*fst a*) (*snd a*)

⟨*proof*⟩

lemma *fst-diag-id*: $(fst \circ (\lambda x.\ (x,\ x)))\ z = id\ z$ ⟨*proof*⟩
lemma *snd-diag-id*: $(snd \circ (\lambda x.\ (x,\ x)))\ z = id\ z$ ⟨*proof*⟩

lemma *fst-diag-fst*: $fst\ o\ ((\lambda x.\ (x,\ x))\ o\ fst) = fst$ ⟨*proof*⟩
lemma *snd-diag-fst*: $snd\ o\ ((\lambda x.\ (x,\ x))\ o\ fst) = fst$ ⟨*proof*⟩
lemma *fst-diag-snd*: $fst\ o\ ((\lambda x.\ (x,\ x))\ o\ snd) = snd$ ⟨*proof*⟩
lemma *snd-diag-snd*: $snd\ o\ ((\lambda x.\ (x,\ x))\ o\ snd) = snd$ ⟨*proof*⟩

definition *Succ* **where** $Succ\ Kl\ kl = \{k\ .\ kl\ @\ [k] \in Kl\}$
definition *Shift* **where** $Shift\ Kl\ k = \{kl.\ k\ \#\ kl \in Kl\}$
definition *shift* **where** $shift\ lab\ k = (\lambda kl.\ lab\ (k\ \#\ kl))$

lemma *empty-Shift*: $[\![[] \in Kl;\ k \in Succ\ Kl\ []]\!] \Longrightarrow [] \in Shift\ Kl\ k$
⟨*proof*⟩

lemma *SuccD*: $k \in Succ\ Kl\ kl \Longrightarrow kl\ @\ [k] \in Kl$
⟨*proof*⟩

lemmas $SuccE = SuccD[elim\text{-}format]$

lemma *SuccI*: $kl\ @\ [k] \in Kl \Longrightarrow k \in Succ\ Kl\ kl$
⟨*proof*⟩

lemma *ShiftD*: $kl \in Shift\ Kl\ k \Longrightarrow k\ \#\ kl \in Kl$
⟨*proof*⟩

lemma *Succ-Shift*: $Succ\ (Shift\ Kl\ k)\ kl = Succ\ Kl\ (k\ \#\ kl)$
⟨*proof*⟩

lemma *length-Cons*: $length\ (x\ \#\ xs) = Suc\ (length\ xs)$
⟨*proof*⟩

lemma *length-append-singleton*: $length\ (xs\ @\ [x]) = Suc\ (length\ xs)$
⟨*proof*⟩

definition *toCard-pred* $A\ r\ f \equiv inj\text{-}on\ f\ A \wedge f\ `\ A \subseteq Field\ r \wedge Card\text{-}order\ r$
definition *toCard* $A\ r \equiv SOME\ f.\ toCard\text{-}pred\ A\ r\ f$

lemma *ex-toCard-pred*:
$[\![|A| \le o\ r;\ Card\text{-}order\ r]\!] \Longrightarrow \exists\ f.\ toCard\text{-}pred\ A\ r\ f$
⟨*proof*⟩

lemma *toCard-pred-toCard*:
$[\![|A| \le o\ r;\ Card\text{-}order\ r]\!] \Longrightarrow toCard\text{-}pred\ A\ r\ (toCard\ A\ r)$
⟨*proof*⟩

lemma *toCard-inj*: $[\![|A| \leq o\ r;\ Card\text{-}order\ r;\ x \in A;\ y \in A]\!] \Longrightarrow toCard\ A\ r\ x = toCard\ A\ r\ y \longleftrightarrow x = y$
⟨*proof*⟩

definition *fromCard* $A\ r\ k \equiv SOME\ b.\ b \in A \wedge toCard\ A\ r\ b = k$

lemma *fromCard-toCard*:
$[\![|A| \leq o\ r;\ Card\text{-}order\ r;\ b \in A]\!] \Longrightarrow fromCard\ A\ r\ (toCard\ A\ r\ b) = b$
⟨*proof*⟩

lemma *Inl-Field-csum*: $a \in Field\ r \Longrightarrow Inl\ a \in Field\ (r +c\ s)$
⟨*proof*⟩

lemma *Inr-Field-csum*: $a \in Field\ s \Longrightarrow Inr\ a \in Field\ (r +c\ s)$
⟨*proof*⟩

lemma *rec-nat-0-imp*: $f = rec\text{-}nat\ f1\ (\lambda n\ rec.\ f2\ n\ rec) \Longrightarrow f\ 0 = f1$
⟨*proof*⟩

lemma *rec-nat-Suc-imp*: $f = rec\text{-}nat\ f1\ (\lambda n\ rec.\ f2\ n\ rec) \Longrightarrow f\ (Suc\ n) = f2\ n\ (f\ n)$
⟨*proof*⟩

lemma *rec-list-Nil-imp*: $f = rec\text{-}list\ f1\ (\lambda x\ xs\ rec.\ f2\ x\ xs\ rec) \Longrightarrow f\ [] = f1$
⟨*proof*⟩

lemma *rec-list-Cons-imp*: $f = rec\text{-}list\ f1\ (\lambda x\ xs\ rec.\ f2\ x\ xs\ rec) \Longrightarrow f\ (x\ \#\ xs) = f2\ x\ xs\ (f\ xs)$
⟨*proof*⟩

lemma *not-arg-cong-Inr*: $x \neq y \Longrightarrow Inr\ x \neq Inr\ y$
⟨*proof*⟩

definition *image2p* **where**
$image2p\ f\ g\ R = (\lambda x\ y.\ \exists x'\ y'.\ R\ x'\ y' \wedge f\ x' = x \wedge g\ y' = y)$

lemma *image2pI*: $R\ x\ y \Longrightarrow image2p\ f\ g\ R\ (f\ x)\ (g\ y)$
⟨*proof*⟩

lemma *image2pE*: $[\![image2p\ f\ g\ R\ fx\ gy;\ (\bigwedge x\ y.\ fx = f\ x \Longrightarrow gy = g\ y \Longrightarrow R\ x\ y \Longrightarrow P)]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *rel-fun-iff-geq-image2p*: $rel\text{-}fun\ R\ S\ f\ g = (image2p\ f\ g\ R \leq S)$
⟨*proof*⟩

lemma *rel-fun-image2p*: $rel\text{-}fun\ R\ (image2p\ f\ g\ R)\ f\ g$
⟨*proof*⟩

88.1 Equivalence relations, quotients, and Hilbert's choice

lemma *equiv-Eps-in*:
$[\![equiv\ A\ r;\ X \in A//r]\!] \Longrightarrow Eps\ (\lambda x.\ x \in X) \in X$
⟨*proof*⟩

lemma *equiv-Eps-preserves*:
assumes *ECH*: *equiv A r* **and** *X*: $X \in A//r$
shows $Eps\ (\lambda x.\ x \in X) \in A$
⟨*proof*⟩

lemma *proj-Eps*:
assumes *equiv A r* **and** $X \in A//r$
shows $proj\ r\ (Eps\ (\lambda x.\ x \in X)) = X$
⟨*proof*⟩

definition *univ* **where** $univ\ f\ X == f\ (Eps\ (\lambda x.\ x \in X))$

lemma *univ-commute*:
assumes *ECH*: *equiv A r* **and** *RES*: *f respects r* **and** *x*: $x \in A$
shows $(univ\ f)\ (proj\ r\ x) = f\ x$
⟨*proof*⟩

lemma *univ-preserves*:
assumes *ECH*: *equiv A r* **and** *RES*: *f respects r* **and** *PRES*: $\forall x \in A.\ f\ x \in B$
shows $\forall X \in A//r.\ univ\ f\ X \in B$
⟨*proof*⟩

⟨*ML*⟩

end

89 Filters on predicates

theory *Filter*
imports *Set-Interval Lifting-Set*
begin

89.1 Filters

This definition also allows non-proper filters.

locale *is-filter* =
fixes $F :: ('a \Rightarrow bool) \Rightarrow bool$
assumes *True*: $F\ (\lambda x.\ True)$
assumes *conj*: $F\ (\lambda x.\ P\ x) \Longrightarrow F\ (\lambda x.\ Q\ x) \Longrightarrow F\ (\lambda x.\ P\ x \wedge Q\ x)$
assumes *mono*: $\forall x.\ P\ x \longrightarrow Q\ x \Longrightarrow F\ (\lambda x.\ P\ x) \Longrightarrow F\ (\lambda x.\ Q\ x)$

typedef $'a\ filter = \{F :: ('a \Rightarrow bool) \Rightarrow bool.\ is\text{-}filter\ F\}$
⟨*proof*⟩

lemma *is-filter-Rep-filter*: *is-filter* (*Rep-filter F*)
⟨*proof*⟩

lemma *Abs-filter-inverse'*:
assumes *is-filter F* **shows** *Rep-filter* (*Abs-filter F*) = *F*
⟨*proof*⟩

89.1.1 Eventually

definition *eventually* :: ('*a* ⇒ *bool*) ⇒ '*a filter* ⇒ *bool*
where *eventually P F* ⟷ *Rep-filter F P*

syntax
-eventually :: *pttrn* => '*a filter* => *bool* => *bool* ((*3*∀$_F$ - *in* -./ -) [*0*, *0*, *10*] *10*)
translations
∀$_F$*x in F*. *P* == *CONST eventually* (λ*x*. *P*) *F*

lemma *eventually-Abs-filter*:
assumes *is-filter F* **shows** *eventually P* (*Abs-filter F*) = *F P*
⟨*proof*⟩

lemma *filter-eq-iff*:
shows *F* = *F'* ⟷ (∀ *P*. *eventually P F* = *eventually P F'*)
⟨*proof*⟩

lemma *eventually-True* [*simp*]: *eventually* (λ*x*. *True*) *F*
⟨*proof*⟩

lemma *always-eventually*: ∀ *x*. *P x* ⟹ *eventually P F*
⟨*proof*⟩

lemma *eventuallyI*: (⋀*x*. *P x*) ⟹ *eventually P F*
⟨*proof*⟩

lemma *eventually-mono*:
⟦*eventually P F*; ⋀*x*. *P x* ⟹ *Q x*⟧ ⟹ *eventually Q F*
⟨*proof*⟩

lemma *eventually-conj*:
assumes *P*: *eventually* (λ*x*. *P x*) *F*
assumes *Q*: *eventually* (λ*x*. *Q x*) *F*
shows *eventually* (λ*x*. *P x* ∧ *Q x*) *F*
⟨*proof*⟩

lemma *eventually-mp*:
assumes *eventually* (λ*x*. *P x* ⟶ *Q x*) *F*
assumes *eventually* (λ*x*. *P x*) *F*

shows *eventually* $(\lambda x.\ Q\ x)\ F$
⟨*proof*⟩

lemma *eventually-rev-mp*:
assumes *eventually* $(\lambda x.\ P\ x)\ F$
assumes *eventually* $(\lambda x.\ P\ x \longrightarrow Q\ x)\ F$
shows *eventually* $(\lambda x.\ Q\ x)\ F$
⟨*proof*⟩

lemma *eventually-conj-iff*:
eventually $(\lambda x.\ P\ x \wedge Q\ x)\ F \longleftrightarrow$ *eventually* $P\ F \wedge$ *eventually* $Q\ F$
⟨*proof*⟩

lemma *eventually-elim2*:
assumes *eventually* $(\lambda i.\ P\ i)\ F$
assumes *eventually* $(\lambda i.\ Q\ i)\ F$
assumes $\bigwedge i.\ P\ i \Longrightarrow Q\ i \Longrightarrow R\ i$
shows *eventually* $(\lambda i.\ R\ i)\ F$
⟨*proof*⟩

lemma *eventually-ball-finite-distrib*:
finite $A \Longrightarrow$ (*eventually* $(\lambda x.\ \forall y{\in}A.\ P\ x\ y)$ *net*) $\longleftrightarrow$ $(\forall y{\in}A.$ *eventually* $(\lambda x.\ P\ x\ y)$ *net*)
⟨*proof*⟩

lemma *eventually-ball-finite*:
finite $A \Longrightarrow \forall y{\in}A.$ *eventually* $(\lambda x.\ P\ x\ y)$ *net* $\Longrightarrow$ *eventually* $(\lambda x.\ \forall y{\in}A.\ P\ x\ y)$ *net*
⟨*proof*⟩

lemma *eventually-all-finite*:
fixes $P :: 'a \Rightarrow 'b{::}finite \Rightarrow bool$
assumes $\bigwedge y.$ *eventually* $(\lambda x.\ P\ x\ y)$ *net*
shows *eventually* $(\lambda x.\ \forall y.\ P\ x\ y)$ *net*
⟨*proof*⟩

lemma *eventually-ex*: $(\forall_F x\ in\ F.\ \exists y.\ P\ x\ y) \longleftrightarrow (\exists Y.\ \forall_F x\ in\ F.\ P\ x\ (Y\ x))$
⟨*proof*⟩

lemma *not-eventually-impI*: *eventually* $P\ F \Longrightarrow \neg$ *eventually* $Q\ F \Longrightarrow \neg$ *eventually* $(\lambda x.\ P\ x \longrightarrow Q\ x)\ F$
⟨*proof*⟩

lemma *not-eventuallyD*: $\neg$ *eventually* $P\ F \Longrightarrow \exists x.\ \neg\ P\ x$
⟨*proof*⟩

lemma *eventually-subst*:
assumes *eventually* $(\lambda n.\ P\ n = Q\ n)\ F$
shows *eventually* $P\ F =$ *eventually* $Q\ F$ (**is** $?L = ?R$)

⟨*proof*⟩

89.2 Frequently as dual to eventually

definition *frequently* :: $('a \Rightarrow bool) \Rightarrow 'a\ filter \Rightarrow bool$
where *frequently* $P\ F \longleftrightarrow \neg$ *eventually* $(\lambda x.\ \neg\ P\ x)\ F$

syntax
-frequently :: $pttrn \Rightarrow 'a\ filter \Rightarrow bool \Rightarrow bool$ ((3∃$_F$ - in -./ -) [0, 0, 10] 10)
translations
$\exists_F x$ *in* $F.\ P$ == *CONST frequently* $(\lambda x.\ P)\ F$

lemma *not-frequently-False* [*simp*]: $\neg\ (\exists_F x$ *in* $F.$ *False*)
⟨*proof*⟩

lemma *frequently-ex*: $\exists_F x$ *in* $F.\ P\ x \Longrightarrow \exists x.\ P\ x$
⟨*proof*⟩

lemma *frequentlyE*: **assumes** *frequently* $P\ F$ **obtains** x **where** $P\ x$
⟨*proof*⟩

lemma *frequently-mp*:
assumes *ev*: $\forall_F x$ *in* $F.\ P\ x \longrightarrow Q\ x$ **and** P: $\exists_F x$ *in* $F.\ P\ x$ **shows** $\exists_F x$ *in* $F.$ $Q\ x$
⟨*proof*⟩

lemma *frequently-rev-mp*:
assumes $\exists_F x$ *in* $F.\ P\ x$
assumes $\forall_F x$ *in* $F.\ P\ x \longrightarrow Q\ x$
shows $\exists_F x$ *in* $F.\ Q\ x$
⟨*proof*⟩

lemma *frequently-mono*: $(\forall x.\ P\ x \longrightarrow Q\ x) \Longrightarrow$ *frequently* $P\ F \Longrightarrow$ *frequently* Q F
⟨*proof*⟩

lemma *frequently-elim1*: $\exists_F x$ *in* $F.\ P\ x \Longrightarrow (\bigwedge i.\ P\ i \Longrightarrow Q\ i) \Longrightarrow \exists_F x$ *in* $F.\ Q$ x
⟨*proof*⟩

lemma *frequently-disj-iff*: $(\exists_F x$ *in* $F.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_F x$ *in* $F.\ P\ x) \vee (\exists_F x$ *in* $F.\ Q\ x)$
⟨*proof*⟩

lemma *frequently-disj*: $\exists_F x$ *in* $F.\ P\ x \Longrightarrow \exists_F x$ *in* $F.\ Q\ x \Longrightarrow \exists_F x$ *in* $F.\ P\ x \vee$ $Q\ x$
⟨*proof*⟩

lemma *frequently-bex-finite-distrib*:

assumes *finite A* **shows** $(\exists_F x\ in\ F.\ \exists y{\in}A.\ P\ x\ y) \longleftrightarrow (\exists y{\in}A.\ \exists_F x\ in\ F.\ P\ x\ y)$
⟨*proof*⟩

lemma *frequently-bex-finite*: $finite\ A \Longrightarrow \exists_F x\ in\ F.\ \exists y{\in}A.\ P\ x\ y \Longrightarrow \exists y{\in}A.\ \exists_F x\ in\ F.\ P\ x\ y$
⟨*proof*⟩

lemma *frequently-all*: $(\exists_F x\ in\ F.\ \forall y.\ P\ x\ y) \longleftrightarrow (\forall Y.\ \exists_F x\ in\ F.\ P\ x\ (Y\ x))$
⟨*proof*⟩

lemma
shows *not-eventually*: $\neg\ eventually\ P\ F \longleftrightarrow (\exists_F x\ in\ F.\ \neg\ P\ x)$
and *not-frequently*: $\neg\ frequently\ P\ F \longleftrightarrow (\forall_F x\ in\ F.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *frequently-imp-iff*:
$(\exists_F x\ in\ F.\ P\ x \longrightarrow Q\ x) \longleftrightarrow (eventually\ P\ F \longrightarrow frequently\ Q\ F)$
⟨*proof*⟩

lemma *eventually-frequently-const-simps*:
$(\exists_F x\ in\ F.\ P\ x \wedge C) \longleftrightarrow (\exists_F x\ in\ F.\ P\ x) \wedge C$
$(\exists_F x\ in\ F.\ C \wedge P\ x) \longleftrightarrow C \wedge (\exists_F x\ in\ F.\ P\ x)$
$(\forall_F x\ in\ F.\ P\ x \vee C) \longleftrightarrow (\forall_F x\ in\ F.\ P\ x) \vee C$
$(\forall_F x\ in\ F.\ C \vee P\ x) \longleftrightarrow C \vee (\forall_F x\ in\ F.\ P\ x)$
$(\forall_F x\ in\ F.\ P\ x \longrightarrow C) \longleftrightarrow ((\exists_F x\ in\ F.\ P\ x) \longrightarrow C)$
$(\forall_F x\ in\ F.\ C \longrightarrow P\ x) \longleftrightarrow (C \longrightarrow (\forall_F x\ in\ F.\ P\ x))$
⟨*proof*⟩

lemmas *eventually-frequently-simps* =
eventually-frequently-const-simps
not-eventually
eventually-conj-iff
eventually-ball-finite-distrib
eventually-ex
not-frequently
frequently-disj-iff
frequently-bex-finite-distrib
frequently-all
frequently-imp-iff

⟨*ML*⟩

89.2.1 Finer-than relation

$F \le F'$ means that filter F is finer than filter F'.

instantiation *filter* :: (*type*) *complete-lattice*
begin

definition *le-filter-def*:
$F \le F' \longleftrightarrow (\forall P.\ eventually\ P\ F' \longrightarrow eventually\ P\ F)$

definition
$(F :: 'a\ filter) < F' \longleftrightarrow F \le F' \wedge \neg\ F' \le F$

definition
$top = Abs\text{-}filter\ (\lambda P.\ \forall x.\ P\ x)$

definition
$bot = Abs\text{-}filter\ (\lambda P.\ True)$

definition
$sup\ F\ F' = Abs\text{-}filter\ (\lambda P.\ eventually\ P\ F \wedge eventually\ P\ F')$

definition
$inf\ F\ F' = Abs\text{-}filter$
$(\lambda P.\ \exists\ Q\ R.\ eventually\ Q\ F \wedge eventually\ R\ F' \wedge (\forall x.\ Q\ x \wedge R\ x \longrightarrow P\ x))$

definition
$Sup\ S = Abs\text{-}filter\ (\lambda P.\ \forall\ F{\in}S.\ eventually\ P\ F)$

definition
$Inf\ S = Sup\ \{F{::}'a\ filter.\ \forall\ F'{\in}S.\ F \le F'\}$

lemma *eventually-top* [*simp*]: $eventually\ P\ top \longleftrightarrow (\forall x.\ P\ x)$
⟨*proof*⟩

lemma *eventually-bot* [*simp*]: $eventually\ P\ bot$
⟨*proof*⟩

lemma *eventually-sup*:
$eventually\ P\ (sup\ F\ F') \longleftrightarrow eventually\ P\ F \wedge eventually\ P\ F'$
⟨*proof*⟩

lemma *eventually-inf*:
$eventually\ P\ (inf\ F\ F') \longleftrightarrow$
$(\exists\ Q\ R.\ eventually\ Q\ F \wedge eventually\ R\ F' \wedge (\forall x.\ Q\ x \wedge R\ x \longrightarrow P\ x))$
⟨*proof*⟩

lemma *eventually-Sup*:
$eventually\ P\ (Sup\ S) \longleftrightarrow (\forall\ F{\in}S.\ eventually\ P\ F)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *filter* :: (*type*) *distrib-lattice*

⟨*proof*⟩

lemma *filter-leD*:
$F \le F' \Longrightarrow$ *eventually* $P\ F' \Longrightarrow$ *eventually* $P\ F$
⟨*proof*⟩

lemma *filter-leI*:
$(\bigwedge P.$ *eventually* $P\ F' \Longrightarrow$ *eventually* $P\ F) \Longrightarrow F \le F'$
⟨*proof*⟩

lemma *eventually-False*:
eventually $(\lambda x.$ *False*$)\ F \longleftrightarrow F =$ *bot*
⟨*proof*⟩

lemma *eventually-frequently*: $F \neq$ *bot* $\Longrightarrow$ *eventually* $P\ F \Longrightarrow$ *frequently* $P\ F$
⟨*proof*⟩

lemma *eventually-const-iff*: *eventually* $(\lambda x.\ P)\ F \longleftrightarrow P \vee F =$ *bot*
⟨*proof*⟩

lemma *eventually-const*[*simp*]: $F \neq$ *bot* $\Longrightarrow$ *eventually* $(\lambda x.\ P)\ F \longleftrightarrow P$
⟨*proof*⟩

lemma *frequently-const-iff*: *frequently* $(\lambda x.\ P)\ F \longleftrightarrow P \wedge F \neq$ *bot*
⟨*proof*⟩

lemma *frequently-const*[*simp*]: $F \neq$ *bot* $\Longrightarrow$ *frequently* $(\lambda x.\ P)\ F \longleftrightarrow P$
⟨*proof*⟩

lemma *eventually-happens*: *eventually* P *net* $\Longrightarrow$ *net* $=$ *bot* $\vee (\exists\, x.\ P\ x)$
⟨*proof*⟩

lemma *eventually-happens'*:
assumes $F \neq$ *bot* *eventually* $P\ F$
shows $\exists\, x.\ P\ x$
⟨*proof*⟩

abbreviation (*input*) *trivial-limit* :: *'a filter* $\Rightarrow$ *bool*
where *trivial-limit* $F \equiv F =$ *bot*

lemma *trivial-limit-def*: *trivial-limit* $F \longleftrightarrow$ *eventually* $(\lambda x.$ *False*$)\ F$
⟨*proof*⟩

lemma *False-imp-not-eventually*: $(\forall\, x.\ \neg\ P\ x\ ) \Longrightarrow \neg$ *trivial-limit net* $\Longrightarrow \neg$ *eventually* $(\lambda x.\ P\ x)$ *net*
⟨*proof*⟩

lemma *eventually-Inf*: *eventually* P (*Inf* B) $\longleftrightarrow (\exists\, X \subseteq B.$ *finite* $X\ \wedge$ *eventually*

P (*Inf* X))
⟨*proof*⟩

lemma *eventually-INF*: *eventually* P (*INF* $b{:}B$. F b) $\longleftrightarrow$ ($\exists X \subseteq B$. *finite* X $\wedge$ *eventually* P (*INF* $b{:}X$. F b))
⟨*proof*⟩

lemma *Inf-filter-not-bot*:
fixes B :: $'a$ *filter set*
shows ($\bigwedge X$. $X \subseteq B \Longrightarrow$ *finite* $X \Longrightarrow$ *Inf* $X \neq$ *bot*) $\Longrightarrow$ *Inf* $B \neq$ *bot*
⟨*proof*⟩

lemma *INF-filter-not-bot*:
fixes F :: $'i \Rightarrow 'a$ *filter*
shows ($\bigwedge X$. $X \subseteq B \Longrightarrow$ *finite* $X \Longrightarrow$ (*INF* $b{:}X$. F b) $\neq$ *bot*) $\Longrightarrow$ (*INF* $b{:}B$. F b) $\neq$ *bot*
⟨*proof*⟩

lemma *eventually-Inf-base*:
assumes $B \neq \{\}$ **and** *base*: $\bigwedge F$ G. $F \in B \Longrightarrow G \in B \Longrightarrow \exists x{\in}B$. $x \leq$ *inf* F G
shows *eventually* P (*Inf* B) $\longleftrightarrow$ ($\exists b{\in}B$. *eventually* P b)
⟨*proof*⟩

lemma *eventually-INF-base*:
$B \neq \{\} \Longrightarrow (\bigwedge a\ b.\ a \in B \Longrightarrow b \in B \Longrightarrow \exists x{\in}B.\ F\ x \leq$ *inf* $(F\ a)\ (F\ b)) \Longrightarrow$
eventually P (*INF* $b{:}B$. F b) $\longleftrightarrow$ ($\exists b{\in}B$. *eventually* P (F b))
⟨*proof*⟩

lemma *eventually-INF1*: $i \in I \Longrightarrow$ *eventually* P (F i) $\Longrightarrow$ *eventually* P (*INF* $i{:}I$. F i)
⟨*proof*⟩

lemma *eventually-INF-mono*:
assumes $*$: $\forall_F$ x *in* $\bigsqcap i{\in}I$. F i. P x
assumes *T1*: $\bigwedge Q\ R\ P$. ($\bigwedge x$. $Q\ x \wedge R\ x \longrightarrow P\ x$) $\Longrightarrow$ ($\bigwedge x$. $T\ Q\ x \Longrightarrow T\ R\ x \Longrightarrow T\ P\ x$)
assumes *T2*: $\bigwedge P$. ($\bigwedge x$. $P\ x$) $\Longrightarrow$ ($\bigwedge x$. $T\ P\ x$)
assumes $**$: $\bigwedge i\ P$. $i \in I \Longrightarrow \forall_F$ x *in* F i. P $x \Longrightarrow \forall_F$ x *in* F' i. T P x
shows $\forall_F$ x *in* $\bigsqcap i{\in}I$. F' i. T P x
⟨*proof*⟩

89.2.2 Map function for filters

definition *filtermap* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *filter* $\Rightarrow 'b$ *filter*
where *filtermap* f F = *Abs-filter* (λP. *eventually* (λx. P (f x)) F)

lemma *eventually-filtermap*:
eventually P (*filtermap* f F) = *eventually* (λx. P (f x)) F
⟨*proof*⟩

lemma *filtermap-ident*: $\mathit{filtermap}\ (\lambda x.\ x)\ F = F$
⟨*proof*⟩

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

lemma *filtermap-mono*: $F \le F' \Longrightarrow \mathit{filtermap}\ f\ F \le \mathit{filtermap}\ f\ F'$
⟨*proof*⟩

lemma *filtermap-bot* [*simp*]: $\mathit{filtermap}\ f\ \mathit{bot} = \mathit{bot}$
⟨*proof*⟩

lemma *filtermap-sup*: $\mathit{filtermap}\ f\ (\mathit{sup}\ F1\ F2) = \mathit{sup}\ (\mathit{filtermap}\ f\ F1)\ (\mathit{filtermap}\ f\ F2)$
⟨*proof*⟩

lemma *filtermap-inf*: $\mathit{filtermap}\ f\ (\mathit{inf}\ F1\ F2) \le \mathit{inf}\ (\mathit{filtermap}\ f\ F1)\ (\mathit{filtermap}\ f\ F2)$
⟨*proof*⟩

lemma *filtermap-INF*: $\mathit{filtermap}\ f\ (\mathit{INF}\ b{:}B.\ F\ b) \le (\mathit{INF}\ b{:}B.\ \mathit{filtermap}\ f\ (F\ b))$
⟨*proof*⟩

89.2.3 Contravariant map function for filters

definition *filtercomap* :: $('a \Rightarrow 'b) \Rightarrow 'b\ \mathit{filter} \Rightarrow 'a\ \mathit{filter}$ **where**
$\mathit{filtercomap}\ f\ F = \mathit{Abs\text{-}filter}\ (\lambda P.\ \exists\, Q.\ \mathit{eventually}\ Q\ F \wedge (\forall\, x.\ Q\ (f\ x) \longrightarrow P\ x))$

lemma *eventually-filtercomap*:
$\mathit{eventually}\ P\ (\mathit{filtercomap}\ f\ F) \longleftrightarrow (\exists\, Q.\ \mathit{eventually}\ Q\ F \wedge (\forall\, x.\ Q\ (f\ x) \longrightarrow P\ x))$
⟨*proof*⟩

lemma *filtercomap-ident*: $\mathit{filtercomap}\ (\lambda x.\ x)\ F = F$
⟨*proof*⟩

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

lemma *filtercomap-mono*: $F \le F' \Longrightarrow \mathit{filtercomap}\ f\ F \le \mathit{filtercomap}\ f\ F'$
⟨*proof*⟩

lemma *filtercomap-bot* [*simp*]: $\mathit{filtercomap}\ f\ \mathit{bot} = \mathit{bot}$
⟨*proof*⟩

lemma *filtercomap-top* [*simp*]: $\mathit{filtercomap}\ f\ \mathit{top} = \mathit{top}$

⟨*proof*⟩

lemma *filtercomap-inf*: *filtercomap f* (*inf F1 F2*) = *inf* (*filtercomap f F1*) (*filtercomap f F2*)
⟨*proof*⟩

lemma *filtercomap-sup*: *filtercomap f* (*sup F1 F2*) $\geq$ *sup* (*filtercomap f F1*) (*filtercomap f F2*)
⟨*proof*⟩

lemma *filtercomap-INF*: *filtercomap f* (*INF b*:*B*. *F b*) = (*INF b*:*B*. *filtercomap f* (*F b*))
⟨*proof*⟩

lemma *filtercomap-SUP-finite*:
finite B $\Longrightarrow$ *filtercomap f* (*SUP b*:*B*. *F b*) $\geq$ (*SUP b*:*B*. *filtercomap f* (*F b*))
⟨*proof*⟩

lemma *eventually-filtercomapI* [*intro*]:
assumes *eventually P F*
shows *eventually* (λx. *P* (*f x*)) (*filtercomap f F*)
⟨*proof*⟩

lemma *filtermap-filtercomap*: *filtermap f* (*filtercomap f F*) $\leq$ *F*
⟨*proof*⟩

lemma *filtercomap-filtermap*: *filtercomap f* (*filtermap f F*) $\geq$ *F*
⟨*proof*⟩

89.2.4 Standard filters

definition *principal* :: *'a set* $\Rightarrow$ *'a filter* **where**
principal S = *Abs-filter* (λP. $\forall x \in S$. *P x*)

lemma *eventually-principal*: *eventually P* (*principal S*) $\longleftrightarrow$ ($\forall x \in S$. *P x*)
⟨*proof*⟩

lemma *eventually-inf-principal*: *eventually P* (*inf F* (*principal s*)) $\longleftrightarrow$ *eventually* (λx. $x \in s \longrightarrow P\ x$) *F*
⟨*proof*⟩

lemma *principal-UNIV*[*simp*]: *principal UNIV* = *top*
⟨*proof*⟩

lemma *principal-empty*[*simp*]: *principal* {} = *bot*
⟨*proof*⟩

lemma *principal-eq-bot-iff*: *principal X* = *bot* $\longleftrightarrow$ *X* = {}
⟨*proof*⟩

lemma *principal-le-iff* [*iff*]: $principal\ A \leq principal\ B \longleftrightarrow A \subseteq B$
⟨*proof*⟩

lemma *le-principal*: $F \leq principal\ A \longleftrightarrow eventually\ (\lambda x.\ x \in A)\ F$
⟨*proof*⟩

lemma *principal-inject*[*iff*]: $principal\ A = principal\ B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *sup-principal*[*simp*]: $sup\ (principal\ A)\ (principal\ B) = principal\ (A \cup B)$
⟨*proof*⟩

lemma *inf-principal*[*simp*]: $inf\ (principal\ A)\ (principal\ B) = principal\ (A \cap B)$
⟨*proof*⟩

lemma *SUP-principal*[*simp*]: $(SUP\ i : I.\ principal\ (A\ i)) = principal\ (\bigcup i{\in}I.\ A\ i)$
⟨*proof*⟩

lemma *INF-principal-finite*: $finite\ X \Longrightarrow (INF\ x{:}X.\ principal\ (f\ x)) = principal\ (\bigcap x{\in}X.\ f\ x)$
⟨*proof*⟩

lemma *filtermap-principal*[*simp*]: $filtermap\ f\ (principal\ A) = principal\ (f\ `\ A)$
⟨*proof*⟩

lemma *filtercomap-principal*[*simp*]: $filtercomap\ f\ (principal\ A) = principal\ (f\ -`\ A)$
⟨*proof*⟩

89.2.5 Order filters

definition *at-top* :: $('a{::}order)\ filter$
where $at\text{-}top = (INF\ k.\ principal\ \{k\ ..\})$

lemma *at-top-sub*: $at\text{-}top = (INF\ k{:}\{c{::}'a{::}linorder..\}.\ principal\ \{k\ ..\})$
⟨*proof*⟩

lemma *eventually-at-top-linorder*: $eventually\ P\ at\text{-}top \longleftrightarrow (\exists N{::}'a{::}linorder.\ \forall n{\geq}N.\ P\ n)$
⟨*proof*⟩

lemma *eventually-filtercomap-at-top-linorder*:
$eventually\ P\ (filtercomap\ f\ at\text{-}top) \longleftrightarrow (\exists N{::}'a{::}linorder.\ \forall x.\ f\ x \geq N \longrightarrow P\ x)$
⟨*proof*⟩

lemma *eventually-at-top-linorderI*:
fixes $c{::}'a{::}linorder$

assumes $\bigwedge x.\ c \le x \Longrightarrow P\ x$
shows *eventually P at-top*
⟨*proof*⟩

lemma *eventually-ge-at-top* [*simp*]:
eventually ($\lambda x.$ (*c*::-::*linorder*) $\le x$) *at-top*
⟨*proof*⟩

lemma *eventually-at-top-dense*: *eventually P at-top* $\longleftrightarrow$ ($\exists N$::'*a*::{*no-top*, *linorder*}. $\forall n>N.\ P\ n$)
⟨*proof*⟩

lemma *eventually-filtercomap-at-top-dense*:
eventually P (*filtercomap f at-top*) $\longleftrightarrow$ ($\exists N$::'*a*::{*no-top*, *linorder*}. $\forall x.\ f\ x > N \longrightarrow P\ x$)
⟨*proof*⟩

lemma *eventually-at-top-not-equal* [*simp*]: *eventually* (λx::'*a*::{*no-top*, *linorder*}. $x \ne c$) *at-top*
⟨*proof*⟩

lemma *eventually-gt-at-top* [*simp*]: *eventually* ($\lambda x.$ (*c*::-::{*no-top*, *linorder*}) $< x$) *at-top*
⟨*proof*⟩

lemma *eventually-all-ge-at-top*:
assumes *eventually P* (*at-top* :: ('*a* :: *linorder*) *filter*)
shows *eventually* ($\lambda x.\ \forall y\ge x.\ P\ y$) *at-top*
⟨*proof*⟩

definition *at-bot* :: ('*a*::*order*) *filter*
where *at-bot* = (*INF k. principal* {.. *k*})

lemma *at-bot-sub*: *at-bot* = (*INF k*:{.. *c*::'*a*::*linorder*}. *principal* {.. *k*})
⟨*proof*⟩

lemma *eventually-at-bot-linorder*:
fixes *P* :: '*a*::*linorder* $\Rightarrow$ *bool* **shows** *eventually P at-bot* $\longleftrightarrow$ ($\exists N.\ \forall n\le N.\ P\ n$)
⟨*proof*⟩

lemma *eventually-filtercomap-at-bot-linorder*:
eventually P (*filtercomap f at-bot*) $\longleftrightarrow$ ($\exists N$::'*a*::*linorder*. $\forall x.\ f\ x \le N \longrightarrow P\ x$)
⟨*proof*⟩

lemma *eventually-le-at-bot* [*simp*]:
eventually ($\lambda x.\ x \le$ (*c*::-::*linorder*)) *at-bot*
⟨*proof*⟩

lemma *eventually-at-bot-dense*: *eventually P at-bot* $\longleftrightarrow$ ($\exists N$::'*a*::{*no-bot*, *linorder*}.

$\forall n<N.\ P\ n)$
⟨proof⟩

lemma *eventually-filtercomap-at-bot-dense*:
eventually P (filtercomap f at-bot) $\longleftrightarrow$ $(\exists N::'a::\{no\text{-}bot, linorder\}.\ \forall x.\ f\ x < N \longrightarrow P\ x)$
⟨proof⟩

lemma *eventually-at-bot-not-equal* [*simp*]: *eventually* $(\lambda x::'a::\{no\text{-}bot, linorder\}.\ x \neq c)$ *at-bot*
⟨proof⟩

lemma *eventually-gt-at-bot* [*simp*]:
eventually $(\lambda x.\ x < (c::\text{-}::unbounded\text{-}dense\text{-}linorder))$ *at-bot*
⟨proof⟩

lemma *trivial-limit-at-bot-linorder* [*simp*]: $\neg$ *trivial-limit* (*at-bot* ::('a::*linorder*) *filter*)
⟨proof⟩

lemma *trivial-limit-at-top-linorder* [*simp*]: $\neg$ *trivial-limit* (*at-top* ::('a::*linorder*) *filter*)
⟨proof⟩

89.3 Sequentially

abbreviation *sequentially* :: *nat filter*
where *sequentially* $\equiv$ *at-top*

lemma *eventually-sequentially*:
eventually P sequentially $\longleftrightarrow$ $(\exists N.\ \forall n\geq N.\ P\ n)$
⟨proof⟩

lemma *sequentially-bot* [*simp*, *intro*]: *sequentially* $\neq$ *bot*
⟨proof⟩

lemmas *trivial-limit-sequentially* = *sequentially-bot*

lemma *eventually-False-sequentially* [*simp*]:
$\neg$ *eventually* $(\lambda n.\ False)$ *sequentially*
⟨proof⟩

lemma *le-sequentially*:
$F \leq$ *sequentially* $\longleftrightarrow$ $(\forall N.\ eventually\ (\lambda n.\ N \leq n)\ F)$
⟨proof⟩

lemma *eventually-sequentiallyI* [*intro?*]:
assumes $\bigwedge x.\ c \leq x \Longrightarrow P\ x$
shows *eventually P sequentially*

⟨*proof*⟩

lemma *eventually-sequentially-Suc* [*simp*]: *eventually* (λi. *P* (*Suc i*)) *sequentially* $\longleftrightarrow$ *eventually P sequentially*
⟨*proof*⟩

lemma *eventually-sequentially-seg* [*simp*]: *eventually* (λn. *P* ($n + k$)) *sequentially* $\longleftrightarrow$ *eventually P sequentially*
⟨*proof*⟩

89.4 The cofinite filter

definition *cofinite* = *Abs-filter* (λP. *finite* $\{x. \neg P\ x\}$)

abbreviation *Inf-many* :: ($'a \Rightarrow$ *bool*) $\Rightarrow$ *bool* (**binder** $\exists_\infty$ *10*)
where *Inf-many P* $\equiv$ *frequently P cofinite*

abbreviation *Alm-all* :: ($'a \Rightarrow$ *bool*) $\Rightarrow$ *bool* (**binder** $\forall_\infty$ *10*)
where *Alm-all P* $\equiv$ *eventually P cofinite*

notation (*ASCII*)
Inf-many (**binder** *INFM* *10*) **and**
Alm-all (**binder** *MOST* *10*)

lemma *eventually-cofinite*: *eventually P cofinite* $\longleftrightarrow$ *finite* $\{x. \neg P\ x\}$
⟨*proof*⟩

lemma *frequently-cofinite*: *frequently P cofinite* $\longleftrightarrow$ $\neg$ *finite* $\{x. P\ x\}$
⟨*proof*⟩

lemma *cofinite-bot*[*simp*]: *cofinite* = (*bot*::$'a$ *filter*) $\longleftrightarrow$ *finite* (*UNIV* :: $'a$ *set*)
⟨*proof*⟩

lemma *cofinite-eq-sequentially*: *cofinite* = *sequentially*
⟨*proof*⟩

89.4.1 Product of filters

lemma *filtermap-sequentually-ne-bot*: *filtermap f sequentially* $\neq$ *bot*
⟨*proof*⟩

definition *prod-filter* :: $'a$ *filter* $\Rightarrow$ $'b$ *filter* $\Rightarrow$ ($'a \times 'b$) *filter* (**infixr** $\times_F$ *80*)
where
prod-filter F G =
(*INF* (*P*, *Q*):$\{(P, Q)$. *eventually P F* $\wedge$ *eventually Q G*$\}$. *principal* $\{(x, y). P\ x \wedge Q\ y\}$)

lemma *eventually-prod-filter*: *eventually P* ($F \times_F G$) $\longleftrightarrow$
($\exists$ *Pf Pg*. *eventually Pf F* $\wedge$ *eventually Pg G* $\wedge$ ($\forall x\ y$. *Pf x* $\longrightarrow$ *Pg y* $\longrightarrow$ *P* (*x*, *y*)))

⟨proof⟩

lemma *eventually-prod1*:
assumes $B \neq bot$
shows $(\forall_F\ (x,\ y)\ in\ A \times_F B.\ P\ x) \longleftrightarrow (\forall_F\ x\ in\ A.\ P\ x)$
⟨proof⟩

lemma *eventually-prod2*:
assumes $A \neq bot$
shows $(\forall_F\ (x,\ y)\ in\ A \times_F B.\ P\ y) \longleftrightarrow (\forall_F\ y\ in\ B.\ P\ y)$
⟨proof⟩

lemma *INF-filter-bot-base*:
fixes $F :: 'a \Rightarrow 'b\ filter$
assumes $*$: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ F\ k \leq F\ i \sqcap F\ j$
shows $(INF\ i{:}I.\ F\ i) = bot \longleftrightarrow (\exists i{\in}I.\ F\ i = bot)$
⟨proof⟩

lemma *Collect-empty-eq-bot*: $Collect\ P = \{\} \longleftrightarrow P = \bot$
⟨proof⟩

lemma *prod-filter-eq-bot*: $A \times_F B = bot \longleftrightarrow A = bot \lor B = bot$
⟨proof⟩

lemma *prod-filter-mono*: $F \leq F' \Longrightarrow G \leq G' \Longrightarrow F \times_F G \leq F' \times_F G'$
⟨proof⟩

lemma *prod-filter-mono-iff*:
assumes *nAB*: $A \neq bot\ B \neq bot$
shows $A \times_F B \leq C \times_F D \longleftrightarrow A \leq C \land B \leq D$
⟨proof⟩

lemma *eventually-prod-same*: $eventually\ P\ (F \times_F F) \longleftrightarrow$
$(\exists Q.\ eventually\ Q\ F \land (\forall x\ y.\ Q\ x \longrightarrow Q\ y \longrightarrow P\ (x,\ y)))$
⟨proof⟩

lemma *eventually-prod-sequentially*:
$eventually\ P\ (sequentially \times_F sequentially) \longleftrightarrow (\exists N.\ \forall m \geq N.\ \forall n \geq N.\ P\ (n,\ m))$
⟨proof⟩

lemma *principal-prod-principal*: $principal\ A \times_F principal\ B = principal\ (A \times B)$
⟨proof⟩

lemma *prod-filter-INF*:
assumes $I \neq \{\}\ J \neq \{\}$
shows $(INF\ i{:}I.\ A\ i) \times_F (INF\ j{:}J.\ B\ j) = (INF\ i{:}I.\ INF\ j{:}J.\ A\ i \times_F B\ j)$
⟨proof⟩

lemma *filtermap-Pair*: $filtermap\ (\lambda x.\ (f\ x,\ g\ x))\ F \le filtermap\ f\ F \times_F filtermap\ g\ F$
⟨*proof*⟩

lemma *eventually-prodI*: $eventually\ P\ F \Longrightarrow eventually\ Q\ G \Longrightarrow eventually\ (\lambda x.\ P\ (fst\ x) \wedge Q\ (snd\ x))\ (F \times_F G)$
⟨*proof*⟩

lemma *prod-filter-INF1*: $I \neq \{\} \Longrightarrow (INF\ i{:}I.\ A\ i) \times_F B = (INF\ i{:}I.\ A\ i \times_F B)$
⟨*proof*⟩

lemma *prod-filter-INF2*: $J \neq \{\} \Longrightarrow A \times_F (INF\ i{:}J.\ B\ i) = (INF\ i{:}J.\ A \times_F B\ i)$
⟨*proof*⟩

89.5 Limits

definition *filterlim* :: $('a \Rightarrow 'b) \Rightarrow 'b\ filter \Rightarrow 'a\ filter \Rightarrow bool$ **where**
$filterlim\ f\ F2\ F1 \longleftrightarrow filtermap\ f\ F1 \le F2$

syntax
-LIM :: $pttrns \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow bool$ ((3LIM (-)/ (-)./ (-) :> (-)) [1000, 10, 0, 10] 10)

translations
LIM x F1. f :> F2 == CONST filterlim (λx. f) F2 F1

lemma *filterlim-top* [*simp*]: *filterlim f top F*
⟨*proof*⟩

lemma *filterlim-iff*:
$(LIM\ x\ F1.\ f\ x :> F2) \longleftrightarrow (\forall P.\ eventually\ P\ F2 \longrightarrow eventually\ (\lambda x.\ P\ (f\ x))\ F1)$
⟨*proof*⟩

lemma *filterlim-compose*:
$filterlim\ g\ F3\ F2 \Longrightarrow filterlim\ f\ F2\ F1 \Longrightarrow filterlim\ (\lambda x.\ g\ (f\ x))\ F3\ F1$
⟨*proof*⟩

lemma *filterlim-mono*:
$filterlim\ f\ F2\ F1 \Longrightarrow F2 \le F2' \Longrightarrow F1' \le F1 \Longrightarrow filterlim\ f\ F2'\ F1'$
⟨*proof*⟩

lemma *filterlim-ident*: *LIM x F. x :> F*
⟨*proof*⟩

lemma *filterlim-cong*:
$F1 = F1' \Longrightarrow F2 = F2' \Longrightarrow eventually\ (\lambda x.\ f\ x = g\ x)\ F2 \Longrightarrow filterlim\ f\ F1\ F2 = filterlim\ g\ F1'\ F2'$

⟨*proof*⟩

lemma *filterlim-mono-eventually*:
assumes *filterlim f F G* **and** *ord*: $F \le F'$ $G' \le G$
assumes *eq*: *eventually* $(\lambda x.\ f\ x = f'\ x)\ G'$
shows *filterlim* $f'\ F'\ G'$
⟨*proof*⟩

lemma *filtermap-mono-strong*: *inj f* $\Longrightarrow$ *filtermap f F* $\le$ *filtermap f G* $\longleftrightarrow$ $F \le G$
⟨*proof*⟩

lemma *filtermap-eq-strong*: *inj f* $\Longrightarrow$ *filtermap f F* $=$ *filtermap f G* $\longleftrightarrow$ $F = G$
⟨*proof*⟩

lemma *filtermap-fun-inverse*:
assumes *g*: *filterlim g F G*
assumes *f*: *filterlim f G F*
assumes *ev*: *eventually* $(\lambda x.\ f\ (g\ x) = x)\ G$
shows *filtermap f F* $= G$
⟨*proof*⟩

lemma *filterlim-principal*:
(*LIM x F. f x* :> *principal S*) $\longleftrightarrow$ (*eventually* $(\lambda x.\ f\ x \in S)$ *F*)
⟨*proof*⟩

lemma *filterlim-inf*:
(*LIM x F1. f x* :> *inf F2 F3*) $\longleftrightarrow$ ((*LIM x F1. f x* :> *F2*) $\wedge$ (*LIM x F1. f x* :> *F3*))
⟨*proof*⟩

lemma *filterlim-INF*:
(*LIM x F. f x* :> (*INF b*:*B. G b*)) $\longleftrightarrow$ ($\forall\, b{\in}B$. *LIM x F. f x* :> *G b*)
⟨*proof*⟩

lemma *filterlim-INF-INF*:
($\bigwedge m.\ m \in J \Longrightarrow \exists\, i{\in}I.$ *filtermap f* $(F\ i) \le G\ m$) $\Longrightarrow$ *LIM x* (*INF i*:*I. F i*). *f x* :> (*INF j*:*J. G j*)
⟨*proof*⟩

lemma *filterlim-base*:
$(\bigwedge m\ x.\ m \in J \Longrightarrow i\ m \in I) \Longrightarrow (\bigwedge m\ x.\ m \in J \Longrightarrow x \in F\ (i\ m) \Longrightarrow f\ x \in G\ m) \Longrightarrow$
LIM x (*INF i*:*I. principal* (*F i*)). *f x* :> (*INF j*:*J. principal* (*G j*))
⟨*proof*⟩

lemma *filterlim-base-iff*:
assumes $I \ne \{\}$ **and** *chain*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow F\ i \subseteq F\ j \vee F\ j \subseteq F\ i$
shows (*LIM x* (*INF i*:*I. principal* (*F i*)). *f x* :> *INF j*:*J. principal* (*G j*)) $\longleftrightarrow$

$(\forall j \in J.\ \exists i \in I.\ \forall x \in F\ i.\ f\ x \in G\ j)$
⟨*proof*⟩

lemma *filterlim-filtermap*: $\mathit{filterlim}\ f\ F1\ (\mathit{filtermap}\ g\ F2) = \mathit{filterlim}\ (\lambda x.\ f\ (g\ x))\ F1\ F2$
⟨*proof*⟩

lemma *filterlim-sup*:
$\mathit{filterlim}\ f\ F\ F1 \Longrightarrow \mathit{filterlim}\ f\ F\ F2 \Longrightarrow \mathit{filterlim}\ f\ F\ (\mathit{sup}\ F1\ F2)$
⟨*proof*⟩

lemma *filterlim-sequentially-Suc*:
$(\mathit{LIM}\ x\ \mathit{sequentially}.\ f\ (\mathit{Suc}\ x) :> F) \longleftrightarrow (\mathit{LIM}\ x\ \mathit{sequentially}.\ f\ x :> F)$
⟨*proof*⟩

lemma *filterlim-Suc*: *filterlim Suc sequentially sequentially*
⟨*proof*⟩

lemma *filterlim-If*:
$\mathit{LIM}\ x\ \mathit{inf}\ F\ (\mathit{principal}\ \{x.\ P\ x\}).\ f\ x :> G \Longrightarrow$
$\mathit{LIM}\ x\ \mathit{inf}\ F\ (\mathit{principal}\ \{x.\ \neg\ P\ x\}).\ g\ x :> G \Longrightarrow$
$\mathit{LIM}\ x\ F.\ \mathit{if}\ P\ x\ \mathit{then}\ f\ x\ \mathit{else}\ g\ x :> G$
⟨*proof*⟩

lemma *filterlim-Pair*:
$\mathit{LIM}\ x\ F.\ f\ x :> G \Longrightarrow \mathit{LIM}\ x\ F.\ g\ x :> H \Longrightarrow \mathit{LIM}\ x\ F.\ (f\ x,\ g\ x) :> G \times_F H$
⟨*proof*⟩

89.6 Limits to *at-top* and *at-bot*

lemma *filterlim-at-top*:
fixes $f :: 'a \Rightarrow ('b::\mathit{linorder})$
shows $(\mathit{LIM}\ x\ F.\ f\ x :> \mathit{at\text{-}top}) \longleftrightarrow (\forall Z.\ \mathit{eventually}\ (\lambda x.\ Z \le f\ x)\ F)$
⟨*proof*⟩

lemma *filterlim-at-top-mono*:
$\mathit{LIM}\ x\ F.\ f\ x :> \mathit{at\text{-}top} \Longrightarrow \mathit{eventually}\ (\lambda x.\ f\ x \le (g\ x::'a::\mathit{linorder}))\ F \Longrightarrow$
$\mathit{LIM}\ x\ F.\ g\ x :> \mathit{at\text{-}top}$
⟨*proof*⟩

lemma *filterlim-at-top-dense*:
fixes $f :: 'a \Rightarrow ('b::\mathit{unbounded\text{-}dense\text{-}linorder})$
shows $(\mathit{LIM}\ x\ F.\ f\ x :> \mathit{at\text{-}top}) \longleftrightarrow (\forall Z.\ \mathit{eventually}\ (\lambda x.\ Z < f\ x)\ F)$
⟨*proof*⟩

lemma *filterlim-at-top-ge*:
fixes $f :: 'a \Rightarrow ('b::\mathit{linorder})$ **and** $c :: 'b$
shows $(\mathit{LIM}\ x\ F.\ f\ x :> \mathit{at\text{-}top}) \longleftrightarrow (\forall Z \ge c.\ \mathit{eventually}\ (\lambda x.\ Z \le f\ x)\ F)$
⟨*proof*⟩

lemma *filterlim-at-top-at-top*:
fixes f :: $'a$::*linorder* $\Rightarrow$ $'b$::*linorder*
assumes *mono*: $\bigwedge x\ y.\ Q\ x \Longrightarrow Q\ y \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
assumes *bij*: $\bigwedge x.\ P\ x \Longrightarrow f\ (g\ x) = x\ \bigwedge x.\ P\ x \Longrightarrow Q\ (g\ x)$
assumes Q: *eventually* Q *at-top*
assumes P: *eventually* P *at-top*
shows *filterlim* f *at-top at-top*
⟨*proof*⟩

lemma *filterlim-at-top-gt*:
fixes f :: $'a \Rightarrow$ ($'b$::*unbounded-dense-linorder*) **and** c :: $'b$
shows (*LIM* x F. $f\ x$:> *at-top*) $\longleftrightarrow$ ($\forall Z{>}c$. *eventually* ($\lambda x.\ Z \leq f\ x$) F)
⟨*proof*⟩

lemma *filterlim-at-bot*:
fixes f :: $'a \Rightarrow$ ($'b$::*linorder*)
shows (*LIM* x F. $f\ x$:> *at-bot*) $\longleftrightarrow$ ($\forall Z$. *eventually* ($\lambda x.\ f\ x \leq Z$) F)
⟨*proof*⟩

lemma *filterlim-at-bot-dense*:
fixes f :: $'a \Rightarrow$ ($'b$::{*dense-linorder*, *no-bot*})
shows (*LIM* x F. $f\ x$:> *at-bot*) $\longleftrightarrow$ ($\forall Z$. *eventually* ($\lambda x.\ f\ x < Z$) F)
⟨*proof*⟩

lemma *filterlim-at-bot-le*:
fixes f :: $'a \Rightarrow$ ($'b$::*linorder*) **and** c :: $'b$
shows (*LIM* x F. $f\ x$:> *at-bot*) $\longleftrightarrow$ ($\forall Z{\leq}c$. *eventually* ($\lambda x.\ Z \geq f\ x$) F)
⟨*proof*⟩

lemma *filterlim-at-bot-lt*:
fixes f :: $'a \Rightarrow$ ($'b$::*unbounded-dense-linorder*) **and** c :: $'b$
shows (*LIM* x F. $f\ x$:> *at-bot*) $\longleftrightarrow$ ($\forall Z{<}c$. *eventually* ($\lambda x.\ Z \geq f\ x$) F)
⟨*proof*⟩

lemma *filterlim-filtercomap* [*intro*]: *filterlim* f F (*filtercomap* f F)
⟨*proof*⟩

89.7 Setup $'a$ *filter* for lifting and transfer

context includes *lifting-syntax*
begin

definition *rel-filter* :: ($'a \Rightarrow$ $'b \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *filter* $\Rightarrow$ $'b$ *filter* $\Rightarrow$ *bool*
where *rel-filter* R F G = ((R ===> *op* =) ===> *op* =) (*Rep-filter* F) (*Rep-filter* G)

lemma *rel-filter-eventually*:
rel-filter R F G $\longleftrightarrow$

$((R ===> op =) ===> op =) (\lambda P.\ eventually\ P\ F) (\lambda P.\ eventually\ P\ G)$
⟨*proof*⟩

lemma *filtermap-id* [*simp*, *id-simps*]: *filtermap id* = *id*
⟨*proof*⟩

lemma *filtermap-id'* [*simp*]: *filtermap* $(\lambda x.\ x) = (\lambda F.\ F)$
⟨*proof*⟩

lemma *Quotient-filter* [*quot-map*]:
assumes *Q*: *Quotient R Abs Rep T*
shows *Quotient* (*rel-filter R*) (*filtermap Abs*) (*filtermap Rep*) (*rel-filter T*)
⟨*proof*⟩

lemma *eventually-parametric* [*transfer-rule*]:
((*A* ===> *op* =) ===> *rel-filter A* ===> *op* =) *eventually eventually*
⟨*proof*⟩

lemma *frequently-parametric* [*transfer-rule*]:
((*A* ===> *op* =) ===> *rel-filter A* ===> *op* =) *frequently frequently*
⟨*proof*⟩

lemma *rel-filter-eq* [*relator-eq*]: *rel-filter op* = = *op* =
⟨*proof*⟩

lemma *rel-filter-mono* [*relator-mono*]:
$A \le B \Longrightarrow$ *rel-filter* $A \le$ *rel-filter* B
⟨*proof*⟩

lemma *rel-filter-conversep* [*simp*]: *rel-filter* A^{-1-1} = (*rel-filter* $A)^{-1-1}$
⟨*proof*⟩

lemma *is-filter-parametric-aux*:
assumes *is-filter F*
assumes [*transfer-rule*]: *bi-total A bi-unique A*
and [*transfer-rule*]: ((*A* ===> *op* =) ===> *op* =) *F G*
shows *is-filter G*
⟨*proof*⟩

lemma *is-filter-parametric* [*transfer-rule*]:
⟦ *bi-total A*; *bi-unique A* ⟧
$\Longrightarrow$ (((*A* ===> *op* =) ===> *op* =) ===> *op* =) *is-filter is-filter*
⟨*proof*⟩

lemma *left-total-rel-filter* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique A*
shows *left-total* (*rel-filter A*)
⟨*proof*⟩

lemma *right-total-rel-filter* [*transfer-rule*]:
⟦ *bi-total A*; *bi-unique A* ⟧ ⟹ *right-total* (*rel-filter A*)
⟨*proof*⟩

lemma *bi-total-rel-filter* [*transfer-rule*]:
assumes *bi-total A bi-unique A*
shows *bi-total* (*rel-filter A*)
⟨*proof*⟩

lemma *left-unique-rel-filter* [*transfer-rule*]:
assumes *left-unique A*
shows *left-unique* (*rel-filter A*)
⟨*proof*⟩

lemma *right-unique-rel-filter* [*transfer-rule*]:
right-unique A ⟹ *right-unique* (*rel-filter A*)
⟨*proof*⟩

lemma *bi-unique-rel-filter* [*transfer-rule*]:
bi-unique A ⟹ *bi-unique* (*rel-filter A*)
⟨*proof*⟩

lemma *top-filter-parametric* [*transfer-rule*]:
bi-total A ⟹ (*rel-filter A*) *top top*
⟨*proof*⟩

lemma *bot-filter-parametric* [*transfer-rule*]: (*rel-filter A*) *bot bot*
⟨*proof*⟩

lemma *sup-filter-parametric* [*transfer-rule*]:
(*rel-filter A* ===> *rel-filter A* ===> *rel-filter A*) *sup sup*
⟨*proof*⟩

lemma *Sup-filter-parametric* [*transfer-rule*]:
(*rel-set* (*rel-filter A*) ===> *rel-filter A*) *Sup Sup*
⟨*proof*⟩

lemma *principal-parametric* [*transfer-rule*]:
(*rel-set A* ===> *rel-filter A*) *principal principal*
⟨*proof*⟩

lemma *filtermap-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-filter A* ===> *rel-filter B*) *filtermap filtermap*
⟨*proof*⟩

lemma *filtercomap-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique B bi-total A*
shows ((*A* ===> *B*) ===> *rel-filter B* ===> *rel-filter A*) *filtercomap*

filtercomap
⟨*proof*⟩

context
  fixes $A :: 'a \Rightarrow 'b \Rightarrow bool$
  assumes [*transfer-rule*]: *bi-unique A*
begin

lemma *le-filter-parametric* [*transfer-rule*]:
  (*rel-filter A* ===> *rel-filter A* ===> *op* =) *op* ≤ *op* ≤
⟨*proof*⟩

lemma *less-filter-parametric* [*transfer-rule*]:
  (*rel-filter A* ===> *rel-filter A* ===> *op* =) *op* < *op* <
⟨*proof*⟩

context
  assumes [*transfer-rule*]: *bi-total A*
begin

lemma *Inf-filter-parametric* [*transfer-rule*]:
  (*rel-set* (*rel-filter A*) ===> *rel-filter A*) *Inf Inf*
⟨*proof*⟩

lemma *inf-filter-parametric* [*transfer-rule*]:
  (*rel-filter A* ===> *rel-filter A* ===> *rel-filter A*) *inf inf*
⟨*proof*⟩

end

end

end

Code generation for filters

definition *abstract-filter* :: (*unit* ⇒ *'a filter*) ⇒ *'a filter*
  where [*simp*]: *abstract-filter f* = *f* ()

code-datatype *principal abstract-filter*

hide-const (**open**) *abstract-filter*

declare [[*code drop*: *filterlim prod-filter filtermap eventually*
  inf :: - *filter* ⇒ - *sup* :: - *filter* ⇒ - *less-eq* :: - *filter* ⇒ -
  Abs-filter]]

declare *filterlim-principal* [*code*]
declare *principal-prod-principal* [*code*]

declare *filtermap-principal* [*code*]
declare *filtercomap-principal* [*code*]
declare *eventually-principal* [*code*]
declare *inf-principal* [*code*]
declare *sup-principal* [*code*]
declare *principal-le-iff* [*code*]

lemma *Rep-filter-iff-eventually* [*simp*, *code*]:
  Rep-filter F P $\longleftrightarrow$ *eventually P F*
  ⟨*proof*⟩

lemma *bot-eq-principal-empty* [*code*]:
  bot = *principal* {}
  ⟨*proof*⟩

lemma *top-eq-principal-UNIV* [*code*]:
  top = *principal UNIV*
  ⟨*proof*⟩

instantiation *filter* :: (*equal*) *equal*
begin

definition *equal-filter* :: *'a filter* ⇒ *'a filter* ⇒ *bool*
  where *equal-filter F F'* $\longleftrightarrow$ *F* = *F'*

lemma *equal-filter* [*code*]:
  HOL.equal (*principal A*) (*principal B*) $\longleftrightarrow$ *A* = *B*
  ⟨*proof*⟩

instance
  ⟨*proof*⟩

end

end

90 Conditionally-complete Lattices

theory *Conditionally-Complete-Lattices*
imports *Finite-Set Lattices-Big Set-Interval*
begin

context *linorder*
begin

lemma *Sup-fin-eq-Max*:
  finite X $\Longrightarrow$ *X* ≠ {} $\Longrightarrow$ *Sup-fin X* = *Max X*
  ⟨*proof*⟩

lemma *Inf-fin-eq-Min*:
finite $X \Longrightarrow X \neq \{\} \Longrightarrow$ *Inf-fin* $X =$ *Min* X
⟨*proof*⟩

end

context *preorder*
begin

definition *bdd-above* $A \longleftrightarrow (\exists M. \forall x \in A. x \leq M)$
definition *bdd-below* $A \longleftrightarrow (\exists m. \forall x \in A. m \leq x)$

lemma *bdd-aboveI*[*intro*]: $(\bigwedge x. x \in A \Longrightarrow x \leq M) \Longrightarrow$ *bdd-above* A
⟨*proof*⟩

lemma *bdd-belowI*[*intro*]: $(\bigwedge x. x \in A \Longrightarrow m \leq x) \Longrightarrow$ *bdd-below* A
⟨*proof*⟩

lemma *bdd-aboveI2*: $(\bigwedge x. x \in A \Longrightarrow f\,x \leq M) \Longrightarrow$ *bdd-above* $(f`A)$
⟨*proof*⟩

lemma *bdd-belowI2*: $(\bigwedge x. x \in A \Longrightarrow m \leq f\,x) \Longrightarrow$ *bdd-below* $(f`A)$
⟨*proof*⟩

lemma *bdd-above-empty* [*simp*, *intro*]: *bdd-above* $\{\}$
⟨*proof*⟩

lemma *bdd-below-empty* [*simp*, *intro*]: *bdd-below* $\{\}$
⟨*proof*⟩

lemma *bdd-above-mono*: *bdd-above* $B \Longrightarrow A \subseteq B \Longrightarrow$ *bdd-above* A
⟨*proof*⟩

lemma *bdd-below-mono*: *bdd-below* $B \Longrightarrow A \subseteq B \Longrightarrow$ *bdd-below* A
⟨*proof*⟩

lemma *bdd-above-Int1* [*simp*]: *bdd-above* $A \Longrightarrow$ *bdd-above* $(A \cap B)$
⟨*proof*⟩

lemma *bdd-above-Int2* [*simp*]: *bdd-above* $B \Longrightarrow$ *bdd-above* $(A \cap B)$
⟨*proof*⟩

lemma *bdd-below-Int1* [*simp*]: *bdd-below* $A \Longrightarrow$ *bdd-below* $(A \cap B)$
⟨*proof*⟩

lemma *bdd-below-Int2* [*simp*]: *bdd-below* $B \Longrightarrow$ *bdd-below* $(A \cap B)$
⟨*proof*⟩

lemma *bdd-above-Ioo* [*simp*, *intro*]: *bdd-above* $\{a <..< b\}$

⟨proof⟩

lemma *bdd-above-Ico* [*simp*, *intro*]: *bdd-above* $\{a ..< b\}$
⟨proof⟩

lemma *bdd-above-Iio* [*simp*, *intro*]: *bdd-above* $\{..< b\}$
⟨proof⟩

lemma *bdd-above-Ioc* [*simp*, *intro*]: *bdd-above* $\{a <.. b\}$
⟨proof⟩

lemma *bdd-above-Icc* [*simp*, *intro*]: *bdd-above* $\{a .. b\}$
⟨proof⟩

lemma *bdd-above-Iic* [*simp*, *intro*]: *bdd-above* $\{.. b\}$
⟨proof⟩

lemma *bdd-below-Ioo* [*simp*, *intro*]: *bdd-below* $\{a <..< b\}$
⟨proof⟩

lemma *bdd-below-Ioc* [*simp*, *intro*]: *bdd-below* $\{a <.. b\}$
⟨proof⟩

lemma *bdd-below-Ioi* [*simp*, *intro*]: *bdd-below* $\{a <..\}$
⟨proof⟩

lemma *bdd-below-Ico* [*simp*, *intro*]: *bdd-below* $\{a ..< b\}$
⟨proof⟩

lemma *bdd-below-Icc* [*simp*, *intro*]: *bdd-below* $\{a .. b\}$
⟨proof⟩

lemma *bdd-below-Ici* [*simp*, *intro*]: *bdd-below* $\{a ..\}$
⟨proof⟩

end

lemma (**in** *order-top*) *bdd-above-top*[*simp*, *intro*!]: *bdd-above* A
⟨proof⟩

lemma (**in** *order-bot*) *bdd-above-bot*[*simp*, *intro*!]: *bdd-below* A
⟨proof⟩

lemma *bdd-above-image-mono*: *mono* $f \Longrightarrow$ *bdd-above* $A \Longrightarrow$ *bdd-above* $(f`A)$
⟨proof⟩

lemma *bdd-below-image-mono*: *mono* $f \Longrightarrow$ *bdd-below* $A \Longrightarrow$ *bdd-below* $(f`A)$
⟨proof⟩

lemma *bdd-above-image-antimono*: *antimono* $f \Longrightarrow$ *bdd-below* $A \Longrightarrow$ *bdd-above* $(f`A)$
⟨*proof*⟩

lemma *bdd-below-image-antimono*: *antimono* $f \Longrightarrow$ *bdd-above* $A \Longrightarrow$ *bdd-below* $(f`A)$
⟨*proof*⟩

lemma
fixes X :: *'a::ordered-ab-group-add set*
shows *bdd-above-uminus*[*simp*]: *bdd-above* (*uminus* ` X) $\longleftrightarrow$ *bdd-below* X
and *bdd-below-uminus*[*simp*]: *bdd-below* (*uminus* ` X) $\longleftrightarrow$ *bdd-above* X
⟨*proof*⟩

context *lattice*
begin

lemma *bdd-above-insert* [*simp*]: *bdd-above* (*insert a A*) $=$ *bdd-above* A
⟨*proof*⟩

lemma *bdd-below-insert* [*simp*]: *bdd-below* (*insert a A*) $=$ *bdd-below* A
⟨*proof*⟩

lemma *bdd-finite* [*simp*]:
assumes *finite* A **shows** *bdd-above-finite*: *bdd-above* A **and** *bdd-below-finite*: *bdd-below* A
⟨*proof*⟩

lemma *bdd-above-Un* [*simp*]: *bdd-above* $(A \cup B) =$ (*bdd-above* $A \wedge$ *bdd-above* B)
⟨*proof*⟩

lemma *bdd-below-Un* [*simp*]: *bdd-below* $(A \cup B) =$ (*bdd-below* $A \wedge$ *bdd-below* B)
⟨*proof*⟩

lemma *bdd-above-sup*[*simp*]: *bdd-above* (($\lambda x.$ *sup* $(f\ x)$ $(g\ x)$) ` $A$) $\longleftrightarrow$ *bdd-above* $(f`A) \wedge$ *bdd-above* $(g`A)$
⟨*proof*⟩

lemma *bdd-below-inf*[*simp*]: *bdd-below* (($\lambda x.$ *inf* $(f\ x)$ $(g\ x)$) ` $A$) $\longleftrightarrow$ *bdd-below* $(f`A) \wedge$ *bdd-below* $(g`A)$
⟨*proof*⟩

end

To avoid name classes with the *complete-lattice*-class we prefix *Sup* and *Inf* in theorem names with c.

class *conditionally-complete-lattice* = *lattice* + *Sup* + *Inf* +
assumes *cInf-lower*: $x \in X \Longrightarrow$ *bdd-below* $X \Longrightarrow$ *Inf* $X \leq x$
and *cInf-greatest*: $X \neq \{\} \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow z \leq x) \Longrightarrow z \leq$ *Inf* X

assumes *cSup-upper*: $x \in X \Longrightarrow \textit{bdd-above}\ X \Longrightarrow x \leq \textit{Sup}\ X$
and *cSup-least*: $X \neq \{\} \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow x \leq z) \Longrightarrow \textit{Sup}\ X \leq z$
begin

lemma *cSup-upper2*: $x \in X \Longrightarrow y \leq x \Longrightarrow \textit{bdd-above}\ X \Longrightarrow y \leq \textit{Sup}\ X$
⟨*proof*⟩

lemma *cInf-lower2*: $x \in X \Longrightarrow x \leq y \Longrightarrow \textit{bdd-below}\ X \Longrightarrow \textit{Inf}\ X \leq y$
⟨*proof*⟩

lemma *cSup-mono*: $B \neq \{\} \Longrightarrow \textit{bdd-above}\ A \Longrightarrow (\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ b \leq a) \Longrightarrow \textit{Sup}\ B \leq \textit{Sup}\ A$
⟨*proof*⟩

lemma *cInf-mono*: $B \neq \{\} \Longrightarrow \textit{bdd-below}\ A \Longrightarrow (\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b) \Longrightarrow \textit{Inf}\ A \leq \textit{Inf}\ B$
⟨*proof*⟩

lemma *cSup-subset-mono*: $A \neq \{\} \Longrightarrow \textit{bdd-above}\ B \Longrightarrow A \subseteq B \Longrightarrow \textit{Sup}\ A \leq \textit{Sup}\ B$
⟨*proof*⟩

lemma *cInf-superset-mono*: $A \neq \{\} \Longrightarrow \textit{bdd-below}\ B \Longrightarrow A \subseteq B \Longrightarrow \textit{Inf}\ B \leq \textit{Inf}\ A$
⟨*proof*⟩

lemma *cSup-eq-maximum*: $z \in X \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow x \leq z) \Longrightarrow \textit{Sup}\ X = z$
⟨*proof*⟩

lemma *cInf-eq-minimum*: $z \in X \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow z \leq x) \Longrightarrow \textit{Inf}\ X = z$
⟨*proof*⟩

lemma *cSup-le-iff*: $S \neq \{\} \Longrightarrow \textit{bdd-above}\ S \Longrightarrow \textit{Sup}\ S \leq a \longleftrightarrow (\forall\, x{\in}S.\ x \leq a)$
⟨*proof*⟩

lemma *le-cInf-iff*: $S \neq \{\} \Longrightarrow \textit{bdd-below}\ S \Longrightarrow a \leq \textit{Inf}\ S \longleftrightarrow (\forall\, x{\in}S.\ a \leq x)$
⟨*proof*⟩

lemma *cSup-eq-non-empty*:
assumes *1*: $X \neq \{\}$
assumes *2*: $\bigwedge x.\ x \in X \Longrightarrow x \leq a$
assumes *3*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow x \leq y) \Longrightarrow a \leq y$
shows $\textit{Sup}\ X = a$
⟨*proof*⟩

lemma *cInf-eq-non-empty*:
assumes *1*: $X \neq \{\}$
assumes *2*: $\bigwedge x.\ x \in X \Longrightarrow a \leq x$
assumes *3*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow y \leq x) \Longrightarrow y \leq a$

shows *Inf* $X = a$
⟨*proof*⟩

lemma *cInf-cSup*: $S \neq \{\} \Longrightarrow$ *bdd-below* $S \Longrightarrow$ *Inf* $S =$ *Sup* $\{x.\ \forall s{\in}S.\ x \leq s\}$
⟨*proof*⟩

lemma *cSup-cInf*: $S \neq \{\} \Longrightarrow$ *bdd-above* $S \Longrightarrow$ *Sup* $S =$ *Inf* $\{x.\ \forall s{\in}S.\ s \leq x\}$
⟨*proof*⟩

lemma *cSup-insert*: $X \neq \{\} \Longrightarrow$ *bdd-above* $X \Longrightarrow$ *Sup* (*insert* a X) $=$ *sup* a (*Sup* X)
⟨*proof*⟩

lemma *cInf-insert*: $X \neq \{\} \Longrightarrow$ *bdd-below* $X \Longrightarrow$ *Inf* (*insert* a X) $=$ *inf* a (*Inf* X)
⟨*proof*⟩

lemma *cSup-singleton* [*simp*]: *Sup* $\{x\} = x$
⟨*proof*⟩

lemma *cInf-singleton* [*simp*]: *Inf* $\{x\} = x$
⟨*proof*⟩

lemma *cSup-insert-If*: *bdd-above* $X \Longrightarrow$ *Sup* (*insert* a X) $=$ (*if* $X = \{\}$ *then* a *else sup* a (*Sup* X))
⟨*proof*⟩

lemma *cInf-insert-If*: *bdd-below* $X \Longrightarrow$ *Inf* (*insert* a X) $=$ (*if* $X = \{\}$ *then* a *else inf* a (*Inf* X))
⟨*proof*⟩

lemma *le-cSup-finite*: *finite* $X \Longrightarrow x \in X \Longrightarrow x \leq$ *Sup* X
⟨*proof*⟩

lemma *cInf-le-finite*: *finite* $X \Longrightarrow x \in X \Longrightarrow$ *Inf* $X \leq x$
⟨*proof*⟩

lemma *cSup-eq-Sup-fin*: *finite* $X \Longrightarrow X \neq \{\} \Longrightarrow$ *Sup* $X =$ *Sup-fin* X
⟨*proof*⟩

lemma *cInf-eq-Inf-fin*: *finite* $X \Longrightarrow X \neq \{\} \Longrightarrow$ *Inf* $X =$ *Inf-fin* X
⟨*proof*⟩

lemma *cSup-atMost*[*simp*]: *Sup* $\{..x\} = x$
⟨*proof*⟩

lemma *cSup-greaterThanAtMost*[*simp*]: $y < x \Longrightarrow$ *Sup* $\{y{<}..x\} = x$
⟨*proof*⟩

lemma *cSup-atLeastAtMost*[*simp*]: $y \leq x \Longrightarrow Sup\ \{y..x\} = x$
⟨*proof*⟩

lemma *cInf-atLeast*[*simp*]: $Inf\ \{x..\} = x$
⟨*proof*⟩

lemma *cInf-atLeastLessThan*[*simp*]: $y < x \Longrightarrow Inf\ \{y..<x\} = y$
⟨*proof*⟩

lemma *cInf-atLeastAtMost*[*simp*]: $y \leq x \Longrightarrow Inf\ \{y..x\} = y$
⟨*proof*⟩

lemma *cINF-lower*: *bdd-below* $(f\ `\ A) \Longrightarrow x \in A \Longrightarrow INFIMUM\ A\ f \leq f\ x$
⟨*proof*⟩

lemma *cINF-greatest*: $A \neq \{\} \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow m \leq f\ x) \Longrightarrow m \leq INFIMUM\ A\ f$
⟨*proof*⟩

lemma *cSUP-upper*: $x \in A \Longrightarrow$ *bdd-above* $(f\ `\ A) \Longrightarrow f\ x \leq SUPREMUM\ A\ f$
⟨*proof*⟩

lemma *cSUP-least*: $A \neq \{\} \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \leq M) \Longrightarrow SUPREMUM\ A\ f \leq M$
⟨*proof*⟩

lemma *cINF-lower2*: *bdd-below* $(f\ `\ A) \Longrightarrow x \in A \Longrightarrow f\ x \leq u \Longrightarrow INFIMUM\ A\ f \leq u$
⟨*proof*⟩

lemma *cSUP-upper2*: *bdd-above* $(f\ `\ A) \Longrightarrow x \in A \Longrightarrow u \leq f\ x \Longrightarrow u \leq SUPREMUM\ A\ f$
⟨*proof*⟩

lemma *cSUP-const* [*simp*]: $A \neq \{\} \Longrightarrow (SUP\ x{:}A.\ c) = c$
⟨*proof*⟩

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

lemma *le-cINF-iff*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f\ `\ A) \Longrightarrow u \leq INFIMUM\ A\ f \longleftrightarrow (\forall x{\in}A.\ u \leq f\ x)$
⟨*proof*⟩

lemma *cSUP-le-iff*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f\ `\ A) \Longrightarrow SUPREMUM\ A\ f \leq u \longleftrightarrow (\forall x{\in}A.\ f\ x \leq u)$
⟨*proof*⟩

lemma *less-cINF-D*: *bdd-below* $(f`A) \Longrightarrow y < (INF\ i{:}A.\ f\ i) \Longrightarrow i \in A \Longrightarrow y <$

$f\ i$
⟨*proof*⟩

lemma *cSUP-lessD*: $bdd\text{-}above\ (f`A) \Longrightarrow (SUP\ i{:}A.\ f\ i) < y \Longrightarrow i \in A \Longrightarrow f\ i < y$
⟨*proof*⟩

lemma *cINF-insert*: $A \neq \{\} \Longrightarrow bdd\text{-}below\ (f\ `\ A) \Longrightarrow INFIMUM\ (insert\ a\ A)\ f = inf\ (f\ a)\ (INFIMUM\ A\ f)$
⟨*proof*⟩

lemma *cSUP-insert*: $A \neq \{\} \Longrightarrow bdd\text{-}above\ (f\ `\ A) \Longrightarrow SUPREMUM\ (insert\ a\ A)\ f = sup\ (f\ a)\ (SUPREMUM\ A\ f)$
⟨*proof*⟩

lemma *cINF-mono*: $B \neq \{\} \Longrightarrow bdd\text{-}below\ (f\ `\ A) \Longrightarrow (\bigwedge m.\ m \in B \Longrightarrow \exists\, n{\in}A.\ f\ n \leq g\ m) \Longrightarrow INFIMUM\ A\ f \leq INFIMUM\ B\ g$
⟨*proof*⟩

lemma *cSUP-mono*: $A \neq \{\} \Longrightarrow bdd\text{-}above\ (g\ `\ B) \Longrightarrow (\bigwedge n.\ n \in A \Longrightarrow \exists\, m{\in}B.\ f\ n \leq g\ m) \Longrightarrow SUPREMUM\ A\ f \leq SUPREMUM\ B\ g$
⟨*proof*⟩

lemma *cINF-superset-mono*: $A \neq \{\} \Longrightarrow bdd\text{-}below\ (g\ `\ B) \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow g\ x \leq f\ x) \Longrightarrow INFIMUM\ B\ g \leq INFIMUM\ A\ f$
⟨*proof*⟩

lemma *cSUP-subset-mono*: $A \neq \{\} \Longrightarrow bdd\text{-}above\ (g\ `\ B) \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow f\ x \leq g\ x) \Longrightarrow SUPREMUM\ A\ f \leq SUPREMUM\ B\ g$
⟨*proof*⟩

lemma *less-eq-cInf-inter*: $bdd\text{-}below\ A \Longrightarrow bdd\text{-}below\ B \Longrightarrow A \cap B \neq \{\} \Longrightarrow inf\ (Inf\ A)\ (Inf\ B) \leq Inf\ (A \cap B)$
⟨*proof*⟩

lemma *cSup-inter-less-eq*: $bdd\text{-}above\ A \Longrightarrow bdd\text{-}above\ B \Longrightarrow A \cap B \neq \{\} \Longrightarrow Sup\ (A \cap B) \leq sup\ (Sup\ A)\ (Sup\ B)$
⟨*proof*⟩

lemma *cInf-union-distrib*: $A \neq \{\} \Longrightarrow bdd\text{-}below\ A \Longrightarrow B \neq \{\} \Longrightarrow bdd\text{-}below\ B \Longrightarrow Inf\ (A \cup B) = inf\ (Inf\ A)\ (Inf\ B)$
⟨*proof*⟩

lemma *cINF-union*: $A \neq \{\} \Longrightarrow bdd\text{-}below\ (f`A) \Longrightarrow B \neq \{\} \Longrightarrow bdd\text{-}below\ (f`B) \Longrightarrow INFIMUM\ (A \cup B)\ f = inf\ (INFIMUM\ A\ f)\ (INFIMUM\ B\ f)$
⟨*proof*⟩

lemma *cSup-union-distrib*: $A \neq \{\} \Longrightarrow bdd\text{-}above\ A \Longrightarrow B \neq \{\} \Longrightarrow bdd\text{-}above\ B \Longrightarrow Sup\ (A \cup B) = sup\ (Sup\ A)\ (Sup\ B)$

⟨*proof*⟩

lemma *cSUP-union*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f\text{`}A) \Longrightarrow B \neq \{\} \Longrightarrow$ *bdd-above* $(f\text{`}B)$ $\Longrightarrow$ *SUPREMUM* $(A \cup B)$ f = *sup* (*SUPREMUM* A f) (*SUPREMUM* B f)
⟨*proof*⟩

lemma *cINF-inf-distrib*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f\text{`}A) \Longrightarrow$ *bdd-below* $(g\text{`}A) \Longrightarrow$ *inf* (*INFIMUM* A f) (*INFIMUM* A g) = (*INF* $a{:}A$. *inf* $(f\ a)$ $(g\ a)$)
⟨*proof*⟩

lemma *SUP-sup-distrib*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f\text{`}A) \Longrightarrow$ *bdd-above* $(g\text{`}A) \Longrightarrow$ *sup* (*SUPREMUM* A f) (*SUPREMUM* A g) = (*SUP* $a{:}A$. *sup* $(f\ a)$ $(g\ a)$)
⟨*proof*⟩

lemma *cInf-le-cSup*:
$A \neq \{\} \Longrightarrow$ *bdd-above* $A \Longrightarrow$ *bdd-below* $A \Longrightarrow$ *Inf* $A \leq$ *Sup* A
⟨*proof*⟩

end

instance *complete-lattice* $\subseteq$ *conditionally-complete-lattice*
⟨*proof*⟩

lemma *cSup-eq*:
fixes a :: $'a$ :: {*conditionally-complete-lattice*, *no-bot*}
assumes *upper*: $\bigwedge x.\ x \in X \Longrightarrow x \leq a$
assumes *least*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow x \leq y) \Longrightarrow a \leq y$
shows *Sup* $X = a$
⟨*proof*⟩

lemma *cInf-eq*:
fixes a :: $'a$ :: {*conditionally-complete-lattice*, *no-top*}
assumes *upper*: $\bigwedge x.\ x \in X \Longrightarrow a \leq x$
assumes *least*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow y \leq x) \Longrightarrow y \leq a$
shows *Inf* $X = a$
⟨*proof*⟩

class *conditionally-complete-linorder* = *conditionally-complete-lattice* + *linorder*
begin

lemma *less-cSup-iff*:
$X \neq \{\} \Longrightarrow$ *bdd-above* $X \Longrightarrow y <$ *Sup* $X \longleftrightarrow (\exists x{\in}X.\ y < x)$
⟨*proof*⟩

lemma *cInf-less-iff*: $X \neq \{\} \Longrightarrow$ *bdd-below* $X \Longrightarrow$ *Inf* $X < y \longleftrightarrow (\exists x{\in}X.\ x < y)$
⟨*proof*⟩

lemma *cINF-less-iff*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f\text{`}A) \Longrightarrow$ (*INF* $i{:}A.\ f\ i$) $< a \longleftrightarrow$

$(\exists x \in A.\ f\ x < a)$
⟨proof⟩

lemma *less-cSUP-iff*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f\text{`}A) \Longrightarrow a < (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\exists x \in A.\ a < f\ x)$
⟨proof⟩

lemma *less-cSupE*:
assumes $y < Sup\ X\ X \neq \{\}$ **obtains** x **where** $x \in X\ y < x$
⟨proof⟩

lemma *less-cSupD*:
$X \neq \{\} \Longrightarrow z < Sup\ X \Longrightarrow \exists x \in X.\ z < x$
⟨proof⟩

lemma *cInf-lessD*:
$X \neq \{\} \Longrightarrow Inf\ X < z \Longrightarrow \exists x \in X.\ x < z$
⟨proof⟩

lemma *complete-interval*:
assumes $a < b$ **and** $P\ a$ **and** $\neg\ P\ b$
shows $\exists c.\ a \leq c \wedge c \leq b \wedge (\forall x.\ a \leq x \wedge x < c \longrightarrow P\ x) \wedge$
$(\forall d.\ (\forall x.\ a \leq x \wedge x < d \longrightarrow P\ x) \longrightarrow d \leq c)$
⟨proof⟩

end

instance *complete-linorder* < *conditionally-complete-linorder*
⟨proof⟩

lemma *cSup-eq-Max*: *finite* $(X{::}'a{::}$*conditionally-complete-linorder set*$) \Longrightarrow X \neq \{\} \Longrightarrow Sup\ X = Max\ X$
⟨proof⟩

lemma *cInf-eq-Min*: *finite* $(X{::}'a{::}$*conditionally-complete-linorder set*$) \Longrightarrow X \neq \{\} \Longrightarrow Inf\ X = Min\ X$
⟨proof⟩

lemma *cSup-lessThan*[*simp*]: $Sup\ \{..{<}x{::}'a{::}\{$*conditionally-complete-linorder*, *no-bot*, *dense-linorder*$\}\} = x$
⟨proof⟩

lemma *cSup-greaterThanLessThan*[*simp*]: $y < x \Longrightarrow Sup\ \{y{<}..{<}x{::}'a{::}\{$*conditionally-complete-linorder*, *dense-linorder*$\}\} = x$
⟨proof⟩

lemma *cSup-atLeastLessThan*[*simp*]: $y < x \Longrightarrow Sup\ \{y..{<}x{::}'a{::}\{$*conditionally-complete-linorder*, *dense-linorder*$\}\} = x$
⟨proof⟩

lemma *cInf-greaterThan*[*simp*]: *Inf* $\{x::'a::\{$*conditionally-complete-linorder*, *no-top*, *dense-linorder*$\}$ <..$\}$ = x
⟨*proof*⟩

lemma *cInf-greaterThanAtMost*[*simp*]: $y < x \Longrightarrow$ *Inf* $\{y$<..$x::'a::\{$*conditionally-complete-linorder*, *dense-linorder*$\}\} = y$
⟨*proof*⟩

lemma *cInf-greaterThanLessThan*[*simp*]: $y < x \Longrightarrow$ *Inf* $\{y$<..<$x::'a::\{$*conditionally-complete-linorder*, *dense-linorder*$\}\} = y$
⟨*proof*⟩

class *linear-continuum* = *conditionally-complete-linorder* + *dense-linorder* +
assumes *UNIV-not-singleton*: $\exists\, a\ b::'a.\ a \neq b$
begin

lemma *ex-gt-or-lt*: $\exists\, b.\ a < b \vee b < a$
⟨*proof*⟩

end

instantiation *nat* :: *conditionally-complete-linorder*
begin

definition *Sup* (X::*nat set*) = *Max X*
definition *Inf* (X::*nat set*) = (*LEAST* $n.\ n \in X$)

lemma *bdd-above-nat*: *bdd-above X* $\longleftrightarrow$ *finite* (X::*nat set*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *int* :: *conditionally-complete-linorder*
begin

definition *Sup* (X::*int set*) = (*THE* $x.\ x \in X \wedge (\forall\, y{\in}X.\ y \leq x)$)
definition *Inf* (X::*int set*) = $-$ (*Sup* (*uminus* ‘ X))

instance
⟨*proof*⟩
end

lemma *interval-cases*:
fixes S :: $'a$ :: *conditionally-complete-linorder set*
assumes *ivl*: $\bigwedge a\ b\ x.\ a \in S \Longrightarrow b \in S \Longrightarrow a \leq x \Longrightarrow x \leq b \Longrightarrow x \in S$

shows $\exists\, a\ b.\ S = \{\} \vee$
$S = UNIV \vee$
$S = \{..<b\} \vee$
$S = \{..b\} \vee$
$S = \{a<..\} \vee$
$S = \{a..\} \vee$
$S = \{a<..<b\} \vee$
$S = \{a<..b\} \vee$
$S = \{a..<b\} \vee$
$S = \{a..b\}$
$\langle proof \rangle$

lemma *cSUP-eq-cINF-D*:
fixes f :: - $\Rightarrow$ $'b$::*conditionally-complete-lattice*
assumes *eq*: $(SUP\ x{:}A.\ f\ x) = (INF\ x{:}A.\ f\ x)$
and *bdd*: *bdd-above* $(f\ `\ A)$ *bdd-below* $(f\ `\ A)$
and *a*: $a \in A$
shows $f\ a = (INF\ x{:}A.\ f\ x)$
$\langle proof \rangle$

lemma *cSUP-UNION*:
fixes f :: - $\Rightarrow$ $'b$::*conditionally-complete-lattice*
assumes *ne*: $A \neq \{\}$ $\bigwedge x.\ x \in A \Longrightarrow B(x) \neq \{\}$
and *bdd-UN*: *bdd-above* $(\bigcup x{\in}A.\ f\ `\ B\ x)$
shows $(SUP\ z : \bigcup x{\in}A.\ B\ x.\ f\ z) = (SUP\ x{:}A.\ SUP\ z{:}B\ x.\ f\ z)$
$\langle proof \rangle$

lemma *cINF-UNION*:
fixes f :: - $\Rightarrow$ $'b$::*conditionally-complete-lattice*
assumes *ne*: $A \neq \{\}$ $\bigwedge x.\ x \in A \Longrightarrow B(x) \neq \{\}$
and *bdd-UN*: *bdd-below* $(\bigcup x{\in}A.\ f\ `\ B\ x)$
shows $(INF\ z : \bigcup x{\in}A.\ B\ x.\ f\ z) = (INF\ x{:}A.\ INF\ z{:}B\ x.\ f\ z)$
$\langle proof \rangle$

lemma *cSup-abs-le*:
fixes S :: $('a$::$\{$*linordered-idom*,*conditionally-complete-linorder*$\})$ *set*
shows $S \neq \{\} \Longrightarrow (\bigwedge x.\ x{\in}S \Longrightarrow |x| \leq a) \Longrightarrow |Sup\ S| \leq a$
$\langle proof \rangle$

end

91 Factorial Function, Rising Factorials

theory *Factorial*
imports *Groups-List*
begin

91.1 Factorial Function

context *semiring-char-0*
begin

definition *fact* :: *nat* ⇒ *'a*
  where *fact-prod*: $fact\ n = of\text{-}nat\ (\prod\{1..n\})$

lemma *fact-prod-Suc*: $fact\ n = of\text{-}nat\ (prod\ Suc\ \{0..<n\})$
  ⟨*proof*⟩

lemma *fact-prod-rev*: $fact\ n = of\text{-}nat\ (\prod i = 0..<n.\ n - i)$
  ⟨*proof*⟩

lemma *fact-0* [*simp*]: $fact\ 0 = 1$
  ⟨*proof*⟩

lemma *fact-1* [*simp*]: $fact\ 1 = 1$
  ⟨*proof*⟩

lemma *fact-Suc-0* [*simp*]: $fact\ (Suc\ 0) = 1$
  ⟨*proof*⟩

lemma *fact-Suc* [*simp*]: $fact\ (Suc\ n) = of\text{-}nat\ (Suc\ n) * fact\ n$
  ⟨*proof*⟩

lemma *fact-2* [*simp*]: $fact\ 2 = 2$
  ⟨*proof*⟩

lemma *fact-split*: $k \leq n \Longrightarrow fact\ n = of\text{-}nat\ (prod\ Suc\ \{n - k..<n\}) * fact\ (n - k)$
  ⟨*proof*⟩

end

lemma *of-nat-fact* [*simp*]: $of\text{-}nat\ (fact\ n) = fact\ n$
  ⟨*proof*⟩

lemma *of-int-fact* [*simp*]: $of\text{-}int\ (fact\ n) = fact\ n$
  ⟨*proof*⟩

lemma *fact-reduce*: $n > 0 \Longrightarrow fact\ n = of\text{-}nat\ n * fact\ (n - 1)$
  ⟨*proof*⟩

lemma *fact-nonzero* [*simp*]: $fact\ n \neq (0::'a::\{semiring\text{-}char\text{-}0,semiring\text{-}no\text{-}zero\text{-}divisors\})$
  ⟨*proof*⟩

lemma *fact-mono-nat*: $m \leq n \Longrightarrow fact\ m \leq (fact\ n :: nat)$
  ⟨*proof*⟩

lemma *fact-in-Nats*: *fact* $n \in \mathbb{N}$
⟨*proof*⟩

lemma *fact-in-Ints*: *fact* $n \in \mathbb{Z}$
⟨*proof*⟩

context
assumes *SORT-CONSTRAINT*(*'a::linordered-semidom*)
begin

lemma *fact-mono*: $m \leq n \Longrightarrow$ *fact* $m \leq$ (*fact* n :: *'a*)
⟨*proof*⟩

lemma *fact-ge-1* [*simp*]: *fact* $n \geq$ (*1* :: *'a*)
⟨*proof*⟩

lemma *fact-gt-zero* [*simp*]: *fact* $n >$ (*0* :: *'a*)
⟨*proof*⟩

lemma *fact-ge-zero* [*simp*]: *fact* $n \geq$ (*0* :: *'a*)
⟨*proof*⟩

lemma *fact-not-neg* [*simp*]: $\neg$ *fact* $n <$ (*0* :: *'a*)
⟨*proof*⟩

lemma *fact-le-power*: *fact* $n \leq$ (*of-nat* ($n \hat{} n$) :: *'a*)
⟨*proof*⟩

end

lemma *fact-less-mono-nat*: *0* $< m \Longrightarrow m < n \Longrightarrow$ *fact* $m <$ (*fact* n :: *nat*)
⟨*proof*⟩

lemma *fact-less-mono*: *0* $< m \Longrightarrow m < n \Longrightarrow$ *fact* $m <$ (*fact* n :: *'a::linordered-semidom*)
⟨*proof*⟩

lemma *fact-ge-Suc-0-nat* [*simp*]: *fact* $n \geq$ *Suc 0*
⟨*proof*⟩

lemma *dvd-fact*: *1* $\leq m \Longrightarrow m \leq n \Longrightarrow m$ *dvd fact* n
⟨*proof*⟩

lemma *fact-ge-self*: *fact* $n \geq n$
⟨*proof*⟩

lemma *fact-dvd*: $n \leq m \Longrightarrow$ *fact* n *dvd* (*fact* m :: *'a::{semiring-div,linordered-semidom}*)
⟨*proof*⟩

lemma *fact-mod*: $m \leq n \Longrightarrow$ *fact* n *mod* (*fact* m :: *'a::{semiring-div,linordered-semidom}*)

$= 0$
⟨*proof*⟩

lemma *fact-div-fact*:
assumes $m \geq n$
shows $\mathit{fact}\ m\ \mathit{div}\ \mathit{fact}\ n = \prod \{n + 1..m\}$
⟨*proof*⟩

lemma *fact-num-eq-if*: $\mathit{fact}\ m = (\mathit{if}\ m = 0\ \mathit{then}\ 1\ \mathit{else}\ \mathit{of\text{-}nat}\ m * \mathit{fact}\ (m - 1))$
⟨*proof*⟩

lemma *fact-div-fact-le-pow*:
assumes $r \leq n$
shows $\mathit{fact}\ n\ \mathit{div}\ \mathit{fact}\ (n - r) \leq n\ \hat{}\ r$
⟨*proof*⟩

lemma *fact-numeral*: $\mathit{fact}\ (\mathit{numeral}\ k) = \mathit{numeral}\ k * \mathit{fact}\ (\mathit{pred\text{-}numeral}\ k)$
— Evaluation for specific numerals
⟨*proof*⟩

91.2 Pochhammer's symbol: generalized rising factorial

See http://en.wikipedia.org/wiki/Pochhammer_symbol.

context *comm-semiring-1*
begin

definition *pochhammer* :: $'a \Rightarrow nat \Rightarrow 'a$
where *pochhammer-prod*: $\mathit{pochhammer}\ a\ n = \mathit{prod}\ (\lambda i.\ a + \mathit{of\text{-}nat}\ i)\ \{0..<n\}$

lemma *pochhammer-prod-rev*: $\mathit{pochhammer}\ a\ n = \mathit{prod}\ (\lambda i.\ a + \mathit{of\text{-}nat}\ (n - i))$ $\{1..n\}$
⟨*proof*⟩

lemma *pochhammer-Suc-prod*: $\mathit{pochhammer}\ a\ (\mathit{Suc}\ n) = \mathit{prod}\ (\lambda i.\ a + \mathit{of\text{-}nat}\ i)$ $\{0..n\}$
⟨*proof*⟩

lemma *pochhammer-Suc-prod-rev*: $\mathit{pochhammer}\ a\ (\mathit{Suc}\ n) = \mathit{prod}\ (\lambda i.\ a + \mathit{of\text{-}nat}$ $(n - i))\ \{0..n\}$
⟨*proof*⟩

lemma *pochhammer-0* [*simp*]: $\mathit{pochhammer}\ a\ 0 = 1$
⟨*proof*⟩

lemma *pochhammer-1* [*simp*]: $\mathit{pochhammer}\ a\ 1 = a$
⟨*proof*⟩

lemma *pochhammer-Suc0* [*simp*]: $\mathit{pochhammer}\ a\ (\mathit{Suc}\ 0) = a$
⟨*proof*⟩

lemma *pochhammer-Suc*: *pochhammer a* (*Suc n*) = *pochhammer a n* ∗ (*a* + *of-nat n*)
⟨*proof*⟩

end

lemma *pochhammer-nonneg*:
fixes *x* :: *'a* :: *linordered-semidom*
shows *x* > *0* ⟹ *pochhammer x n* ≥ *0*
⟨*proof*⟩

lemma *pochhammer-pos*:
fixes *x* :: *'a* :: *linordered-semidom*
shows *x* > *0* ⟹ *pochhammer x n* > *0*
⟨*proof*⟩

lemma *pochhammer-of-nat*: *pochhammer* (*of-nat x*) *n* = *of-nat* (*pochhammer x n*)
⟨*proof*⟩

lemma *pochhammer-of-int*: *pochhammer* (*of-int x*) *n* = *of-int* (*pochhammer x n*)
⟨*proof*⟩

lemma *pochhammer-rec*: *pochhammer a* (*Suc n*) = *a* ∗ *pochhammer* (*a* + *1*) *n*
⟨*proof*⟩

lemma *pochhammer-rec'*: *pochhammer z* (*Suc n*) = (*z* + *of-nat n*) ∗ *pochhammer z n*
⟨*proof*⟩

lemma *pochhammer-fact*: *fact n* = *pochhammer 1 n*
⟨*proof*⟩

lemma *pochhammer-of-nat-eq-0-lemma*: *k* > *n* ⟹ *pochhammer* (− (*of-nat n* :: *'a*:: *idom*)) *k* = *0*
⟨*proof*⟩

lemma *pochhammer-of-nat-eq-0-lemma'*:
assumes *kn*: *k* ≤ *n*
shows *pochhammer* (− (*of-nat n* :: *'a*::{*idom*,*ring-char-0*})) *k* ≠ *0*
⟨*proof*⟩

lemma *pochhammer-of-nat-eq-0-iff*:
pochhammer (− (*of-nat n* :: *'a*::{*idom*,*ring-char-0*})) *k* = *0* ⟷ *k* > *n*
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *pochhammer-0-left*:
pochhammer 0 n = (*if n* = *0 then 1 else 0*)

⟨*proof*⟩

lemma *pochhammer-eq-0-iff*: *pochhammer a n* = (*0*::′*a*::*field-char-0*) ⟷ (∃ *k* < *n*. *a* = − *of-nat k*)
⟨*proof*⟩

lemma *pochhammer-eq-0-mono*:
pochhammer a n = (*0*::′*a*::*field-char-0*) ⟹ *m* ≥ *n* ⟹ *pochhammer a m* = *0*
⟨*proof*⟩

lemma *pochhammer-neq-0-mono*:
pochhammer a m ≠ (*0*::′*a*::*field-char-0*) ⟹ *m* ≥ *n* ⟹ *pochhammer a n* ≠ *0*
⟨*proof*⟩

lemma *pochhammer-minus*:
pochhammer (− *b*) *k* = ((− *1*) ^ *k* :: ′*a*::*comm-ring-1*) ∗ *pochhammer* (*b* − *of-nat k* + *1*) *k*
⟨*proof*⟩

lemma *pochhammer-minus′*:
pochhammer (*b* − *of-nat k* + *1*) *k* = ((− *1*) ^ *k* :: ′*a*::*comm-ring-1*) ∗ *pochhammer* (− *b*) *k*
⟨*proof*⟩

lemma *pochhammer-same*: *pochhammer* (− *of-nat n*) *n* =
((− *1*) ^ *n* :: ′*a*::{*semiring-char-0*,*comm-ring-1*,*semiring-no-zero-divisors*}) ∗ *fact n*
⟨*proof*⟩

lemma *pochhammer-product′*: *pochhammer z* (*n* + *m*) = *pochhammer z n* ∗ *pochhammer* (*z* + *of-nat n*) *m*
⟨*proof*⟩

lemma *pochhammer-product*:
m ≤ *n* ⟹ *pochhammer z n* = *pochhammer z m* ∗ *pochhammer* (*z* + *of-nat m*) (*n* − *m*)
⟨*proof*⟩

lemma *pochhammer-times-pochhammer-half*:
fixes *z* :: ′*a*::*field-char-0*
shows *pochhammer z* (*Suc n*) ∗ *pochhammer* (*z* + *1*/*2*) (*Suc n*) = (∏ *k*=*0*..*2*∗*n*+*1*. *z* + *of-nat k* / *2*)
⟨*proof*⟩

lemma *pochhammer-double*:
fixes *z* :: ′*a*::*field-char-0*
shows *pochhammer* (*2* ∗ *z*) (*2* ∗ *n*) = *of-nat* (*2*^(*2*∗*n*)) ∗ *pochhammer z n* ∗ *pochhammer* (*z*+*1*/*2*) *n*
⟨*proof*⟩

lemma *fact-double*:
  fact $(2 * n) = (2 \hat{} (2 * n) * pochhammer\ (1 / 2)\ n * fact\ n :: 'a::field\text{-}char\text{-}0)$
  ⟨*proof*⟩

lemma *pochhammer-absorb-comp*: $(r - of\text{-}nat\ k) * pochhammer\ (- r)\ k = r * pochhammer\ (-r + 1)\ k$
  (**is** *?lhs* = *?rhs*)
  for $r :: 'a::comm\text{-}ring\text{-}1$
⟨*proof*⟩

91.3 Misc

lemma *fact-code* [*code*]:
  $fact\ n = (of\text{-}nat\ (fold\text{-}atLeastAtMost\text{-}nat\ (op\ *)\ 2\ n\ 1) :: 'a::semiring\text{-}char\text{-}0)$
⟨*proof*⟩

lemma *pochhammer-code* [*code*]:
  pochhammer a n =
    (*if n* = *0 then 1*
     else fold-atLeastAtMost-nat $(\lambda n\ acc.\ (a + of\text{-}nat\ n) * acc)\ 0\ (n - 1)\ 1)$
  ⟨*proof*⟩

end

92 Binomial Coefficients and Binomial Theorem

theory *Binomial*
  imports *Presburger Factorial*
begin

92.1 Binomial coefficients

This development is based on the work of Andy Gordon and Florian Kammueller.

Combinatorial definition

definition *binomial* :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *choose 65*)
  where $n\ choose\ k = card\ \{K \in Pow\ \{0..<n\}.\ card\ K = k\}$

theorem *n-subsets*:
  assumes *finite A*
  shows $card\ \{B.\ B \subseteq A \wedge card\ B = k\} = card\ A\ choose\ k$
⟨*proof*⟩

Recursive characterization

lemma *binomial-n-0* [*simp*, *code*]: *n choose 0* = *1*
⟨*proof*⟩

lemma *binomial-0-Suc* [*simp*, *code*]: 0 *choose Suc* $k = 0$
⟨*proof*⟩

lemma *binomial-Suc-Suc* [*simp*, *code*]: *Suc* n *choose Suc* k = (n *choose* k) + (n *choose Suc* k)
⟨*proof*⟩

lemma *binomial-eq-0*: $n < k \Longrightarrow n$ *choose* $k = 0$
⟨*proof*⟩

lemma *zero-less-binomial*: $k \leq n \Longrightarrow n$ *choose* $k > 0$
⟨*proof*⟩

lemma *binomial-eq-0-iff* [*simp*]: n *choose* $k = 0 \longleftrightarrow n < k$
⟨*proof*⟩

lemma *zero-less-binomial-iff* [*simp*]: n *choose* $k > 0 \longleftrightarrow k \leq n$
⟨*proof*⟩

lemma *binomial-n-n* [*simp*]: n *choose* $n = 1$
⟨*proof*⟩

lemma *binomial-Suc-n* [*simp*]: *Suc* n *choose* n = *Suc* n
⟨*proof*⟩

lemma *binomial-1* [*simp*]: n *choose Suc* $0 = n$
⟨*proof*⟩

lemma *choose-reduce-nat*:
$0 < n \Longrightarrow 0 < k \Longrightarrow$
n *choose* k = (($n - 1$) *choose* ($k - 1$)) + (($n - 1$) *choose* k)
⟨*proof*⟩

lemma *Suc-times-binomial-eq*: *Suc* n * (n *choose* k) = (*Suc* n *choose Suc* k) * *Suc* k
⟨*proof*⟩

lemma *binomial-le-pow2*: n *choose* $k \leq 2\hat{\ }n$
⟨*proof*⟩

The absorption property.

lemma *Suc-times-binomial*: *Suc* k * (*Suc* n *choose Suc* k) = *Suc* n * (n *choose* k)
⟨*proof*⟩

This is the well-known version of absorption, but it's harder to use because of the need to reason about division.

lemma *binomial-Suc-Suc-eq-times*: (*Suc* n *choose Suc* k) = (*Suc* n * (n *choose* k)) *div Suc* k

⟨*proof*⟩

Another version of absorption, with -1 instead of *Suc*.

lemma *times-binomial-minus1-eq*: $0 < k \Longrightarrow k * (n \text{ choose } k) = n * ((n - 1) \text{ choose } (k - 1))$
⟨*proof*⟩

92.2 The binomial theorem (courtesy of Tobias Nipkow):

Avigad's version, generalized to any commutative ring

theorem *binomial-ring*: $(a + b :: 'a::\{comm\text{-}ring\text{-}1,power\})\hat{\ }n = (\sum k=0..n. (of\text{-}nat (n \text{ choose } k)) * a\hat{\ }k * b\hat{\ }(n-k))$
⟨*proof*⟩

Original version for the naturals.

corollary *binomial*: $(a + b :: nat)\hat{\ }n = (\sum k=0..n. (of\text{-}nat (n \text{ choose } k)) * a\hat{\ }k * b\hat{\ }(n - k))$
⟨*proof*⟩

lemma *binomial-fact-lemma*: $k \le n \Longrightarrow fact\ k * fact\ (n - k) * (n \text{ choose } k) = fact\ n$
⟨*proof*⟩

lemma *binomial-fact'*:
assumes $k \le n$
shows $n \text{ choose } k = fact\ n\ div\ (fact\ k * fact\ (n - k))$
⟨*proof*⟩

lemma *binomial-fact*:
assumes *kn*: $k \le n$
shows $(of\text{-}nat\ (n \text{ choose } k) :: 'a::field\text{-}char\text{-}0) = fact\ n\ /\ (fact\ k * fact\ (n - k))$
⟨*proof*⟩

lemma *fact-binomial*:
assumes $k \le n$
shows $fact\ k * of\text{-}nat\ (n \text{ choose } k) = (fact\ n\ /\ fact\ (n - k) :: 'a::field\text{-}char\text{-}0)$
⟨*proof*⟩

lemma *choose-two*: $n \text{ choose } 2 = n * (n - 1)\ div\ 2$
⟨*proof*⟩

lemma *choose-row-sum*: $(\sum k=0..n.\ n \text{ choose } k) = 2\hat{\ }n$
⟨*proof*⟩

lemma *sum-choose-lower*: $(\sum k=0..n.\ (r+k) \text{ choose } k) = Suc\ (r+n) \text{ choose } n$
⟨*proof*⟩

lemma *sum-choose-upper*: $(\sum k=0..n.\ k \text{ choose } m) = Suc\ n \text{ choose } Suc\ m$

⟨*proof*⟩

lemma *choose-alternating-sum*:
$n > 0 \Longrightarrow (\sum i \le n.\ (-1)\,\hat{}\,i * \textit{of-nat}\ (n\ \textit{choose}\ i)) = (0 :: 'a::\textit{comm-ring-1})$
⟨*proof*⟩

lemma *choose-even-sum*:
assumes $n > 0$
shows $2 * (\sum i \le n.\ \textit{if even } i \textit{ then of-nat } (n\ \textit{choose}\ i)\ \textit{else}\ 0) = (2\ \hat{}\ n :: 'a::\textit{comm-ring-1})$
⟨*proof*⟩

lemma *choose-odd-sum*:
assumes $n > 0$
shows $2 * (\sum i \le n.\ \textit{if odd } i \textit{ then of-nat } (n\ \textit{choose}\ i)\ \textit{else}\ 0) = (2\ \hat{}\ n :: 'a::\textit{comm-ring-1})$
⟨*proof*⟩

lemma *choose-row-sum'*: $(\sum k \le n.\ (n\ \textit{choose}\ k)) = 2\ \hat{}\ n$
⟨*proof*⟩

NW diagonal sum property

lemma *sum-choose-diagonal*:
assumes $m \le n$
shows $(\sum k{=}0..m.\ (n - k)\ \textit{choose}\ (m - k)) = \textit{Suc}\ n\ \textit{choose}\ m$
⟨*proof*⟩

92.3 Generalized binomial coefficients

definition *gbinomial* :: $'a::\{\textit{semidom-divide},\textit{semiring-char-0}\} \Rightarrow \textit{nat} \Rightarrow 'a$ (**infixl** *gchoose 65*)
where *gbinomial-prod-rev*: $a\ \textit{gchoose}\ n = \textit{prod}\ (\lambda i.\ a - \textit{of-nat}\ i)\ \{0..<n\}\ \textit{div}\ \textit{fact}\ n$

lemma *gbinomial-0* [*simp*]:
$a\ \textit{gchoose}\ 0 = 1$
$0\ \textit{gchoose}\ (\textit{Suc}\ n) = 0$
⟨*proof*⟩

lemma *gbinomial-Suc*: $a\ \textit{gchoose}\ (\textit{Suc}\ k) = \textit{prod}\ (\lambda i.\ a - \textit{of-nat}\ i)\ \{0..k\}\ \textit{div}\ \textit{fact}\ (\textit{Suc}\ k)$
⟨*proof*⟩

lemma *gbinomial-mult-fact*: $\textit{fact}\ n * (a\ \textit{gchoose}\ n) = (\prod i = 0..<n.\ a - \textit{of-nat}\ i)$
for $a :: 'a::\textit{field-char-0}$
⟨*proof*⟩

lemma *gbinomial-mult-fact'*: $(a\ \textit{gchoose}\ n) * \textit{fact}\ n = (\prod i = 0..<n.\ a - \textit{of-nat}\ i)$

for a :: $'a$::*field-char-0*
⟨*proof*⟩

lemma *gbinomial-pochhammer*: a *gchoose* n = $(- 1)$ ^ n ∗ *pochhammer* $(- a)$ n / *fact* n
for a :: $'a$::*field-char-0*
⟨*proof*⟩

lemma *gbinomial-pochhammer'*: s *gchoose* n = *pochhammer* (s − *of-nat* n + *1*) n / *fact* n
for s :: $'a$::*field-char-0*
⟨*proof*⟩

lemma *gbinomial-binomial*: n *gchoose* k = n *choose* k
⟨*proof*⟩

lemma *of-nat-gbinomial*: *of-nat* (n *gchoose* k) = (*of-nat* n *gchoose* k :: $'a$::*field-char-0*)
⟨*proof*⟩

lemma *binomial-gbinomial*: *of-nat* (n *choose* k) = (*of-nat* n *gchoose* k :: $'a$::*field-char-0*)
⟨*proof*⟩

⟨*ML*⟩

lemma *gbinomial-1*[*simp*]: a *gchoose* *1* = a
⟨*proof*⟩

lemma *gbinomial-Suc0*[*simp*]: a *gchoose* (*Suc* *0*) = a
⟨*proof*⟩

lemma *gbinomial-mult-1*:
fixes a :: $'a$::*field-char-0*
shows a ∗ (a *gchoose* n) = *of-nat* n ∗ (a *gchoose* n) + *of-nat* (*Suc* n) ∗ (a *gchoose* (*Suc* n))
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *gbinomial-mult-1'*:
(a *gchoose* n) ∗ a = *of-nat* n ∗ (a *gchoose* n) + *of-nat* (*Suc* n) ∗ (a *gchoose* (*Suc* n))
for a :: $'a$::*field-char-0*
⟨*proof*⟩

lemma *gbinomial-Suc-Suc*: (a + *1*) *gchoose* (*Suc* k) = a *gchoose* k + (a *gchoose* (*Suc* k))
for a :: $'a$::*field-char-0*
⟨*proof*⟩

lemma *gbinomial-reduce-nat*: *0* < k ⟹ a *gchoose* k = (a − *1*) *gchoose* (k − *1*)

+ ((*a* − *1*) *gchoose k*)
for *a* :: *'a::field-char-0*
⟨*proof*⟩

lemma *gchoose-row-sum-weighted*:
($\sum$ *k* = *0..m*. (*r gchoose k*) ∗ (*r*/*2* − *of-nat k*)) = *of-nat*(*Suc m*) / *2* ∗ (*r gchoose* (*Suc m*))
for *r* :: *'a::field-char-0*
⟨*proof*⟩

lemma *binomial-symmetric*:
assumes *kn*: *k* ≤ *n*
shows *n choose k* = *n choose* (*n* − *k*)
⟨*proof*⟩

lemma *choose-rising-sum*:
($\sum$ *j*≤*m*. ((*n* + *j*) *choose n*)) = ((*n* + *m* + *1*) *choose* (*n* + *1*))
($\sum$ *j*≤*m*. ((*n* + *j*) *choose n*)) = ((*n* + *m* + *1*) *choose m*)
⟨*proof*⟩

lemma *choose-linear-sum*: ($\sum$ *i*≤*n*. *i* ∗ (*n choose i*)) = *n* ∗ *2* ^ (*n* − *1*)
⟨*proof*⟩

lemma *choose-alternating-linear-sum*:
assumes *n* ≠ *1*
shows ($\sum$ *i*≤*n*. (−*1*)^*i* ∗ *of-nat i* ∗ *of-nat* (*n choose i*) :: *'a::comm-ring-1*) = *0*
⟨*proof*⟩

lemma *vandermonde*: ($\sum$ *k*≤*r*. (*m choose k*) ∗ (*n choose* (*r* − *k*))) = (*m* + *n*) *choose r*
⟨*proof*⟩

lemma *choose-square-sum*: ($\sum$ *k*≤*n*. (*n choose k*)^*2*) = ((*2*∗*n*) *choose n*)
⟨*proof*⟩

lemma *pochhammer-binomial-sum*:
fixes *a b* :: *'a::comm-ring-1*
shows *pochhammer* (*a* + *b*) *n* = ($\sum$ *k*≤*n*. *of-nat* (*n choose k*) ∗ *pochhammer a k* ∗ *pochhammer b* (*n* − *k*))
⟨*proof*⟩

Contributed by Manuel Eberl, generalised by LCP. Alternative definition of the binomial coefficient as $\prod$ *i*<*k*. (*n* − *i*) / (*k* − *i*).

lemma *gbinomial-altdef-of-nat*: *x gchoose k* = ($\prod$ *i* = *0..*<*k*. (*x* − *of-nat i*) / *of-nat* (*k* − *i*) :: *'a*)
for *k* :: *nat* **and** *x* :: *'a::field-char-0*
⟨*proof*⟩

lemma *gbinomial-ge-n-over-k-pow-k*:

fixes k :: *nat*
and x :: *'a::linordered-field*
assumes *of-nat* $k \leq x$
shows $(x$ / *of-nat* k :: *'a*) ^ $k \leq x$ *gchoose* k
⟨*proof*⟩

lemma *gbinomial-negated-upper*: (a *gchoose* b) = (−*1*) ^ b ∗ ((*of-nat* b − a − *1*) *gchoose* b)
⟨*proof*⟩

lemma *gbinomial-minus*: ((−a) *gchoose* b) = (−*1*) ^ b ∗ ((a + *of-nat* b − *1*) *gchoose* b)
⟨*proof*⟩

lemma *Suc-times-gbinomial*: *of-nat* (*Suc* b) ∗ ((a + *1*) *gchoose* (*Suc* b)) = (a + *1*) ∗ (a *gchoose* b)
⟨*proof*⟩

lemma *gbinomial-factors*: ((a + *1*) *gchoose* (*Suc* b)) = (a + *1*) / *of-nat* (*Suc* b) ∗ (a *gchoose* b)
⟨*proof*⟩

lemma *gbinomial-rec*: ((r + *1*) *gchoose* (*Suc* k)) = (r *gchoose* k) ∗ ((r + *1*) / *of-nat* (*Suc* k))
⟨*proof*⟩

lemma *gbinomial-of-nat-symmetric*: $k \leq n \Longrightarrow$ (*of-nat* n) *gchoose* k = (*of-nat* n) *gchoose* (n − k)
⟨*proof*⟩

The absorption identity (equation 5.5 [**?**, p. 157]):

$$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}, \quad \text{integer } k \neq 0.$$

lemma *gbinomial-absorption'*: $k > 0 \Longrightarrow r$ *gchoose* k = (r / *of-nat* k) ∗ (r − *1* *gchoose* (k − *1*))
⟨*proof*⟩

The absorption identity is written in the following form to avoid division by k (the lower index) and therefore remove the $k \neq 0$ restriction[**?**, p. 157]:

$$k\binom{r}{k} = r\binom{r-1}{k-1}, \quad \text{integer } k.$$

lemma *gbinomial-absorption*: *of-nat* (*Suc* k) ∗ (r *gchoose Suc* k) = r ∗ ((r − *1*) *gchoose* k)
⟨*proof*⟩

The absorption identity for natural number binomial coefficients:

lemma *binomial-absorption*: $Suc\ k * (n\ choose\ Suc\ k) = n * ((n - 1)\ choose\ k)$
⟨*proof*⟩

The absorption companion identity for natural number coefficients, following the proof by GKP [**?**, p. 157]:

lemma *binomial-absorb-comp*: $(n - k) * (n\ choose\ k) = n * ((n - 1)\ choose\ k)$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

The generalised absorption companion identity:

lemma *gbinomial-absorb-comp*: $(r - of\text{-}nat\ k) * (r\ gchoose\ k) = r * ((r - 1)\ gchoose\ k)$
⟨*proof*⟩

lemma *gbinomial-addition-formula*:
$r\ gchoose\ (Suc\ k) = ((r - 1)\ gchoose\ (Suc\ k)) + ((r - 1)\ gchoose\ k)$
⟨*proof*⟩

lemma *binomial-addition-formula*:
$0 < n \Longrightarrow n\ choose\ (Suc\ k) = ((n - 1)\ choose\ (Suc\ k)) + ((n - 1)\ choose\ k)$
⟨*proof*⟩

Equation 5.9 of the reference material [**?**, p. 159] is a useful summation formula, operating on both indices:

$$\sum_{k \le n} \binom{r+k}{k} = \binom{r+n+1}{n}, \quad \text{integer } n.$$

lemma *gbinomial-parallel-sum*: $(\sum k \le n.\ (r + of\text{-}nat\ k)\ gchoose\ k) = (r + of\text{-}nat\ n + 1)\ gchoose\ n$
⟨*proof*⟩

92.3.1 Summation on the upper index

Another summation formula is equation 5.10 of the reference material [**?**, p. 160], aptly named *summation on the upper index*:

$$\sum_{0 \le k \le n} \binom{k}{m} = \binom{n+1}{m+1}, \quad \text{integers } m, n \ge 0.$$

lemma *gbinomial-sum-up-index*:
$(\sum k = 0..n.\ (of\text{-}nat\ k\ gchoose\ m) :: 'a::field\text{-}char\text{-}0) = (of\text{-}nat\ n + 1)\ gchoose\ (m + 1)$
⟨*proof*⟩

lemma *gbinomial-index-swap*:

$((-1)\ \hat{}\ m) * ((-\ (of\text{-}nat\ n) - 1)\ gchoose\ m) = ((-1)\ \hat{}\ n) * ((-\ (of\text{-}nat\ m) - 1)\ gchoose\ n)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *gbinomial-sum-lower-neg*: $(\sum k \le m.\ (r\ gchoose\ k) * (-\ 1)\ \hat{}\ k) = (-\ 1)\ \hat{}\ m * (r - 1\ gchoose\ m)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *gbinomial-partial-row-sum*:
$(\sum k \le m.\ (r\ gchoose\ k) * ((r\ /\ 2) - of\text{-}nat\ k)) = ((of\text{-}nat\ m + 1)/2) * (r\ gchoose\ (m + 1))$
⟨*proof*⟩

lemma *sum-bounds-lt-plus1*: $(\sum k<mm.\ f\ (Suc\ k)) = (\sum k=1..mm.\ f\ k)$
⟨*proof*⟩

lemma *gbinomial-partial-sum-poly*:
$(\sum k \le m.\ (of\text{-}nat\ m + r\ gchoose\ k) * x\hat{}k * y\hat{}(m-k)) =$
$(\sum k \le m.\ (-r\ gchoose\ k) * (-x)\hat{}k * (x + y)\hat{}(m-k))$
(**is** $?lhs\ m = ?rhs\ m$)
⟨*proof*⟩

lemma *gbinomial-partial-sum-poly-xpos*:
$(\sum k \le m.\ (of\text{-}nat\ m + r\ gchoose\ k) * x\hat{}k * y\hat{}(m-k)) =$
$(\sum k \le m.\ (of\text{-}nat\ k + r - 1\ gchoose\ k) * x\hat{}k * (x + y)\hat{}(m-k))$
⟨*proof*⟩

lemma *binomial-r-part-sum*: $(\sum k \le m.\ (2 * m + 1\ choose\ k)) = 2\ \hat{}\ (2 * m)$
⟨*proof*⟩

lemma *gbinomial-r-part-sum*: $(\sum k \le m.\ (2 * (of\text{-}nat\ m) + 1\ gchoose\ k)) = 2\ \hat{}\ (2 * m)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *gbinomial-sum-nat-pow2*:
$(\sum k \le m.\ (of\text{-}nat\ (m + k)\ gchoose\ k :: 'a::field\text{-}char\text{-}0)\ /\ 2\ \hat{}\ k) = 2\ \hat{}\ m$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *gbinomial-trinomial-revision*:
assumes $k \le m$
shows $(r\ gchoose\ m) * (of\text{-}nat\ m\ gchoose\ k) = (r\ gchoose\ k) * (r - of\text{-}nat\ k\ gchoose\ (m - k))$
⟨*proof*⟩

Versions of the theorems above for the natural-number version of "choose"

lemma *binomial-altdef-of-nat*:
$k \le n \Longrightarrow$ *of-nat* (n *choose* k) = ($\prod i = 0..<k.$ *of-nat* ($n - i$) / *of-nat* ($k - i$) :: $'a$)
for $n\ k$ :: *nat* **and** x :: $'a$::*field-char-0*
⟨*proof*⟩

lemma *binomial-ge-n-over-k-pow-k*: $k \le n \Longrightarrow$ (*of-nat* n / *of-nat* k :: $'a$) ^ $k \le$ *of-nat* (n *choose* k)
for $k\ n$ :: *nat* **and** x :: $'a$::*linordered-field*
⟨*proof*⟩

lemma *binomial-le-pow*:
assumes $r \le n$
shows n *choose* $r \le n$ ^ r
⟨*proof*⟩

lemma *binomial-altdef-nat*: $k \le n \Longrightarrow n$ *choose* k = *fact* n *div* (*fact* k ∗ *fact* ($n - k$))
for $k\ n$ :: *nat*
⟨*proof*⟩

lemma *choose-dvd*:
$k \le n \Longrightarrow$ *fact* k ∗ *fact* ($n - k$) *dvd* (*fact* n :: $'a$::{*semiring-div*,*linordered-semidom*})
⟨*proof*⟩

lemma *fact-fact-dvd-fact*:
fact k ∗ *fact* n *dvd* (*fact* ($k + n$) :: $'a$::{*semiring-div*,*linordered-semidom*})
⟨*proof*⟩

lemma *choose-mult-lemma*:
(($m + r + k$) *choose* ($m + k$)) ∗ (($m + k$) *choose* k) = (($m + r + k$) *choose* k) ∗ (($m + r$) *choose* m)
(**is** *?lhs* = -)
⟨*proof*⟩

The "Subset of a Subset" identity.

lemma *choose-mult*:
$k \le m \Longrightarrow m \le n \Longrightarrow$ (n *choose* m) ∗ (m *choose* k) = (n *choose* k) ∗ (($n - k$) *choose* ($m - k$))
⟨*proof*⟩

92.4 More on Binomial Coefficients

lemma *choose-one*: n *choose* $1 = n$ **for** n :: *nat*
⟨*proof*⟩

lemma *card-UNION*:
assumes *finite* A
and $\forall k \in A.$ *finite* k

shows *card* $(\bigcup A) = nat\ (\sum I \mid I \subseteq A \wedge I \neq \{\}.\ (-\ 1)\ \hat{}\ (card\ I + 1) * int\ (card\ (\bigcap I)))$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

The number of nat lists of length m summing to N is $N + m - 1$ *choose* N:

lemma *card-length-sum-list-rec*:
assumes $m \geq 1$
shows *card* $\{l::nat\ list.\ length\ l = m \wedge sum\text{-}list\ l = N\} =$
card $\{l.\ length\ l = (m - 1) \wedge sum\text{-}list\ l = N\}$ +
card $\{l.\ length\ l = m \wedge sum\text{-}list\ l + 1 = N\}$
(**is** *card ?C = card ?A + card ?B*)
⟨*proof*⟩

lemma *card-length-sum-list*: *card* $\{l::nat\ list.\ size\ l = m \wedge sum\text{-}list\ l = N\} = (N + m - 1)$ *choose* N
— by Holden Lee, tidied by Tobias Nipkow
⟨*proof*⟩

lemma *card-disjoint-shuffle*:
assumes *set xs* $\cap$ *set ys* = {}
shows *card (shuffle xs ys)* = (*length xs* + *length ys*) *choose length xs*
⟨*proof*⟩

lemma *Suc-times-binomial-add*: *Suc a* $*$ (*Suc* $(a + b)$ *choose Suc a*) = *Suc b* $*$ (*Suc* $(a + b)$ *choose a*)
— by Lukas Bulwahn
⟨*proof*⟩

92.5 Misc

lemma *gbinomial-code* [*code*]:
a gchoose n =
(*if* $n = 0$ *then* 1
else fold-atLeastAtMost-nat ($\lambda n\ acc.\ (a - of\text{-}nat\ n) * acc$) 0 $(n - 1)$ 1 / *fact* n)
⟨*proof*⟩

declare [[*code drop*: *binomial*]]

lemma *binomial-code* [*code*]:
(*n choose k*) =
(*if* $k > n$ *then* 0
else if $2 * k > n$ *then* (*n choose* $(n - k)$)
else (*fold-atLeastAtMost-nat* (*op* $*$) $(n-k+1)$ *n* 1 *div fact k*))
⟨*proof*⟩

end

93 Main HOL

Classical Higher-order Logic – only "Main", excluding real and complex numbers etc.

theory *Main*
imports *Predicate-Compile Quickcheck-Narrowing Extraction Nunchaku BNF-Greatest-Fixpoint Filter Conditionally-Complete-Lattices Binomial GCD*
begin

Classical Higher-order Logic – only "Main", excluding real and complex numbers etc.

no-notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*) **and**
ordLeq2 (**infix** <=*o 50*) **and**
ordLeq3 (**infix** ≤*o 50*) **and**
ordLess2 (**infix** <*o 50*) **and**
ordIso2 (**infix** =*o 50*) **and**
card-of (|-|) **and**
csum (**infixr** +*c 65*) **and**
cprod (**infixr** ∗*c 80*) **and**
cexp (**infixr** ^*c 90*) **and**
convol (⟨(-,/ -)⟩)

hide-const (**open**)
czero cinfinite cfinite csum cone ctwo Csum cprod cexp image2 image2p vimage2p Gr Grp collect
fsts snds setl setr convol pick-middlep fstOp sndOp csquare relImage relInvImage Succ Shift
shift proj id-bnf

hide-fact (**open**) *id-bnf-def type-definition-id-bnf-UNIV*

no-syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)

end

94 Archimedean Fields, Floor and Ceiling Functions

theory *Archimedean-Field*
imports *Main*
begin

lemma *cInf-abs-ge*:
  fixes S :: $'a$::{*linordered-idom*,*conditionally-complete-linorder*} *set*
  assumes $S \neq \{\}$
    and *bdd*: $\bigwedge x.\ x{\in}S \Longrightarrow |x| \leq a$
  shows $|Inf\ S| \leq a$
⟨*proof*⟩

lemma *cSup-asclose*:
  fixes S :: $'a$::{*linordered-idom*,*conditionally-complete-linorder*} *set*
  assumes S: $S \neq \{\}$
    and b: $\forall x{\in}S.\ |x - l| \leq e$
  shows $|Sup\ S - l| \leq e$
⟨*proof*⟩

lemma *cInf-asclose*:
  fixes S :: $'a$::{*linordered-idom*,*conditionally-complete-linorder*} *set*
  assumes S: $S \neq \{\}$
    and b: $\forall x{\in}S.\ |x - l| \leq e$
  shows $|Inf\ S - l| \leq e$
⟨*proof*⟩

94.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

class *archimedean-field* = *linordered-field* +
  assumes *ex-le-of-int*: $\exists z.\ x \leq$ *of-int* z

lemma *ex-less-of-int*: $\exists z.\ x <$ *of-int* z
  for x :: $'a$::*archimedean-field*
⟨*proof*⟩

lemma *ex-of-int-less*: $\exists z.$ *of-int* $z < x$
  for x :: $'a$::*archimedean-field*
⟨*proof*⟩

lemma *reals-Archimedean2*: $\exists n.\ x <$ *of-nat* n
  for x :: $'a$::*archimedean-field*
⟨*proof*⟩

lemma *real-arch-simple*: $\exists n.\ x \leq$ *of-nat* n
  for x :: $'a$::*archimedean-field*

⟨*proof*⟩

Archimedean fields have no infinitesimal elements.

lemma *reals-Archimedean*:
fixes $x :: 'a::archimedean\text{-}field$
assumes $0 < x$
shows $\exists n.\ inverse\ (of\text{-}nat\ (Suc\ n)) < x$
⟨*proof*⟩

lemma *ex-inverse-of-nat-less*:
fixes $x :: 'a::archimedean\text{-}field$
assumes $0 < x$
shows $\exists n>0.\ inverse\ (of\text{-}nat\ n) < x$
⟨*proof*⟩

lemma *ex-less-of-nat-mult*:
fixes $x :: 'a::archimedean\text{-}field$
assumes $0 < x$
shows $\exists n.\ y < of\text{-}nat\ n * x$
⟨*proof*⟩

94.2 Existence and uniqueness of floor function

lemma *exists-least-lemma*:
assumes $\neg\ P\ 0$ **and** $\exists n.\ P\ n$
shows $\exists n.\ \neg\ P\ n \wedge P\ (Suc\ n)$
⟨*proof*⟩

lemma *floor-exists*:
fixes $x :: 'a::archimedean\text{-}field$
shows $\exists z.\ of\text{-}int\ z \le x \wedge x < of\text{-}int\ (z + 1)$
⟨*proof*⟩

lemma *floor-exists1*: $\exists!z.\ of\text{-}int\ z \le x \wedge x < of\text{-}int\ (z + 1)$
for $x :: 'a::archimedean\text{-}field$
⟨*proof*⟩

94.3 Floor function

class *floor-ceiling* = *archimedean-field* +
fixes $floor :: 'a \Rightarrow int\ \ (\lfloor\text{-}\rfloor)$
assumes *floor-correct*: $of\text{-}int\ \lfloor x\rfloor \le x \wedge x < of\text{-}int\ (\lfloor x\rfloor + 1)$

lemma *floor-unique*: $of\text{-}int\ z \le x \Longrightarrow x < of\text{-}int\ z + 1 \Longrightarrow \lfloor x\rfloor = z$
⟨*proof*⟩

lemma *floor-eq-iff*: $\lfloor x\rfloor = a \longleftrightarrow of\text{-}int\ a \le x \wedge x < of\text{-}int\ a + 1$
⟨*proof*⟩

lemma *of-int-floor-le* [*simp*]: *of-int* $\lfloor x \rfloor \leq x$
⟨*proof*⟩

lemma *le-floor-iff*: $z \leq \lfloor x \rfloor \longleftrightarrow$ *of-int* $z \leq x$
⟨*proof*⟩

lemma *floor-less-iff*: $\lfloor x \rfloor < z \longleftrightarrow x <$ *of-int* z
⟨*proof*⟩

lemma *less-floor-iff*: $z < \lfloor x \rfloor \longleftrightarrow$ *of-int* $z + 1 \leq x$
⟨*proof*⟩

lemma *floor-le-iff*: $\lfloor x \rfloor \leq z \longleftrightarrow x <$ *of-int* $z + 1$
⟨*proof*⟩

lemma *floor-split*[*arith-split*]: $P\ \lfloor t \rfloor \longleftrightarrow (\forall\, i.$ *of-int* $i \leq t \wedge t <$ *of-int* $i + 1 \longrightarrow P\ i)$
⟨*proof*⟩

lemma *floor-mono*:
assumes $x \leq y$
shows $\lfloor x \rfloor \leq \lfloor y \rfloor$
⟨*proof*⟩

lemma *floor-less-cancel*: $\lfloor x \rfloor < \lfloor y \rfloor \Longrightarrow x < y$
⟨*proof*⟩

lemma *floor-of-int* [*simp*]: $\lfloor$*of-int* $z\rfloor = z$
⟨*proof*⟩

lemma *floor-of-nat* [*simp*]: $\lfloor$*of-nat* $n\rfloor =$ *int* n
⟨*proof*⟩

lemma *le-floor-add*: $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$
⟨*proof*⟩

Floor with numerals.

lemma *floor-zero* [*simp*]: $\lfloor 0 \rfloor = 0$
⟨*proof*⟩

lemma *floor-one* [*simp*]: $\lfloor 1 \rfloor = 1$
⟨*proof*⟩

lemma *floor-numeral* [*simp*]: $\lfloor$*numeral* $v\rfloor =$ *numeral* v
⟨*proof*⟩

lemma *floor-neg-numeral* [*simp*]: $\lfloor -$ *numeral* $v\rfloor = -$ *numeral* v
⟨*proof*⟩

lemma *zero-le-floor* [*simp*]: $0 \leq \lfloor x \rfloor \longleftrightarrow 0 \leq x$
⟨*proof*⟩

lemma *one-le-floor* [*simp*]: $1 \leq \lfloor x \rfloor \longleftrightarrow 1 \leq x$
⟨*proof*⟩

lemma *numeral-le-floor* [*simp*]: $\mathit{numeral}\ v \leq \lfloor x \rfloor \longleftrightarrow \mathit{numeral}\ v \leq x$
⟨*proof*⟩

lemma *neg-numeral-le-floor* [*simp*]: $-\ \mathit{numeral}\ v \leq \lfloor x \rfloor \longleftrightarrow -\ \mathit{numeral}\ v \leq x$
⟨*proof*⟩

lemma *zero-less-floor* [*simp*]: $0 < \lfloor x \rfloor \longleftrightarrow 1 \leq x$
⟨*proof*⟩

lemma *one-less-floor* [*simp*]: $1 < \lfloor x \rfloor \longleftrightarrow 2 \leq x$
⟨*proof*⟩

lemma *numeral-less-floor* [*simp*]: $\mathit{numeral}\ v < \lfloor x \rfloor \longleftrightarrow \mathit{numeral}\ v + 1 \leq x$
⟨*proof*⟩

lemma *neg-numeral-less-floor* [*simp*]: $-\ \mathit{numeral}\ v < \lfloor x \rfloor \longleftrightarrow -\ \mathit{numeral}\ v + 1 \leq x$
⟨*proof*⟩

lemma *floor-le-zero* [*simp*]: $\lfloor x \rfloor \leq 0 \longleftrightarrow x < 1$
⟨*proof*⟩

lemma *floor-le-one* [*simp*]: $\lfloor x \rfloor \leq 1 \longleftrightarrow x < 2$
⟨*proof*⟩

lemma *floor-le-numeral* [*simp*]: $\lfloor x \rfloor \leq \mathit{numeral}\ v \longleftrightarrow x < \mathit{numeral}\ v + 1$
⟨*proof*⟩

lemma *floor-le-neg-numeral* [*simp*]: $\lfloor x \rfloor \leq -\ \mathit{numeral}\ v \longleftrightarrow x < -\ \mathit{numeral}\ v + 1$
⟨*proof*⟩

lemma *floor-less-zero* [*simp*]: $\lfloor x \rfloor < 0 \longleftrightarrow x < 0$
⟨*proof*⟩

lemma *floor-less-one* [*simp*]: $\lfloor x \rfloor < 1 \longleftrightarrow x < 1$
⟨*proof*⟩

lemma *floor-less-numeral* [*simp*]: $\lfloor x \rfloor < \mathit{numeral}\ v \longleftrightarrow x < \mathit{numeral}\ v$
⟨*proof*⟩

lemma *floor-less-neg-numeral* [*simp*]: $\lfloor x \rfloor < -\ \mathit{numeral}\ v \longleftrightarrow x < -\ \mathit{numeral}\ v$
⟨*proof*⟩

lemma *le-mult-floor-Ints*:
  assumes $0 \leq a$ $a \in Ints$
  shows $\textit{of-int}\ (\lfloor a \rfloor * \lfloor b \rfloor) \leq (\textit{of-int}\lfloor a * b \rfloor :: 'a :: \textit{linordered-idom})$
  ⟨*proof*⟩

Addition and subtraction of integers.

lemma *floor-add-int*: $\lfloor x \rfloor + z = \lfloor x + \textit{of-int}\ z \rfloor$
  ⟨*proof*⟩

lemma *int-add-floor*: $z + \lfloor x \rfloor = \lfloor \textit{of-int}\ z + x \rfloor$
  ⟨*proof*⟩

lemma *one-add-floor*: $\lfloor x \rfloor + 1 = \lfloor x + 1 \rfloor$
  ⟨*proof*⟩

lemma *floor-diff-of-int* [*simp*]: $\lfloor x - \textit{of-int}\ z \rfloor = \lfloor x \rfloor - z$
  ⟨*proof*⟩

lemma *floor-uminus-of-int* [*simp*]: $\lfloor - (\textit{of-int}\ z) \rfloor = - z$
  ⟨*proof*⟩

lemma *floor-diff-numeral* [*simp*]: $\lfloor x - \textit{numeral}\ v \rfloor = \lfloor x \rfloor - \textit{numeral}\ v$
  ⟨*proof*⟩

lemma *floor-diff-one* [*simp*]: $\lfloor x - 1 \rfloor = \lfloor x \rfloor - 1$
  ⟨*proof*⟩

lemma *le-mult-floor*:
  assumes $0 \leq a$ **and** $0 \leq b$
  shows $\lfloor a \rfloor * \lfloor b \rfloor \leq \lfloor a * b \rfloor$
⟨*proof*⟩

lemma *floor-divide-of-int-eq*: $\lfloor \textit{of-int}\ k\ /\ \textit{of-int}\ l \rfloor = k\ \textit{div}\ l$
  for $k\ l :: \textit{int}$
⟨*proof*⟩

lemma *floor-divide-of-nat-eq*: $\lfloor \textit{of-nat}\ m\ /\ \textit{of-nat}\ n \rfloor = \textit{of-nat}\ (m\ \textit{div}\ n)$
  for $m\ n :: \textit{nat}$
⟨*proof*⟩

94.4 Ceiling function

definition *ceiling* :: $'a::\textit{floor-ceiling} \Rightarrow \textit{int}$ $(\lceil\text{-}\rceil)$
  where $\lceil x \rceil = - \lfloor - x \rfloor$

lemma *ceiling-correct*: $\textit{of-int}\ \lceil x \rceil - 1 < x \wedge x \leq \textit{of-int}\ \lceil x \rceil$
  ⟨*proof*⟩

lemma *ceiling-unique*: $\mathit{of\text{-}int}\ z - 1 < x \Longrightarrow x \le \mathit{of\text{-}int}\ z \Longrightarrow \lceil x \rceil = z$
⟨proof⟩

lemma *ceiling-eq-iff*: $\lceil x \rceil = a \longleftrightarrow \mathit{of\text{-}int}\ a - 1 < x \wedge x \le \mathit{of\text{-}int}\ a$
⟨proof⟩

lemma *le-of-int-ceiling* [*simp*]: $x \le \mathit{of\text{-}int}\ \lceil x \rceil$
⟨proof⟩

lemma *ceiling-le-iff*: $\lceil x \rceil \le z \longleftrightarrow x \le \mathit{of\text{-}int}\ z$
⟨proof⟩

lemma *less-ceiling-iff*: $z < \lceil x \rceil \longleftrightarrow \mathit{of\text{-}int}\ z < x$
⟨proof⟩

lemma *ceiling-less-iff*: $\lceil x \rceil < z \longleftrightarrow x \le \mathit{of\text{-}int}\ z - 1$
⟨proof⟩

lemma *le-ceiling-iff*: $z \le \lceil x \rceil \longleftrightarrow \mathit{of\text{-}int}\ z - 1 < x$
⟨proof⟩

lemma *ceiling-mono*: $x \ge y \Longrightarrow \lceil x \rceil \ge \lceil y \rceil$
⟨proof⟩

lemma *ceiling-less-cancel*: $\lceil x \rceil < \lceil y \rceil \Longrightarrow x < y$
⟨proof⟩

lemma *ceiling-of-int* [*simp*]: $\lceil \mathit{of\text{-}int}\ z \rceil = z$
⟨proof⟩

lemma *ceiling-of-nat* [*simp*]: $\lceil \mathit{of\text{-}nat}\ n \rceil = \mathit{int}\ n$
⟨proof⟩

lemma *ceiling-add-le*: $\lceil x + y \rceil \le \lceil x \rceil + \lceil y \rceil$
⟨proof⟩

lemma *mult-ceiling-le*:
assumes $0 \le a$ **and** $0 \le b$
shows $\lceil a * b \rceil \le \lceil a \rceil * \lceil b \rceil$
⟨proof⟩

lemma *mult-ceiling-le-Ints*:
assumes $0 \le a\ a \in \mathit{Ints}$
shows $(\mathit{of\text{-}int}\ \lceil a * b \rceil :: 'a :: \mathit{linordered\text{-}idom}) \le \mathit{of\text{-}int}(\lceil a \rceil * \lceil b \rceil)$
⟨proof⟩

lemma *finite-int-segment*:
fixes $a :: 'a{::}\mathit{floor\text{-}ceiling}$
shows $\mathit{finite}\ \{x \in \mathbb{Z}.\ a \le x \wedge x \le b\}$

⟨*proof*⟩

corollary *finite-abs-int-segment*:
fixes a :: $'a$::*floor-ceiling*
shows *finite* $\{k \in \mathbb{Z}.\ |k| \leq a\}$
⟨*proof*⟩

Ceiling with numerals.

lemma *ceiling-zero* [*simp*]: $\lceil 0 \rceil = 0$
⟨*proof*⟩

lemma *ceiling-one* [*simp*]: $\lceil 1 \rceil = 1$
⟨*proof*⟩

lemma *ceiling-numeral* [*simp*]: $\lceil \mathit{numeral}\ v \rceil = \mathit{numeral}\ v$
⟨*proof*⟩

lemma *ceiling-neg-numeral* [*simp*]: $\lceil -\ \mathit{numeral}\ v \rceil = -\ \mathit{numeral}\ v$
⟨*proof*⟩

lemma *ceiling-le-zero* [*simp*]: $\lceil x \rceil \leq 0 \longleftrightarrow x \leq 0$
⟨*proof*⟩

lemma *ceiling-le-one* [*simp*]: $\lceil x \rceil \leq 1 \longleftrightarrow x \leq 1$
⟨*proof*⟩

lemma *ceiling-le-numeral* [*simp*]: $\lceil x \rceil \leq \mathit{numeral}\ v \longleftrightarrow x \leq \mathit{numeral}\ v$
⟨*proof*⟩

lemma *ceiling-le-neg-numeral* [*simp*]: $\lceil x \rceil \leq -\ \mathit{numeral}\ v \longleftrightarrow x \leq -\ \mathit{numeral}\ v$
⟨*proof*⟩

lemma *ceiling-less-zero* [*simp*]: $\lceil x \rceil < 0 \longleftrightarrow x \leq -1$
⟨*proof*⟩

lemma *ceiling-less-one* [*simp*]: $\lceil x \rceil < 1 \longleftrightarrow x \leq 0$
⟨*proof*⟩

lemma *ceiling-less-numeral* [*simp*]: $\lceil x \rceil < \mathit{numeral}\ v \longleftrightarrow x \leq \mathit{numeral}\ v - 1$
⟨*proof*⟩

lemma *ceiling-less-neg-numeral* [*simp*]: $\lceil x \rceil < -\ \mathit{numeral}\ v \longleftrightarrow x \leq -\ \mathit{numeral}\ v - 1$
⟨*proof*⟩

lemma *zero-le-ceiling* [*simp*]: $0 \leq \lceil x \rceil \longleftrightarrow -1 < x$
⟨*proof*⟩

lemma *one-le-ceiling* [*simp*]: $1 \leq \lceil x \rceil \longleftrightarrow 0 < x$

⟨*proof*⟩

lemma *numeral-le-ceiling* [*simp*]: $numeral\ v \leq \lceil x \rceil \longleftrightarrow numeral\ v - 1 < x$
⟨*proof*⟩

lemma *neg-numeral-le-ceiling* [*simp*]: $- numeral\ v \leq \lceil x \rceil \longleftrightarrow - numeral\ v - 1 < x$
⟨*proof*⟩

lemma *zero-less-ceiling* [*simp*]: $0 < \lceil x \rceil \longleftrightarrow 0 < x$
⟨*proof*⟩

lemma *one-less-ceiling* [*simp*]: $1 < \lceil x \rceil \longleftrightarrow 1 < x$
⟨*proof*⟩

lemma *numeral-less-ceiling* [*simp*]: $numeral\ v < \lceil x \rceil \longleftrightarrow numeral\ v < x$
⟨*proof*⟩

lemma *neg-numeral-less-ceiling* [*simp*]: $- numeral\ v < \lceil x \rceil \longleftrightarrow - numeral\ v < x$
⟨*proof*⟩

lemma *ceiling-altdef*: $\lceil x \rceil = (if\ x = of\text{-}int\ \lfloor x \rfloor\ then\ \lfloor x \rfloor\ else\ \lfloor x \rfloor + 1)$
⟨*proof*⟩

lemma *floor-le-ceiling* [*simp*]: $\lfloor x \rfloor \leq \lceil x \rceil$
⟨*proof*⟩

Addition and subtraction of integers.

lemma *ceiling-add-of-int* [*simp*]: $\lceil x + of\text{-}int\ z \rceil = \lceil x \rceil + z$
⟨*proof*⟩

lemma *ceiling-add-numeral* [*simp*]: $\lceil x + numeral\ v \rceil = \lceil x \rceil + numeral\ v$
⟨*proof*⟩

lemma *ceiling-add-one* [*simp*]: $\lceil x + 1 \rceil = \lceil x \rceil + 1$
⟨*proof*⟩

lemma *ceiling-diff-of-int* [*simp*]: $\lceil x - of\text{-}int\ z \rceil = \lceil x \rceil - z$
⟨*proof*⟩

lemma *ceiling-diff-numeral* [*simp*]: $\lceil x - numeral\ v \rceil = \lceil x \rceil - numeral\ v$
⟨*proof*⟩

lemma *ceiling-diff-one* [*simp*]: $\lceil x - 1 \rceil = \lceil x \rceil - 1$
⟨*proof*⟩

lemma *ceiling-split*[*arith-split*]: $P\ \lceil t \rceil \longleftrightarrow (\forall i.\ of\text{-}int\ i - 1 < t \wedge t \leq of\text{-}int\ i \longrightarrow P\ i)$
⟨*proof*⟩

lemma *ceiling-diff-floor-le-1*: $\lceil x \rceil - \lfloor x \rfloor \leq 1$
⟨*proof*⟩

94.5 Negation

lemma *floor-minus*: $\lfloor - x \rfloor = - \lceil x \rceil$
⟨*proof*⟩

lemma *ceiling-minus*: $\lceil - x \rceil = - \lfloor x \rfloor$
⟨*proof*⟩

94.6 Natural numbers

lemma *of-nat-floor*: $r \geq 0 \Longrightarrow$ *of-nat* (*nat* $\lfloor r \rfloor$) $\leq r$
⟨*proof*⟩

lemma *of-nat-ceiling*: *of-nat* (*nat* $\lceil r \rceil$) $\geq r$
⟨*proof*⟩

94.7 Frac Function

definition *frac* :: $'a \Rightarrow 'a$::*floor-ceiling*
where *frac* $x \equiv x -$ *of-int* $\lfloor x \rfloor$

lemma *frac-lt-1*: *frac* $x < 1$
⟨*proof*⟩

lemma *frac-eq-0-iff* [*simp*]: *frac* $x = 0 \longleftrightarrow x \in \mathbb{Z}$
⟨*proof*⟩

lemma *frac-ge-0* [*simp*]: *frac* $x \geq 0$
⟨*proof*⟩

lemma *frac-gt-0-iff* [*simp*]: *frac* $x > 0 \longleftrightarrow x \notin \mathbb{Z}$
⟨*proof*⟩

lemma *frac-of-int* [*simp*]: *frac* (*of-int* z) $= 0$
⟨*proof*⟩

lemma *floor-add*: $\lfloor x + y \rfloor = ($*if frac* $x +$ *frac* $y < 1$ *then* $\lfloor x \rfloor + \lfloor y \rfloor$ *else* $(\lfloor x \rfloor + \lfloor y \rfloor) + 1)$
⟨*proof*⟩

lemma *floor-add2*[*simp*]: $x \in \mathbb{Z} \vee y \in \mathbb{Z} \Longrightarrow \lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor$
⟨*proof*⟩

lemma *frac-add*:
frac $(x + y) = ($*if frac* $x +$ *frac* $y < 1$ *then frac* $x +$ *frac* y *else* (*frac* $x +$ *frac* $y) - 1)$

⟨*proof*⟩

lemma *frac-unique-iff*: $frac\ x = a \longleftrightarrow x - a \in \mathbb{Z} \wedge 0 \leq a \wedge a < 1$
for $x :: 'a::floor\text{-}ceiling$
⟨*proof*⟩

lemma *frac-eq*: $frac\ x = x \longleftrightarrow 0 \leq x \wedge x < 1$
⟨*proof*⟩

lemma *frac-neg*: $frac\ (-\ x) = (if\ x \in \mathbb{Z}\ then\ 0\ else\ 1 - frac\ x)$
for $x :: 'a::floor\text{-}ceiling$
⟨*proof*⟩

94.8 Rounding to the nearest integer

definition $round :: 'a::floor\text{-}ceiling \Rightarrow int$
where $round\ x = \lfloor x + 1/2 \rfloor$

lemma *of-int-round-ge*: $of\text{-}int\ (round\ x) \geq x - 1/2$
and *of-int-round-le*: $of\text{-}int\ (round\ x) \leq x + 1/2$
and *of-int-round-abs-le*: $|of\text{-}int\ (round\ x) - x| \leq 1/2$
and *of-int-round-gt*: $of\text{-}int\ (round\ x) > x - 1/2$
⟨*proof*⟩

lemma *round-of-int* [*simp*]: $round\ (of\text{-}int\ n) = n$
⟨*proof*⟩

lemma *round-0* [*simp*]: $round\ 0 = 0$
⟨*proof*⟩

lemma *round-1* [*simp*]: $round\ 1 = 1$
⟨*proof*⟩

lemma *round-numeral* [*simp*]: $round\ (numeral\ n) = numeral\ n$
⟨*proof*⟩

lemma *round-neg-numeral* [*simp*]: $round\ (-numeral\ n) = -numeral\ n$
⟨*proof*⟩

lemma *round-of-nat* [*simp*]: $round\ (of\text{-}nat\ n) = of\text{-}nat\ n$
⟨*proof*⟩

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

lemma *round-unique*: $of\text{-}int\ y > x - 1/2 \Longrightarrow of\text{-}int\ y \leq x + 1/2 \Longrightarrow round\ x = y$
⟨*proof*⟩

lemma *round-unique'*: $|x - \textit{of-int } n| < 1/2 \Longrightarrow \textit{round } x = n$
⟨*proof*⟩

lemma *round-altdef*: $\textit{round } x = (\textit{if frac } x \geq 1/2 \textit{ then } \lceil x \rceil \textit{ else } \lfloor x \rfloor)$
⟨*proof*⟩

lemma *floor-le-round*: $\lfloor x \rfloor \leq \textit{round } x$
⟨*proof*⟩

lemma *ceiling-ge-round*: $\lceil x \rceil \geq \textit{round } x$
⟨*proof*⟩

lemma *round-diff-minimal*: $|z - \textit{of-int } (\textit{round } z)| \leq |z - \textit{of-int } m|$
for z :: *'a::floor-ceiling*
⟨*proof*⟩

end

95 Rational numbers

theory *Rat*
imports *Archimedean-Field*
begin

95.1 Rational numbers as quotient

95.1.1 Construction of the type of rational numbers

definition *ratrel* :: $(\textit{int} \times \textit{int}) \Rightarrow (\textit{int} \times \textit{int}) \Rightarrow \textit{bool}$
where $\textit{ratrel} = (\lambda x\ y.\ \textit{snd } x \neq 0 \land \textit{snd } y \neq 0 \land \textit{fst } x * \textit{snd } y = \textit{fst } y * \textit{snd } x)$

lemma *ratrel-iff* [*simp*]: $\textit{ratrel } x\ y \longleftrightarrow \textit{snd } x \neq 0 \land \textit{snd } y \neq 0 \land \textit{fst } x * \textit{snd } y = \textit{fst } y * \textit{snd } x$
⟨*proof*⟩

lemma *exists-ratrel-refl*: $\exists x.\ \textit{ratrel } x\ x$
⟨*proof*⟩

lemma *symp-ratrel*: *symp ratrel*
⟨*proof*⟩

lemma *transp-ratrel*: *transp ratrel*
⟨*proof*⟩

lemma *part-equivp-ratrel*: *part-equivp ratrel*
⟨*proof*⟩

quotient-type $\textit{rat} = \textit{int} \times \textit{int}$ / *partial*: *ratrel*
morphisms *Rep-Rat Abs-Rat*

⟨*proof*⟩

lemma *Domainp-cr-rat* [*transfer-domain-rule*]: $Domainp\ pcr\text{-}rat = (\lambda x.\ snd\ x \neq 0)$
⟨*proof*⟩

95.1.2 Representation and basic operations

lift-definition $Fract :: int \Rightarrow int \Rightarrow rat$
is $\lambda a\ b.\ if\ b = 0\ then\ (0,\ 1)\ else\ (a,\ b)$
⟨*proof*⟩

lemma *eq-rat*:
$\bigwedge a\ b\ c\ d.\ b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow Fract\ a\ b = Fract\ c\ d \longleftrightarrow a * d = c * b$
$\bigwedge a.\ Fract\ a\ 0 = Fract\ 0\ 1$
$\bigwedge a\ c.\ Fract\ 0\ a = Fract\ 0\ c$
⟨*proof*⟩

lemma *Rat-cases* [*case-names Fract*, *cases type*: *rat*]:
assumes *that*: $\bigwedge a\ b.\ q = Fract\ a\ b \Longrightarrow b > 0 \Longrightarrow coprime\ a\ b \Longrightarrow C$
shows C
⟨*proof*⟩

lemma *Rat-induct* [*case-names Fract*, *induct type*: *rat*]:
assumes $\bigwedge a\ b.\ b > 0 \Longrightarrow coprime\ a\ b \Longrightarrow P\ (Fract\ a\ b)$
shows $P\ q$
⟨*proof*⟩

instantiation $rat :: field$
begin

lift-definition $zero\text{-}rat :: rat$ **is** $(0,\ 1)$
⟨*proof*⟩

lift-definition $one\text{-}rat :: rat$ **is** $(1,\ 1)$
⟨*proof*⟩

lemma *Zero-rat-def*: $0 = Fract\ 0\ 1$
⟨*proof*⟩

lemma *One-rat-def*: $1 = Fract\ 1\ 1$
⟨*proof*⟩

lift-definition $plus\text{-}rat :: rat \Rightarrow rat \Rightarrow rat$
is $\lambda x\ y.\ (fst\ x * snd\ y + fst\ y * snd\ x,\ snd\ x * snd\ y)$
⟨*proof*⟩

lemma *add-rat* [*simp*]:
assumes $b \neq 0$ **and** $d \neq 0$

shows *Fract a b + Fract c d = Fract (a ∗ d + c ∗ b) (b ∗ d)*
⟨*proof*⟩

lift-definition *uminus-rat* :: *rat ⇒ rat* **is** *λx. (− fst x, snd x)*
⟨*proof*⟩

lemma *minus-rat* [*simp*]: *− Fract a b = Fract (− a) b*
⟨*proof*⟩

lemma *minus-rat-cancel* [*simp*]: *Fract (− a) (− b) = Fract a b*
⟨*proof*⟩

definition *diff-rat-def*: *q − r = q + − r* **for** *q r* :: *rat*

lemma *diff-rat* [*simp*]:
b ≠ 0 ⟹ d ≠ 0 ⟹ Fract a b − Fract c d = Fract (a ∗ d − c ∗ b) (b ∗ d)
⟨*proof*⟩

lift-definition *times-rat* :: *rat ⇒ rat ⇒ rat*
is *λx y. (fst x ∗ fst y, snd x ∗ snd y)*
⟨*proof*⟩

lemma *mult-rat* [*simp*]: *Fract a b ∗ Fract c d = Fract (a ∗ c) (b ∗ d)*
⟨*proof*⟩

lemma *mult-rat-cancel*: *c ≠ 0 ⟹ Fract (c ∗ a) (c ∗ b) = Fract a b*
⟨*proof*⟩

lift-definition *inverse-rat* :: *rat ⇒ rat*
is *λx. if fst x = 0 then (0, 1) else (snd x, fst x)*
⟨*proof*⟩

lemma *inverse-rat* [*simp*]: *inverse (Fract a b) = Fract b a*
⟨*proof*⟩

definition *divide-rat-def*: *q div r = q ∗ inverse r* **for** *q r* :: *rat*

lemma *divide-rat* [*simp*]: *Fract a b div Fract c d = Fract (a ∗ d) (b ∗ c)*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *div-add-self1-no-field* [*simp*]:
assumes *NO-MATCH (x :: 'b :: field) b (b :: 'a :: semiring-div) ≠ 0*
shows *(b + a) div b = a div b + 1*

⟨*proof*⟩

lemma *div-add-self2-no-field* [*simp*]:
assumes *NO-MATCH* $(x :: 'b :: field)$ b $(b :: 'a :: semiring\text{-}div) \neq 0$
shows $(a + b)$ *div* $b = a$ *div* $b + 1$
⟨*proof*⟩

lemma *of-nat-rat*: *of-nat* k = *Fract* (*of-nat* k) 1
⟨*proof*⟩

lemma *of-int-rat*: *of-int* k = *Fract* k 1
⟨*proof*⟩

lemma *Fract-of-nat-eq*: *Fract* (*of-nat* k) 1 = *of-nat* k
⟨*proof*⟩

lemma *Fract-of-int-eq*: *Fract* k 1 = *of-int* k
⟨*proof*⟩

lemma *rat-number-collapse*:
Fract 0 $k = 0$
Fract 1 $1 = 1$
Fract (*numeral* w) 1 = *numeral* w
Fract ($-$ *numeral* w) 1 = $-$ *numeral* w
Fract $(-\ 1)$ $1 = -\ 1$
Fract k $0 = 0$
⟨*proof*⟩

lemma *rat-number-expand*:
0 = *Fract* 0 1
1 = *Fract* 1 1
numeral k = *Fract* (*numeral* k) 1
$-\ 1$ = *Fract* $(-\ 1)$ 1
$-$ *numeral* k = *Fract* ($-$ *numeral* k) 1
⟨*proof*⟩

lemma *Rat-cases-nonzero* [*case-names Fract 0*]:
assumes *Fract*: $\bigwedge a\ b.\ q = Fract\ a\ b \Longrightarrow b > 0 \Longrightarrow a \neq 0 \Longrightarrow coprime\ a\ b \Longrightarrow C$
and 0: $q = 0 \Longrightarrow C$
shows C
⟨*proof*⟩

95.1.3 Function *normalize*

lemma *Fract-coprime*: *Fract* (a *div gcd* a b) (b *div gcd* a b) = *Fract* a b
⟨*proof*⟩

definition *normalize* :: $int \times int \Rightarrow int \times int$

where *normalize p* =
(*if snd p* > *0 then* (*let a* = *gcd* (*fst p*) (*snd p*) *in* (*fst p div a*, *snd p div a*))
else if snd p = *0 then* (*0*, *1*)
else (*let a* = − *gcd* (*fst p*) (*snd p*) *in* (*fst p div a*, *snd p div a*)))

lemma *normalize-crossproduct*:
assumes $q \neq 0$ $s \neq 0$
assumes *normalize* (*p*, *q*) = *normalize* (*r*, *s*)
shows $p * s = r * q$
⟨*proof*⟩

lemma *normalize-eq*: *normalize* (*a*, *b*) = (*p*, *q*) ⟹ *Fract p q* = *Fract a b*
⟨*proof*⟩

lemma *normalize-denom-pos*: *normalize r* = (*p*, *q*) ⟹ $q > 0$
⟨*proof*⟩

lemma *normalize-coprime*: *normalize r* = (*p*, *q*) ⟹ *coprime p q*
⟨*proof*⟩

lemma *normalize-stable* [*simp*]: $q > 0$ ⟹ *coprime p q* ⟹ *normalize* (*p*, *q*) = (*p*, *q*)
⟨*proof*⟩

lemma *normalize-denom-zero* [*simp*]: *normalize* (*p*, *0*) = (*0*, *1*)
⟨*proof*⟩

lemma *normalize-negative* [*simp*]: $q < 0$ ⟹ *normalize* (*p*, *q*) = *normalize* (− *p*, − *q*)
⟨*proof*⟩

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

definition *quotient-of* :: *rat* ⇒ *int* × *int*
where *quotient-of x* =
(*THE pair*. *x* = *Fract* (*fst pair*) (*snd pair*) ∧ *snd pair* > *0* ∧ *coprime* (*fst pair*) (*snd pair*))

lemma *quotient-of-unique*: ∃!*p*. *r* = *Fract* (*fst p*) (*snd p*) ∧ *snd p* > *0* ∧ *coprime* (*fst p*) (*snd p*)
⟨*proof*⟩

lemma *quotient-of-Fract* [*code*]: *quotient-of* (*Fract a b*) = *normalize* (*a*, *b*)
⟨*proof*⟩

lemma *quotient-of-number* [*simp*]:
quotient-of 0 = (*0*, *1*)
quotient-of 1 = (*1*, *1*)
quotient-of (*numeral k*) = (*numeral k*, *1*)

quotient-of $(- 1) = (- 1, 1)$
quotient-of $(-$ *numeral* $k) = (-$ *numeral* $k, 1)$
⟨*proof*⟩

lemma *quotient-of-eq*: *quotient-of* (*Fract* a b) $= (p, q) \Longrightarrow$ *Fract* p q $=$ *Fract* a b
⟨*proof*⟩

lemma *quotient-of-denom-pos*: *quotient-of* $r = (p, q) \Longrightarrow q > 0$
⟨*proof*⟩

lemma *quotient-of-denom-pos'*: *snd* (*quotient-of* r) > 0
⟨*proof*⟩

lemma *quotient-of-coprime*: *quotient-of* $r = (p, q) \Longrightarrow$ *coprime* p q
⟨*proof*⟩

lemma *quotient-of-inject*:
assumes *quotient-of* $a =$ *quotient-of* b
shows $a = b$
⟨*proof*⟩

lemma *quotient-of-inject-eq*: *quotient-of* $a =$ *quotient-of* $b \longleftrightarrow a = b$
⟨*proof*⟩

95.1.4 Various

lemma *Fract-of-int-quotient*: *Fract* k l $=$ *of-int* k / *of-int* l
⟨*proof*⟩

lemma *Fract-add-one*: $n \neq 0 \Longrightarrow$ *Fract* $(m + n)$ n $=$ *Fract* m $n + 1$
⟨*proof*⟩

lemma *quotient-of-div*:
assumes r: *quotient-of* $r = (n,d)$
shows $r =$ *of-int* n / *of-int* d
⟨*proof*⟩

95.1.5 The ordered field of rational numbers

lift-definition *positive* :: *rat* $\Rightarrow$ *bool*
is $\lambda x.$ $0 <$ *fst* $x *$ *snd* x
⟨*proof*⟩

lemma *positive-zero*: $\neg$ *positive* 0
⟨*proof*⟩

lemma *positive-add*: *positive* $x \Longrightarrow$ *positive* $y \Longrightarrow$ *positive* $(x + y)$
⟨*proof*⟩

lemma *positive-mult*: *positive* $x \Longrightarrow$ *positive* $y \Longrightarrow$ *positive* $(x * y)$

⟨*proof*⟩

lemma *positive-minus*: $\neg$ *positive* $x \Longrightarrow x \neq 0 \Longrightarrow$ *positive* $(- x)$
⟨*proof*⟩

instantiation *rat* :: *linordered-field*
begin

definition $x < y \longleftrightarrow$ *positive* $(y - x)$

definition $x \leq y \longleftrightarrow x < y \vee x = y$ **for** $x\ y$ :: *rat*

definition $|a| = ($*if* $a < 0$ *then* $- a$ *else* $a)$ **for** a :: *rat*

definition *sgn* $a = ($*if* $a = 0$ *then* 0 *else if* $0 < a$ *then* 1 *else* $- 1)$ **for** a :: *rat*

instance
⟨*proof*⟩

end

instantiation *rat* :: *distrib-lattice*
begin

definition $($*inf* :: *rat* $\Rightarrow$ *rat* $\Rightarrow$ *rat*$)$ = *min*

definition $($*sup* :: *rat* $\Rightarrow$ *rat* $\Rightarrow$ *rat*$)$ = *max*

instance
⟨*proof*⟩

end

lemma *positive-rat*: *positive* $($*Fract* $a\ b) \longleftrightarrow 0 < a * b$
⟨*proof*⟩

lemma *less-rat* [*simp*]:
$b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow$ *Fract* $a\ b <$ *Fract* $c\ d \longleftrightarrow (a * d) * (b * d) < (c * b) * (b * d)$
⟨*proof*⟩

lemma *le-rat* [*simp*]:
$b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow$ *Fract* $a\ b \leq$ *Fract* $c\ d \longleftrightarrow (a * d) * (b * d) \leq (c * b) * (b * d)$
⟨*proof*⟩

lemma *abs-rat* [*simp*, *code*]: $|$*Fract* $a\ b| =$ *Fract* $|a|\ |b|$
⟨*proof*⟩

lemma *sgn-rat* [*simp*, *code*]: $sgn\ (Fract\ a\ b) = of\text{-}int\ (sgn\ a * sgn\ b)$
⟨*proof*⟩

lemma *Rat-induct-pos* [*case-names Fract*, *induct type*: *rat*]:
assumes *step*: $\bigwedge a\ b.\ 0 < b \Longrightarrow P\ (Fract\ a\ b)$
shows $P\ q$
⟨*proof*⟩

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

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

lemma *zero-le-Fract-iff*: $0 < b \Longrightarrow 0 \leq Fract\ a\ b \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *Fract-le-zero-iff*: $0 < b \Longrightarrow Fract\ a\ b \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *one-less-Fract-iff*: $0 < b \Longrightarrow 1 < Fract\ a\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *Fract-less-one-iff*: $0 < b \Longrightarrow Fract\ a\ b < 1 \longleftrightarrow a < b$
⟨*proof*⟩

lemma *one-le-Fract-iff*: $0 < b \Longrightarrow 1 \leq Fract\ a\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

lemma *Fract-le-one-iff*: $0 < b \Longrightarrow Fract\ a\ b \leq 1 \longleftrightarrow a \leq b$
⟨*proof*⟩

95.1.6 Rationals are an Archimedean field

lemma *rat-floor-lemma*: $of\text{-}int\ (a\ div\ b) \leq Fract\ a\ b \wedge Fract\ a\ b < of\text{-}int\ (a\ div\ b + 1)$
⟨*proof*⟩

instance *rat* :: *archimedean-field*
⟨*proof*⟩

instantiation *rat* :: *floor-ceiling*
begin

definition [*code del*]: $\lfloor x\rfloor = (THE\ z.\ of\text{-}int\ z \leq x \wedge x < of\text{-}int\ (z + 1))$ **for** x :: *rat*

instance
⟨*proof*⟩

end

lemma *floor-Fract*: $\lfloor Fract\ a\ b \rfloor = a\ div\ b$
⟨*proof*⟩

95.2 Linear arithmetic setup

⟨*ML*⟩

95.3 Embedding from Rationals to other Fields

context *field-char-0*
begin

lift-definition *of-rat* :: $rat \Rightarrow 'a$
is $\lambda x.\ of\text{-}int\ (fst\ x)\ /\ of\text{-}int\ (snd\ x)$
⟨*proof*⟩

end

lemma *of-rat-rat*: $b \neq 0 \Longrightarrow of\text{-}rat\ (Fract\ a\ b) = of\text{-}int\ a\ /\ of\text{-}int\ b$
⟨*proof*⟩

lemma *of-rat-0* [*simp*]: $of\text{-}rat\ 0 = 0$
⟨*proof*⟩

lemma *of-rat-1* [*simp*]: $of\text{-}rat\ 1 = 1$
⟨*proof*⟩

lemma *of-rat-add*: $of\text{-}rat\ (a + b) = of\text{-}rat\ a + of\text{-}rat\ b$
⟨*proof*⟩

lemma *of-rat-minus*: $of\text{-}rat\ (-\ a) = -\ of\text{-}rat\ a$
⟨*proof*⟩

lemma *of-rat-neg-one* [*simp*]: $of\text{-}rat\ (-\ 1) = -\ 1$
⟨*proof*⟩

lemma *of-rat-diff*: $of\text{-}rat\ (a - b) = of\text{-}rat\ a - of\text{-}rat\ b$
⟨*proof*⟩

lemma *of-rat-mult*: $of\text{-}rat\ (a * b) = of\text{-}rat\ a * of\text{-}rat\ b$
⟨*proof*⟩

lemma *of-rat-sum*: $of\text{-}rat\ (\sum a \in A.\ f\ a) = (\sum a \in A.\ of\text{-}rat\ (f\ a))$
⟨*proof*⟩

lemma *of-rat-prod*: $of\text{-}rat\ (\prod a \in A.\ f\ a) = (\prod a \in A.\ of\text{-}rat\ (f\ a))$
⟨*proof*⟩

lemma *nonzero-of-rat-inverse*: $a \neq 0 \Longrightarrow$ *of-rat* (*inverse* a) = *inverse* (*of-rat* a)
⟨*proof*⟩

lemma *of-rat-inverse*: (*of-rat* (*inverse* a) :: $'a$::{*field-char-0*,*field*}) = *inverse* (*of-rat* a)
⟨*proof*⟩

lemma *nonzero-of-rat-divide*: $b \neq 0 \Longrightarrow$ *of-rat* (a / b) = *of-rat* a / *of-rat* b
⟨*proof*⟩

lemma *of-rat-divide*: (*of-rat* (a / b) :: $'a$::{*field-char-0*,*field*}) = *of-rat* a / *of-rat* b
⟨*proof*⟩

lemma *of-rat-power*: (*of-rat* (a ^ n) :: $'a$::*field-char-0*) = *of-rat* a ^ n
⟨*proof*⟩

lemma *of-rat-eq-iff* [*simp*]: *of-rat* a = *of-rat* $b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *of-rat-eq-0-iff* [*simp*]: *of-rat* $a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-eq-of-rat-iff* [*simp*]: 0 = *of-rat* $a \longleftrightarrow 0 = a$
⟨*proof*⟩

lemma *of-rat-eq-1-iff* [*simp*]: *of-rat* $a = 1 \longleftrightarrow a = 1$
⟨*proof*⟩

lemma *one-eq-of-rat-iff* [*simp*]: 1 = *of-rat* $a \longleftrightarrow 1 = a$
⟨*proof*⟩

lemma *of-rat-less*: (*of-rat* r :: $'a$::*linordered-field*) < *of-rat* $s \longleftrightarrow r < s$
⟨*proof*⟩

lemma *of-rat-less-eq*: (*of-rat* r :: $'a$::*linordered-field*) $\leq$ *of-rat* $s \longleftrightarrow r \leq s$
⟨*proof*⟩

lemma *of-rat-le-0-iff* [*simp*]: (*of-rat* r :: $'a$::*linordered-field*) $\leq 0 \longleftrightarrow r \leq 0$
⟨*proof*⟩

lemma *zero-le-of-rat-iff* [*simp*]: $0 \leq$ (*of-rat* r :: $'a$::*linordered-field*) $\longleftrightarrow 0 \leq r$
⟨*proof*⟩

lemma *of-rat-le-1-iff* [*simp*]: (*of-rat* r :: $'a$::*linordered-field*) $\leq 1 \longleftrightarrow r \leq 1$
⟨*proof*⟩

lemma *one-le-of-rat-iff* [*simp*]: $1 \leq$ (*of-rat* r :: $'a$::*linordered-field*) $\longleftrightarrow 1 \leq r$

⟨*proof*⟩

lemma *of-rat-less-0-iff* [*simp*]: (*of-rat r* :: *'a::linordered-field*) < *0* ⟷ *r* < *0*
⟨*proof*⟩

lemma *zero-less-of-rat-iff* [*simp*]: *0* < (*of-rat r* :: *'a::linordered-field*) ⟷ *0* < *r*
⟨*proof*⟩

lemma *of-rat-less-1-iff* [*simp*]: (*of-rat r* :: *'a::linordered-field*) < *1* ⟷ *r* < *1*
⟨*proof*⟩

lemma *one-less-of-rat-iff* [*simp*]: *1* < (*of-rat r* :: *'a::linordered-field*) ⟷ *1* < *r*
⟨*proof*⟩

lemma *of-rat-eq-id* [*simp*]: *of-rat* = *id*
⟨*proof*⟩

Collapse nested embeddings.

lemma *of-rat-of-nat-eq* [*simp*]: *of-rat* (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *of-rat-of-int-eq* [*simp*]: *of-rat* (*of-int z*) = *of-int z*
⟨*proof*⟩

lemma *of-rat-numeral-eq* [*simp*]: *of-rat* (*numeral w*) = *numeral w*
⟨*proof*⟩

lemma *of-rat-neg-numeral-eq* [*simp*]: *of-rat* (− *numeral w*) = − *numeral w*
⟨*proof*⟩

lemmas *zero-rat* = *Zero-rat-def*
lemmas *one-rat* = *One-rat-def*

abbreviation *rat-of-nat* :: *nat* ⇒ *rat*
where *rat-of-nat* ≡ *of-nat*

abbreviation *rat-of-int* :: *int* ⇒ *rat*
where *rat-of-int* ≡ *of-int*

95.4 The Set of Rational Numbers

context *field-char-0*
begin

definition *Rats* :: *'a set* (ℚ)
where ℚ = *range of-rat*

end

lemma *Rats-of-rat* [*simp*]: *of-rat* $r \in \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-of-int* [*simp*]: *of-int* $z \in \mathbb{Q}$
⟨*proof*⟩

lemma *Ints-subset-Rats*: $\mathbb{Z} \subseteq \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-of-nat* [*simp*]: *of-nat* $n \in \mathbb{Q}$
⟨*proof*⟩

lemma *Nats-subset-Rats*: $\mathbb{N} \subseteq \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-number-of* [*simp*]: *numeral* $w \in \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-0* [*simp*]: $0 \in \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-1* [*simp*]: $1 \in \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-add* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a + b \in \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-minus* [*simp*]: $a \in \mathbb{Q} \Longrightarrow - a \in \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-diff* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a - b \in \mathbb{Q}$
⟨*proof*⟩

lemma *Rats-mult* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a * b \in \mathbb{Q}$
⟨*proof*⟩

lemma *nonzero-Rats-inverse*: $a \in \mathbb{Q} \Longrightarrow a \neq 0 \Longrightarrow$ *inverse* $a \in \mathbb{Q}$
for *a* :: *'a::field-char-0*
⟨*proof*⟩

lemma *Rats-inverse* [*simp*]: $a \in \mathbb{Q} \Longrightarrow$ *inverse* $a \in \mathbb{Q}$
for *a* :: *'a::{field-char-0,field}*
⟨*proof*⟩

lemma *nonzero-Rats-divide*: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow b \neq 0 \Longrightarrow a / b \in \mathbb{Q}$
for *a b* :: *'a::field-char-0*
⟨*proof*⟩

lemma *Rats-divide* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a / b \in \mathbb{Q}$

for $a\ b$ :: $'a$::{*field-char-0*, *field*}
⟨*proof*⟩

lemma *Rats-power* [*simp*]: $a \in \mathbb{Q} \Longrightarrow a \hat{\ } n \in \mathbb{Q}$
for a :: $'a$::*field-char-0*
⟨*proof*⟩

lemma *Rats-cases* [*cases set*: *Rats*]:
assumes $q \in \mathbb{Q}$
obtains (*of-rat*) r **where** q = *of-rat* r
⟨*proof*⟩

lemma *Rats-induct* [*case-names of-rat*, *induct set*: *Rats*]: $q \in \mathbb{Q} \Longrightarrow (\bigwedge r.\ P$ (*of-rat* $r)) \Longrightarrow P\ q$
⟨*proof*⟩

lemma *Rats-infinite*: ¬ *finite* $\mathbb{Q}$
⟨*proof*⟩

95.5 Implementation of rational numbers as pairs of integers

Formal constructor

definition *Frct* :: *int* × *int* ⇒ *rat*
where [*simp*]: *Frct p* = *Fract* (*fst p*) (*snd p*)

lemma [*code abstype*]: *Frct* (*quotient-of q*) = *q*
⟨*proof*⟩

Numerals

declare *quotient-of-Fract* [*code abstract*]

definition *of-int* :: *int* ⇒ *rat*
where [*code-abbrev*]: *of-int* = *Int.of-int*

hide-const (**open**) *of-int*

lemma *quotient-of-int* [*code abstract*]: *quotient-of* (*Rat.of-int a*) = (*a*, *1*)
⟨*proof*⟩

lemma [*code-unfold*]: *numeral k* = *Rat.of-int* (*numeral k*)
⟨*proof*⟩

lemma [*code-unfold*]: − *numeral k* = *Rat.of-int* (− *numeral k*)
⟨*proof*⟩

lemma *Frct-code-post* [*code-post*]:
Frct (*0*, *a*) = *0*
Frct (*a*, *0*) = *0*
Frct (*1*, *1*) = *1*

Frct (*numeral k*, *1*) = *numeral k*
Frct (*1*, *numeral k*) = *1* / *numeral k*
Frct (*numeral k*, *numeral l*) = *numeral k* / *numeral l*
Frct (− *a*, *b*) = − *Frct* (*a*, *b*)
Frct (*a*, − *b*) = − *Frct* (*a*, *b*)
− (− *Frct q*) = *Frct q*
⟨*proof*⟩

Operations

lemma *rat-zero-code* [*code abstract*]: *quotient-of 0* = (*0*, *1*)
⟨*proof*⟩

lemma *rat-one-code* [*code abstract*]: *quotient-of 1* = (*1*, *1*)
⟨*proof*⟩

lemma *rat-plus-code* [*code abstract*]:
quotient-of (*p* + *q*) = (*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *d* + *b* ∗ *c*, *c* ∗ *d*))
⟨*proof*⟩

lemma *rat-uminus-code* [*code abstract*]:
quotient-of (− *p*) = (*let* (*a*, *b*) = *quotient-of p in* (− *a*, *b*))
⟨*proof*⟩

lemma *rat-minus-code* [*code abstract*]:
quotient-of (*p* − *q*) =
(*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *d* − *b* ∗ *c*, *c* ∗ *d*))
⟨*proof*⟩

lemma *rat-times-code* [*code abstract*]:
quotient-of (*p* ∗ *q*) =
(*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *b*, *c* ∗ *d*))
⟨*proof*⟩

lemma *rat-inverse-code* [*code abstract*]:
quotient-of (*inverse p*) =
(*let* (*a*, *b*) = *quotient-of p*
in if a = *0 then* (*0*, *1*) *else* (*sgn a* ∗ *b*, |*a*|))
⟨*proof*⟩

lemma *rat-divide-code* [*code abstract*]:
quotient-of (*p* / *q*) =
(*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *d*, *c* ∗ *b*))
⟨*proof*⟩

lemma *rat-abs-code* [*code abstract*]: *quotient-of* |*p*| = (*let* (*a*, *b*) = *quotient-of p*

in $(|a|,\ b))$
⟨*proof*⟩

lemma *rat-sgn-code* [*code abstract*]: *quotient-of* $(sgn\ p) = (sgn\ (fst\ (quotient\text{-}of\ p)),\ 1)$
⟨*proof*⟩

lemma *rat-floor-code* [*code*]: $\lfloor p \rfloor = (let\ (a,\ b) = quotient\text{-}of\ p\ in\ a\ div\ b)$
⟨*proof*⟩

instantiation *rat* :: *equal*
begin

definition [*code*]: $HOL.equal\ a\ b \longleftrightarrow quotient\text{-}of\ a = quotient\text{-}of\ b$

instance
⟨*proof*⟩

lemma *rat-eq-refl* [*code nbe*]: $HOL.equal\ (r::rat)\ r \longleftrightarrow True$
⟨*proof*⟩

end

lemma *rat-less-eq-code* [*code*]:
$p \le q \longleftrightarrow (let\ (a,\ c) = quotient\text{-}of\ p;\ (b,\ d) = quotient\text{-}of\ q\ in\ a * d \le c * b)$
⟨*proof*⟩

lemma *rat-less-code* [*code*]:
$p < q \longleftrightarrow (let\ (a,\ c) = quotient\text{-}of\ p;\ (b,\ d) = quotient\text{-}of\ q\ in\ a * d < c * b)$
⟨*proof*⟩

lemma [*code*]: $of\text{-}rat\ p = (let\ (a,\ b) = quotient\text{-}of\ p\ in\ of\text{-}int\ a\ /\ of\text{-}int\ b)$
⟨*proof*⟩

Quickcheck

definition (**in** *term-syntax*)
valterm-fract :: $int \times (unit \Rightarrow Code\text{-}Evaluation.term) \Rightarrow$
$int \times (unit \Rightarrow Code\text{-}Evaluation.term) \Rightarrow$
$rat \times (unit \Rightarrow Code\text{-}Evaluation.term)$
where [*code-unfold*]: $valterm\text{-}fract\ k\ l = Code\text{-}Evaluation.valtermify\ Fract\ \{\cdot\}\ k\ \{\cdot\}\ l$

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

instantiation *rat* :: *random*
begin

definition

Quickcheck-Random.random i =
  Quickcheck-Random.random i ∘→ (λ*num*. *Random.range i* ∘→ (λ*denom*. *Pair*
    (*let j* = *int-of-integer* (*integer-of-natural* (*denom* + *1*))
     in valterm-fract num (*j*, λ*u*. *Code-Evaluation.term-of j*))))

instance ⟨*proof*⟩

end

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

instantiation *rat* :: *exhaustive*
begin

definition
  exhaustive-rat f d =
    Quickcheck-Exhaustive.exhaustive
      (λ*l*. *Quickcheck-Exhaustive.exhaustive*
        (λ*k*. *f* (*Fract k* (*int-of-integer* (*integer-of-natural l*) + *1*))) *d*) *d*

instance ⟨*proof*⟩

end

instantiation *rat* :: *full-exhaustive*
begin

definition
  full-exhaustive-rat f d =
    Quickcheck-Exhaustive.full-exhaustive
      (λ(*l*, -). *Quickcheck-Exhaustive.full-exhaustive*
        (λ*k*. *f*
          (*let j* = *int-of-integer* (*integer-of-natural l*) + *1*
           in valterm-fract k (*j*, λ-. *Code-Evaluation.term-of j*))) *d*) *d*

instance ⟨*proof*⟩

end

instance *rat* :: *partial-term-of* ⟨*proof*⟩

lemma [*code*]:
  partial-term-of (*ty* :: *rat itself*) (*Quickcheck-Narrowing.Narrowing-variable p tt*)
≡
    Code-Evaluation.Free (*STR* ''-'') (*Typerep.Typerep* (*STR* ''Rat.rat'') [])
  partial-term-of (*ty* :: *rat itself*) (*Quickcheck-Narrowing.Narrowing-constructor 0*
[*l*, *k*]) ≡
    Code-Evaluation.App

(*Code-Evaluation.Const* (*STR* ''*Rat.Frct*'')
(*Typerep.Typerep* (*STR* ''*fun*'')
[*Typerep.Typerep* (*STR* ''*Product-Type.prod*'')
[*Typerep.Typerep* (*STR* ''*Int.int*'') [], *Typerep.Typerep* (*STR* ''*Int.int*'')
[]],
Typerep.Typerep (*STR* ''*Rat.rat*'') []]))
(*Code-Evaluation.App*
(*Code-Evaluation.App*
(*Code-Evaluation.Const* (*STR* ''*Product-Type.Pair*'')
(*Typerep.Typerep* (*STR* ''*fun*'')
[*Typerep.Typerep* (*STR* ''*Int.int*'') [],
Typerep.Typerep (*STR* ''*fun*'')
[*Typerep.Typerep* (*STR* ''*Int.int*'') [],
Typerep.Typerep (*STR* ''*Product-Type.prod*'')
[*Typerep.Typerep* (*STR* ''*Int.int*'') [], *Typerep.Typerep* (*STR* ''*Int.int*'')
[]]]]]))
(*partial-term-of* (*TYPE*(*int*)) *l*)) (*partial-term-of* (*TYPE*(*int*)) *k*))
⟨*proof*⟩

instantiation *rat* :: *narrowing*
begin

definition
narrowing =
Quickcheck-Narrowing.apply
(*Quickcheck-Narrowing.apply*
(*Quickcheck-Narrowing.cons* (λ*nom denom. Fract nom denom*)) *narrowing*)
narrowing

instance ⟨*proof*⟩

end

95.6 Setup for Nitpick

⟨*ML*⟩

lemmas [*nitpick-unfold*] =
inverse-rat-inst.inverse-rat
one-rat-inst.one-rat ord-rat-inst.less-rat
ord-rat-inst.less-eq-rat plus-rat-inst.plus-rat times-rat-inst.times-rat
uminus-rat-inst.uminus-rat zero-rat-inst.zero-rat

95.7 Float syntax

syntax *-Float* :: *float-const* ⇒ '*a* (-)

⟨*ML*⟩

Test:

lemma $123.456 = -111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::rat)$
⟨*proof*⟩

95.8 Hiding implementation details

hide-const (**open**) *normalize positive*

lifting-update *rat.lifting*
lifting-forget *rat.lifting*

end

96 Development of the Reals using Cauchy Sequences

theory *Real*
imports *Rat*
begin

This theory contains a formalization of the real numbers as equivalence classes of Cauchy sequences of rationals. See `~~/src/HOL/ex/Dedekind_Real.thy` for an alternative construction using Dedekind cuts.

96.1 Preliminary lemmas

lemma *inj-add-left* [*simp*]: $inj\ (op + x)$
for $x :: 'a::cancel\text{-}semigroup\text{-}add$
⟨*proof*⟩

lemma *inj-mult-left* [*simp*]: $inj\ (op * x) \longleftrightarrow x \neq 0$
for $x :: 'a::idom$
⟨*proof*⟩

lemma *add-diff-add*: $(a + c) - (b + d) = (a - b) + (c - d)$
for $a\ b\ c\ d :: 'a::ab\text{-}group\text{-}add$
⟨*proof*⟩

lemma *minus-diff-minus*: $- a - - b = - (a - b)$
for $a\ b :: 'a::ab\text{-}group\text{-}add$
⟨*proof*⟩

lemma *mult-diff-mult*: $(x * y - a * b) = x * (y - b) + (x - a) * b$
for $x\ y\ a\ b :: 'a::ring$
⟨*proof*⟩

lemma *inverse-diff-inverse*:
fixes $a\ b :: 'a::division\text{-}ring$
assumes $a \neq 0$ **and** $b \neq 0$

shows *inverse a* − *inverse b* = − (*inverse a* ∗ (*a* − *b*) ∗ *inverse b*)
⟨*proof*⟩

lemma *obtain-pos-sum*:
fixes *r* :: *rat* **assumes** *r*: *0* < *r*
obtains *s t* **where** *0* < *s* **and** *0* < *t* **and** *r* = *s* + *t*
⟨*proof*⟩

96.2 Sequences that converge to zero

definition *vanishes* :: (*nat* ⇒ *rat*) ⇒ *bool*
where *vanishes X* ⟷ (∀ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. |*X n*| < *r*)

lemma *vanishesI*: (⋀*r*. *0* < *r* ⟹ ∃ *k*. ∀ *n*≥*k*. |*X n*| < *r*) ⟹ *vanishes X*
⟨*proof*⟩

lemma *vanishesD*: *vanishes X* ⟹ *0* < *r* ⟹ ∃ *k*. ∀ *n*≥*k*. |*X n*| < *r*
⟨*proof*⟩

lemma *vanishes-const* [*simp*]: *vanishes* (λ*n*. *c*) ⟷ *c* = *0*
⟨*proof*⟩

lemma *vanishes-minus*: *vanishes X* ⟹ *vanishes* (λ*n*. − *X n*)
⟨*proof*⟩

lemma *vanishes-add*:
assumes *X*: *vanishes X*
and *Y*: *vanishes Y*
shows *vanishes* (λ*n*. *X n* + *Y n*)
⟨*proof*⟩

lemma *vanishes-diff*:
assumes *vanishes X vanishes Y*
shows *vanishes* (λ*n*. *X n* − *Y n*)
⟨*proof*⟩

lemma *vanishes-mult-bounded*:
assumes *X*: ∃ *a*>*0*. ∀ *n*. |*X n*| < *a*
assumes *Y*: *vanishes* (λ*n*. *Y n*)
shows *vanishes* (λ*n*. *X n* ∗ *Y n*)
⟨*proof*⟩

96.3 Cauchy sequences

definition *cauchy* :: (*nat* ⇒ *rat*) ⇒ *bool*
where *cauchy X* ⟷ (∀ *r*>*0*. ∃ *k*. ∀ *m*≥*k*. ∀ *n*≥*k*. |*X m* − *X n*| < *r*)

lemma *cauchyI*: (⋀*r*. *0* < *r* ⟹ ∃ *k*. ∀ *m*≥*k*. ∀ *n*≥*k*. |*X m* − *X n*| < *r*) ⟹ *cauchy X*
⟨*proof*⟩

lemma *cauchyD*: $cauchy\ X \Longrightarrow 0 < r \Longrightarrow \exists\, k.\ \forall\, m{\geq}k.\ \forall\, n{\geq}k.\ |X\ m - X\ n| < r$
⟨*proof*⟩

lemma *cauchy-const* [*simp*]: $cauchy\ (\lambda n.\ x)$
⟨*proof*⟩

lemma *cauchy-add* [*simp*]:
assumes X: $cauchy\ X$ **and** Y: $cauchy\ Y$
shows $cauchy\ (\lambda n.\ X\ n + Y\ n)$
⟨*proof*⟩

lemma *cauchy-minus* [*simp*]:
assumes X: $cauchy\ X$
shows $cauchy\ (\lambda n.\ - X\ n)$
⟨*proof*⟩

lemma *cauchy-diff* [*simp*]:
assumes $cauchy\ X\ cauchy\ Y$
shows $cauchy\ (\lambda n.\ X\ n - Y\ n)$
⟨*proof*⟩

lemma *cauchy-imp-bounded*:
assumes $cauchy\ X$
shows $\exists\, b{>}0.\ \forall\, n.\ |X\ n| < b$
⟨*proof*⟩

lemma *cauchy-mult* [*simp*]:
assumes X: $cauchy\ X$ **and** Y: $cauchy\ Y$
shows $cauchy\ (\lambda n.\ X\ n * Y\ n)$
⟨*proof*⟩

lemma *cauchy-not-vanishes-cases*:
assumes X: $cauchy\ X$
assumes nz: $\neg\ vanishes\ X$
shows $\exists\, b{>}0.\ \exists\, k.\ (\forall\, n{\geq}k.\ b < - X\ n) \vee (\forall\, n{\geq}k.\ b < X\ n)$
⟨*proof*⟩

lemma *cauchy-not-vanishes*:
assumes X: $cauchy\ X$
and nz: $\neg\ vanishes\ X$
shows $\exists\, b{>}0.\ \exists\, k.\ \forall\, n{\geq}k.\ b < |X\ n|$
⟨*proof*⟩

lemma *cauchy-inverse* [*simp*]:
assumes X: $cauchy\ X$
and nz: $\neg\ vanishes\ X$
shows $cauchy\ (\lambda n.\ inverse\ (X\ n))$
⟨*proof*⟩

lemma *vanishes-diff-inverse*:
  assumes X: *cauchy* X ¬ *vanishes* X
    and Y: *cauchy* Y ¬ *vanishes* Y
    and XY: *vanishes* $(\lambda n.\ X\ n - Y\ n)$
  shows *vanishes* $(\lambda n.\ inverse\ (X\ n) - inverse\ (Y\ n))$
⟨*proof*⟩

96.4 Equivalence relation on Cauchy sequences

definition *realrel* :: $(nat \Rightarrow rat) \Rightarrow (nat \Rightarrow rat) \Rightarrow bool$
  where $realrel = (\lambda X\ Y.\ cauchy\ X \wedge cauchy\ Y \wedge vanishes\ (\lambda n.\ X\ n - Y\ n))$

lemma *realrelI* [*intro?*]: $cauchy\ X \Longrightarrow cauchy\ Y \Longrightarrow vanishes\ (\lambda n.\ X\ n - Y\ n) \Longrightarrow realrel\ X\ Y$
  ⟨*proof*⟩

lemma *realrel-refl*: $cauchy\ X \Longrightarrow realrel\ X\ X$
  ⟨*proof*⟩

lemma *symp-realrel*: *symp realrel*
  ⟨*proof*⟩

lemma *transp-realrel*: *transp realrel*
  ⟨*proof*⟩

lemma *part-equivp-realrel*: *part-equivp realrel*
  ⟨*proof*⟩

96.5 The field of real numbers

quotient-type $real = nat \Rightarrow rat$ / *partial*: *realrel*
  morphisms *rep-real Real*
  ⟨*proof*⟩

lemma *cr-real-eq*: $pcr\text{-}real = (\lambda x\ y.\ cauchy\ x \wedge Real\ x = y)$
  ⟨*proof*⟩

lemma *Real-induct* [*induct type*: *real*]:
  assumes $\bigwedge X.\ cauchy\ X \Longrightarrow P\ (Real\ X)$
  shows $P\ x$
⟨*proof*⟩

lemma *eq-Real*: $cauchy\ X \Longrightarrow cauchy\ Y \Longrightarrow Real\ X = Real\ Y \longleftrightarrow vanishes\ (\lambda n.\ X\ n - Y\ n)$
  ⟨*proof*⟩

lemma *Domainp-pcr-real* [*transfer-domain-rule*]: $Domainp\ pcr\text{-}real = cauchy$
  ⟨*proof*⟩

instantiation *real* :: *field*
begin

lift-definition *zero-real* :: *real* **is** $\lambda n.\ 0$
⟨*proof*⟩

lift-definition *one-real* :: *real* **is** $\lambda n.\ 1$
⟨*proof*⟩

lift-definition *plus-real* :: *real* ⇒ *real* ⇒ *real* **is** $\lambda X\ Y\ n.\ X\ n + Y\ n$
⟨*proof*⟩

lift-definition *uminus-real* :: *real* ⇒ *real* **is** $\lambda X\ n.\ -\ X\ n$
⟨*proof*⟩

lift-definition *times-real* :: *real* ⇒ *real* ⇒ *real* **is** $\lambda X\ Y\ n.\ X\ n * Y\ n$
⟨*proof*⟩

lift-definition *inverse-real* :: *real* ⇒ *real*
is $\lambda X.$ *if vanishes X then* $(\lambda n.\ 0)$ *else* $(\lambda n.$ *inverse* $(X\ n))$
⟨*proof*⟩

definition $x - y = x + - y$ **for** *x y* :: *real*

definition *x div y* = *x* ∗ *inverse y* **for** *x y* :: *real*

lemma *add-Real*: *cauchy X* ⟹ *cauchy Y* ⟹ *Real X* + *Real Y* = *Real* $(\lambda n.\ X\ n + Y\ n)$
⟨*proof*⟩

lemma *minus-Real*: *cauchy X* ⟹ − *Real X* = *Real* $(\lambda n.\ -\ X\ n)$
⟨*proof*⟩

lemma *diff-Real*: *cauchy X* ⟹ *cauchy Y* ⟹ *Real X* − *Real Y* = *Real* $(\lambda n.\ X\ n - Y\ n)$
⟨*proof*⟩

lemma *mult-Real*: *cauchy X* ⟹ *cauchy Y* ⟹ *Real X* ∗ *Real Y* = *Real* $(\lambda n.\ X\ n * Y\ n)$
⟨*proof*⟩

lemma *inverse-Real*:
cauchy X ⟹ *inverse* (*Real X*) = (*if vanishes X then 0 else Real* $(\lambda n.$ *inverse* $(X\ n)))$
⟨*proof*⟩

instance
⟨*proof*⟩

end

96.6 Positive reals

lift-definition *positive* :: *real* $\Rightarrow$ *bool*
is $\lambda X.\ \exists\, r>0.\ \exists\, k.\ \forall\, n\geq k.\ r < X\ n$
⟨*proof*⟩

lemma *positive-Real*: $cauchy\ X \Longrightarrow positive\ (Real\ X) \longleftrightarrow (\exists\, r>0.\ \exists\, k.\ \forall\, n\geq k.\ r < X\ n)$
⟨*proof*⟩

lemma *positive-zero*: $\neg\ positive\ 0$
⟨*proof*⟩

lemma *positive-add*: $positive\ x \Longrightarrow positive\ y \Longrightarrow positive\ (x + y)$
⟨*proof*⟩

lemma *positive-mult*: $positive\ x \Longrightarrow positive\ y \Longrightarrow positive\ (x * y)$
⟨*proof*⟩

lemma *positive-minus*: $\neg\ positive\ x \Longrightarrow x \neq 0 \Longrightarrow positive\ (-\ x)$
⟨*proof*⟩

instantiation *real* :: *linordered-field*
begin

definition $x < y \longleftrightarrow positive\ (y - x)$

definition $x \leq y \longleftrightarrow x < y \lor x = y$ **for** $x\ y$ :: *real*

definition $|a| = (if\ a < 0\ then\ -\ a\ else\ a)$ **for** a :: *real*

definition $sgn\ a = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ 1\ else\ -\ 1)$ **for** a :: *real*

instance
⟨*proof*⟩

end

instantiation *real* :: *distrib-lattice*
begin

definition $(inf :: real \Rightarrow real \Rightarrow real) = min$

definition $(sup :: real \Rightarrow real \Rightarrow real) = max$

instance
⟨*proof*⟩

end

lemma *of-nat-Real*: *of-nat* x = *Real* (λn. *of-nat* x)
⟨*proof*⟩

lemma *of-int-Real*: *of-int* x = *Real* (λn. *of-int* x)
⟨*proof*⟩

lemma *of-rat-Real*: *of-rat* x = *Real* (λn. x)
⟨*proof*⟩

instance *real* :: *archimedean-field*
⟨*proof*⟩

instantiation *real* :: *floor-ceiling*
begin

definition [*code del*]: $\lfloor x$::*real*$\rfloor$ = (*THE* z. *of-int* $z \le x \wedge x <$ *of-int* (z + *1*))

instance
⟨*proof*⟩

end

96.7 Completeness

lemma *not-positive-Real*: $\neg$ *positive* (*Real* X) $\longleftrightarrow$ ($\forall r>0$. $\exists k$. $\forall n \ge k$. $X\ n \le r$)
if *cauchy* X
⟨*proof*⟩

lemma *le-Real*:
assumes *cauchy* X *cauchy* Y
shows *Real* $X \le$ *Real* Y = ($\forall r>0$. $\exists k$. $\forall n \ge k$. $X\ n \le Y\ n + r$)
⟨*proof*⟩

lemma *le-RealI*:
assumes Y: *cauchy* Y
shows $\forall n$. $x \le$ *of-rat* ($Y\ n$) $\Longrightarrow x \le$ *Real* Y
⟨*proof*⟩

lemma *Real-leI*:
assumes X: *cauchy* X
assumes *le*: $\forall n$. *of-rat* ($X\ n$) $\le y$
shows *Real* $X \le y$
⟨*proof*⟩

lemma *less-RealD*:
assumes *cauchy* Y

shows $x < \mathit{Real}\ Y \Longrightarrow \exists\, n.\ x < \mathit{of\text{-}rat}\ (Y\ n)$
⟨*proof*⟩

lemma *of-nat-less-two-power* [*simp*]: $\mathit{of\text{-}nat}\ n < (2{::}'a{::}\mathit{linordered\text{-}idom})\ \hat{}\ n$
⟨*proof*⟩

lemma *complete-real*:
fixes $S :: \mathit{real\ set}$
assumes $\exists\, x.\ x \in S$ **and** $\exists\, z.\ \forall\, x{\in}S.\ x \le z$
shows $\exists\, y.\ (\forall\, x{\in}S.\ x \le y) \wedge (\forall\, z.\ (\forall\, x{\in}S.\ x \le z) \longrightarrow y \le z)$
⟨*proof*⟩

instantiation *real* :: *linear-continuum*
begin

96.8 Supremum of a set of reals

definition $\mathit{Sup}\ X = (\mathit{LEAST}\ z{::}\mathit{real}.\ \forall\, x{\in}X.\ x \le z)$
definition $\mathit{Inf}\ X = -\ \mathit{Sup}\ (\mathit{uminus}\ \text{`}\ X)$ **for** $X :: \mathit{real\ set}$

instance
⟨*proof*⟩

end

96.9 Hiding implementation details

hide-const (**open**) *vanishes cauchy positive Real*

declare *Real-induct* [*induct del*]
declare *Abs-real-induct* [*induct del*]
declare *Abs-real-cases* [*cases del*]

lifting-update *real.lifting*
lifting-forget *real.lifting*

96.10 More Lemmas

BH: These lemmas should not be necessary; they should be covered by existing simp rules and simplification procedures.

lemma *real-mult-less-iff1* [*simp*]: $0 < z \Longrightarrow x * z < y * z \longleftrightarrow x < y$
for $x\ y\ z :: \mathit{real}$
⟨*proof*⟩

lemma *real-mult-le-cancel-iff1* [*simp*]: $0 < z \Longrightarrow x * z \le y * z \longleftrightarrow x \le y$
for $x\ y\ z :: \mathit{real}$
⟨*proof*⟩

lemma *real-mult-le-cancel-iff2* [*simp*]: $0 < z \Longrightarrow z * x \le z * y \longleftrightarrow x \le y$

for $x\ y\ z$ :: *real*
⟨*proof*⟩

96.11 Embedding numbers into the Reals

abbreviation *real-of-nat* :: *nat* ⇒ *real*
where *real-of-nat* ≡ *of-nat*

abbreviation *real* :: *nat* ⇒ *real*
where *real* ≡ *of-nat*

abbreviation *real-of-int* :: *int* ⇒ *real*
where *real-of-int* ≡ *of-int*

abbreviation *real-of-rat* :: *rat* ⇒ *real*
where *real-of-rat* ≡ *of-rat*

declare [[*coercion-enabled*]]

declare [[*coercion of-nat* :: *nat* ⇒ *int*]]
declare [[*coercion of-nat* :: *nat* ⇒ *real*]]
declare [[*coercion of-int* :: *int* ⇒ *real*]]

declare [[*coercion-map map*]]
declare [[*coercion-map* $\lambda f\ g\ h\ x.\ g\ (h\ (f\ x))$]]
declare [[*coercion-map* $\lambda f\ g\ (x,y).\ (f\ x,\ g\ y)$]]

declare *of-int-eq-0-iff* [*algebra*, *presburger*]
declare *of-int-eq-1-iff* [*algebra*, *presburger*]
declare *of-int-eq-iff* [*algebra*, *presburger*]
declare *of-int-less-0-iff* [*algebra*, *presburger*]
declare *of-int-less-1-iff* [*algebra*, *presburger*]
declare *of-int-less-iff* [*algebra*, *presburger*]
declare *of-int-le-0-iff* [*algebra*, *presburger*]
declare *of-int-le-1-iff* [*algebra*, *presburger*]
declare *of-int-le-iff* [*algebra*, *presburger*]
declare *of-int-0-less-iff* [*algebra*, *presburger*]
declare *of-int-0-le-iff* [*algebra*, *presburger*]
declare *of-int-1-less-iff* [*algebra*, *presburger*]
declare *of-int-1-le-iff* [*algebra*, *presburger*]

lemma *int-less-real-le*: $n < m \longleftrightarrow$ *real-of-int* $n + 1 \le$ *real-of-int* m
⟨*proof*⟩

lemma *int-le-real-less*: $n \le m \longleftrightarrow$ *real-of-int* $n <$ *real-of-int* $m + 1$
⟨*proof*⟩

lemma *real-of-int-div-aux*:
$(\mathit{real\text{-}of\text{-}int}\ x)\ /\ (\mathit{real\text{-}of\text{-}int}\ d) =$
$\mathit{real\text{-}of\text{-}int}\ (x\ \mathit{div}\ d) + (\mathit{real\text{-}of\text{-}int}\ (x\ \mathit{mod}\ d))\ /\ (\mathit{real\text{-}of\text{-}int}\ d)$
⟨*proof*⟩

lemma *real-of-int-div*:
$d\ \mathit{dvd}\ n \Longrightarrow \mathit{real\text{-}of\text{-}int}\ (n\ \mathit{div}\ d) = \mathit{real\text{-}of\text{-}int}\ n\ /\ \mathit{real\text{-}of\text{-}int}\ d$ **for** $d\ n :: \mathit{int}$
⟨*proof*⟩

lemma *real-of-int-div2*: $0 \le \mathit{real\text{-}of\text{-}int}\ n\ /\ \mathit{real\text{-}of\text{-}int}\ x - \mathit{real\text{-}of\text{-}int}\ (n\ \mathit{div}\ x)$
⟨*proof*⟩

lemma *real-of-int-div3*: $\mathit{real\text{-}of\text{-}int}\ n\ /\ \mathit{real\text{-}of\text{-}int}\ x - \mathit{real\text{-}of\text{-}int}\ (n\ \mathit{div}\ x) \le 1$
⟨*proof*⟩

lemma *real-of-int-div4*: $\mathit{real\text{-}of\text{-}int}\ (n\ \mathit{div}\ x) \le \mathit{real\text{-}of\text{-}int}\ n\ /\ \mathit{real\text{-}of\text{-}int}\ x$
⟨*proof*⟩

96.12 Embedding the Naturals into the Reals

lemma *real-of-card*: $\mathit{real}\ (\mathit{card}\ A) = \mathit{sum}\ (\lambda x.\ 1)\ A$
⟨*proof*⟩

lemma *nat-less-real-le*: $n < m \longleftrightarrow \mathit{real}\ n + 1 \le \mathit{real}\ m$
⟨*proof*⟩

lemma *nat-le-real-less*: $n \le m \longleftrightarrow \mathit{real}\ n < \mathit{real}\ m + 1$
for $m\ n :: \mathit{nat}$
⟨*proof*⟩

lemma *real-of-nat-div-aux*: $\mathit{real}\ x\ /\ \mathit{real}\ d = \mathit{real}\ (x\ \mathit{div}\ d) + \mathit{real}\ (x\ \mathit{mod}\ d)\ /\ \mathit{real}\ d$
⟨*proof*⟩

lemma *real-of-nat-div*: $d\ \mathit{dvd}\ n \Longrightarrow \mathit{real}(n\ \mathit{div}\ d) = \mathit{real}\ n\ /\ \mathit{real}\ d$
⟨*proof*⟩

lemma *real-of-nat-div2*: $0 \le \mathit{real}\ n\ /\ \mathit{real}\ x - \mathit{real}\ (n\ \mathit{div}\ x)$ **for** $n\ x :: \mathit{nat}$
⟨*proof*⟩

lemma *real-of-nat-div3*: $\mathit{real}\ n\ /\ \mathit{real}\ x - \mathit{real}\ (n\ \mathit{div}\ x) \le 1$ **for** $n\ x :: \mathit{nat}$
⟨*proof*⟩

lemma *real-of-nat-div4*: $\mathit{real}\ (n\ \mathit{div}\ x) \le \mathit{real}\ n\ /\ \mathit{real}\ x$ **for** $n\ x :: \mathit{nat}$
⟨*proof*⟩

96.13 The Archimedean Property of the Reals

lemma *real-arch-inverse*: $0 < e \longleftrightarrow (\exists\, n{::}\mathit{nat}.\ n \ne 0 \wedge 0 < \mathit{inverse}\ (\mathit{real}\ n) \wedge \mathit{inverse}\ (\mathit{real}\ n) < e)$

⟨*proof*⟩

lemma *reals-Archimedean3*: $0 < x \Longrightarrow \forall y.\ \exists n.\ y < real\ n * x$
⟨*proof*⟩

lemma *real-archimedian-rdiv-eq-0*:
assumes *x0*: $x \geq 0$
and *c*: $c \geq 0$
and *xc*: $\bigwedge m{::}nat.\ m > 0 \Longrightarrow real\ m * x \leq c$
shows $x = 0$
⟨*proof*⟩

96.14 Rationals

lemma *Rats-eq-int-div-int*: $\mathbb{Q} = \{real\text{-}of\text{-}int\ i\ /\ real\text{-}of\text{-}int\ j \mid i\ j.\ j \neq 0\}$ (**is** - = *?S*)
⟨*proof*⟩

lemma *Rats-eq-int-div-nat*: $\mathbb{Q} = \{\ real\text{-}of\text{-}int\ i\ /\ real\ n \mid i\ n.\ n \neq 0\}$
⟨*proof*⟩

lemma *Rats-abs-nat-div-natE*:
assumes $x \in \mathbb{Q}$
obtains $m\ n :: nat$ **where** $n \neq 0$ **and** $|x| = real\ m\ /\ real\ n$ **and** $gcd\ m\ n = 1$
⟨*proof*⟩

96.15 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is *Real Analysis* by H.L. Royden. It employs the Archimedean property of the reals.

lemma *Rats-dense-in-real*:
fixes $x :: real$
assumes $x < y$
shows $\exists r{\in}\mathbb{Q}.\ x < r \wedge r < y$
⟨*proof*⟩

lemma *of-rat-dense*:
fixes $x\ y :: real$
assumes $x < y$
shows $\exists q :: rat.\ x < of\text{-}rat\ q \wedge of\text{-}rat\ q < y$
⟨*proof*⟩

96.16 Numerals and Arithmetic

⟨*ML*⟩

96.17 Simprules combining $x + y$ and 0

lemma *real-add-minus-iff* [*simp*]: $x + - a = 0 \longleftrightarrow x = a$
  for $x\ a$ :: *real*
  ⟨*proof*⟩

lemma *real-add-less-0-iff*: $x + y < 0 \longleftrightarrow y < - x$
  for $x\ y$ :: *real*
  ⟨*proof*⟩

lemma *real-0-less-add-iff*: $0 < x + y \longleftrightarrow - x < y$
  for $x\ y$ :: *real*
  ⟨*proof*⟩

lemma *real-add-le-0-iff*: $x + y \leq 0 \longleftrightarrow y \leq - x$
  for $x\ y$ :: *real*
  ⟨*proof*⟩

lemma *real-0-le-add-iff*: $0 \leq x + y \longleftrightarrow - x \leq y$
  for $x\ y$ :: *real*
  ⟨*proof*⟩

96.18 Lemmas about powers

lemma *two-realpow-ge-one*: $(1::real) \leq 2\ \hat{}\ n$
  ⟨*proof*⟩

declare *sum-squares-eq-zero-iff* [*simp*] *sum-power2-eq-zero-iff* [*simp*]

lemma *real-minus-mult-self-le* [*simp*]: $- (u * u) \leq x * x$
  for $u\ x$ :: *real*
  ⟨*proof*⟩

lemma *realpow-square-minus-le* [*simp*]: $- u^2 \leq x^2$
  for $u\ x$ :: *real*
  ⟨*proof*⟩

lemma *numeral-power-eq-real-of-int-cancel-iff* [*simp*]:
  numeral $x\ \hat{}\ n =$ *real-of-int* $y \longleftrightarrow$ *numeral* $x\ \hat{}\ n = y$
  ⟨*proof*⟩

lemma *real-of-int-eq-numeral-power-cancel-iff* [*simp*]:
  real-of-int $y =$ *numeral* $x\ \hat{}\ n \longleftrightarrow y =$ *numeral* $x\ \hat{}\ n$
  ⟨*proof*⟩

lemma *numeral-power-eq-real-of-nat-cancel-iff* [*simp*]:
  numeral $x\ \hat{}\ n =$ *real* $y \longleftrightarrow$ *numeral* $x\ \hat{}\ n = y$
  ⟨*proof*⟩

lemma *real-of-nat-eq-numeral-power-cancel-iff* [*simp*]:
$real\ y = numeral\ x\ \hat{}\ n \longleftrightarrow y = numeral\ x\ \hat{}\ n$
⟨*proof*⟩

lemma *numeral-power-le-real-of-nat-cancel-iff* [*simp*]:
$(numeral\ x :: real)\ \hat{}\ n \leq real\ a \longleftrightarrow (numeral\ x::nat)\ \hat{}\ n \leq a$
⟨*proof*⟩

lemma *real-of-nat-le-numeral-power-cancel-iff* [*simp*]:
$real\ a \leq (numeral\ x::real)\ \hat{}\ n \longleftrightarrow a \leq (numeral\ x::nat)\ \hat{}\ n$
⟨*proof*⟩

lemma *numeral-power-le-real-of-int-cancel-iff* [*simp*]:
$(numeral\ x::real)\ \hat{}\ n \leq real\text{-}of\text{-}int\ a \longleftrightarrow (numeral\ x::int)\ \hat{}\ n \leq a$
⟨*proof*⟩

lemma *real-of-int-le-numeral-power-cancel-iff* [*simp*]:
$real\text{-}of\text{-}int\ a \leq (numeral\ x::real)\ \hat{}\ n \longleftrightarrow a \leq (numeral\ x::int)\ \hat{}\ n$
⟨*proof*⟩

lemma *numeral-power-less-real-of-nat-cancel-iff* [*simp*]:
$(numeral\ x::real)\ \hat{}\ n < real\ a \longleftrightarrow (numeral\ x::nat)\ \hat{}\ n < a$
⟨*proof*⟩

lemma *real-of-nat-less-numeral-power-cancel-iff* [*simp*]:
$real\ a < (numeral\ x::real)\ \hat{}\ n \longleftrightarrow a < (numeral\ x::nat)\ \hat{}\ n$
⟨*proof*⟩

lemma *numeral-power-less-real-of-int-cancel-iff* [*simp*]:
$(numeral\ x::real)\ \hat{}\ n < real\text{-}of\text{-}int\ a \longleftrightarrow (numeral\ x::int)\ \hat{}\ n < a$
⟨*proof*⟩

lemma *real-of-int-less-numeral-power-cancel-iff* [*simp*]:
$real\text{-}of\text{-}int\ a < (numeral\ x::real)\ \hat{}\ n \longleftrightarrow a < (numeral\ x::int)\ \hat{}\ n$
⟨*proof*⟩

lemma *neg-numeral-power-le-real-of-int-cancel-iff* [*simp*]:
$(-\ numeral\ x::real)\ \hat{}\ n \leq real\text{-}of\text{-}int\ a \longleftrightarrow (-\ numeral\ x::int)\ \hat{}\ n \leq a$
⟨*proof*⟩

lemma *real-of-int-le-neg-numeral-power-cancel-iff* [*simp*]:
$real\text{-}of\text{-}int\ a \leq (-\ numeral\ x::real)\ \hat{}\ n \longleftrightarrow a \leq (-\ numeral\ x::int)\ \hat{}\ n$
⟨*proof*⟩

96.19 Density of the Reals

lemma *real-lbound-gt-zero*: $0 < d1 \Longrightarrow 0 < d2 \Longrightarrow \exists e.\ 0 < e \wedge e < d1 \wedge e < d2$
for $d1\ d2 :: real$

⟨*proof*⟩

Similar results are proved in *Fields*

lemma *real-less-half-sum*: $x < y \Longrightarrow x < (x + y) / 2$
for $x\ y$ :: *real*
⟨*proof*⟩

lemma *real-gt-half-sum*: $x < y \Longrightarrow (x + y) / 2 < y$
for $x\ y$ :: *real*
⟨*proof*⟩

lemma *real-sum-of-halves*: $x / 2 + x / 2 = x$
for x :: *real*
⟨*proof*⟩

96.20 Floor and Ceiling Functions from the Reals to the Integers

lemma *real-of-nat-less-numeral-iff* [*simp*]: *real* $n <$ *numeral* $w \longleftrightarrow n <$ *numeral* w
for n :: *nat*
⟨*proof*⟩

lemma *numeral-less-real-of-nat-iff* [*simp*]: *numeral* $w <$ *real* $n \longleftrightarrow$ *numeral* $w < n$
for n :: *nat*
⟨*proof*⟩

lemma *numeral-le-real-of-nat-iff* [*simp*]: *numeral* $n \leq$ *real* $m \longleftrightarrow$ *numeral* $n \leq m$
for m :: *nat*
⟨*proof*⟩

declare *of-int-floor-le* [*simp*]

lemma *of-int-floor-cancel* [*simp*]: *of-int* $\lfloor x \rfloor = x \longleftrightarrow (\exists\, n{::}int.\ x =$ *of-int* $n)$
⟨*proof*⟩

lemma *floor-eq*: *real-of-int* $n < x \Longrightarrow x <$ *real-of-int* $n + 1 \Longrightarrow \lfloor x \rfloor = n$
⟨*proof*⟩

lemma *floor-eq2*: *real-of-int* $n \leq x \Longrightarrow x <$ *real-of-int* $n + 1 \Longrightarrow \lfloor x \rfloor = n$
⟨*proof*⟩

lemma *floor-eq3*: *real* $n < x \Longrightarrow x <$ *real* $(Suc\ n) \Longrightarrow$ *nat* $\lfloor x \rfloor = n$
⟨*proof*⟩

lemma *floor-eq4*: *real* $n \leq x \Longrightarrow x <$ *real* $(Suc\ n) \Longrightarrow$ *nat* $\lfloor x \rfloor = n$
⟨*proof*⟩

lemma *real-of-int-floor-ge-diff-one* [*simp*]: $r - 1 \leq$ *real-of-int* $\lfloor r \rfloor$
⟨*proof*⟩

lemma *real-of-int-floor-gt-diff-one* [*simp*]: $r - 1 <$ *real-of-int* $\lfloor r \rfloor$
⟨*proof*⟩

lemma *real-of-int-floor-add-one-ge* [*simp*]: $r \leq$ *real-of-int* $\lfloor r \rfloor + 1$
⟨*proof*⟩

lemma *real-of-int-floor-add-one-gt* [*simp*]: $r <$ *real-of-int* $\lfloor r \rfloor + 1$
⟨*proof*⟩

lemma *floor-divide-real-eq-div*:
assumes $0 \leq b$
shows $\lfloor a$ / *real-of-int* $b \rfloor = \lfloor a \rfloor$ *div* b
⟨*proof*⟩

lemma *floor-one-divide-eq-div-numeral* [*simp*]:
$\lfloor 1$ / *numeral* b::*real*$\rfloor = 1$ *div numeral* b
⟨*proof*⟩

lemma *floor-minus-one-divide-eq-div-numeral* [*simp*]:
$\lfloor -$ (1 / *numeral* b)::*real*$\rfloor = - 1$ *div numeral* b
⟨*proof*⟩

lemma *floor-divide-eq-div-numeral* [*simp*]:
$\lfloor$*numeral* a / *numeral* b::*real*$\rfloor$ = *numeral* a *div numeral* b
⟨*proof*⟩

lemma *floor-minus-divide-eq-div-numeral* [*simp*]:
$\lfloor -$ (*numeral* a / *numeral* b)::*real*$\rfloor = -$ *numeral* a *div numeral* b
⟨*proof*⟩

lemma *of-int-ceiling-cancel* [*simp*]: *of-int* $\lceil x \rceil = x \longleftrightarrow (\exists\, n$::*int*. $x =$ *of-int* n)
⟨*proof*⟩

lemma *ceiling-eq*: *of-int* $n < x \Longrightarrow x \leq$ *of-int* $n + 1 \Longrightarrow \lceil x \rceil = n + 1$
⟨*proof*⟩

lemma *of-int-ceiling-diff-one-le* [*simp*]: *of-int* $\lceil r \rceil - 1 \leq r$
⟨*proof*⟩

lemma *of-int-ceiling-le-add-one* [*simp*]: *of-int* $\lceil r \rceil \leq r + 1$
⟨*proof*⟩

lemma *ceiling-le*: $x \leq$ *of-int* $a \Longrightarrow \lceil x \rceil \leq a$
⟨*proof*⟩

lemma *ceiling-divide-eq-div*: $\lceil$*of-int* a / *of-int* $b\rceil = - (- a$ *div* $b)$

⟨*proof*⟩

lemma *ceiling-divide-eq-div-numeral* [*simp*]:
$\lceil numeral\ a\ /\ numeral\ b :: real\rceil = -\ (-\ numeral\ a\ div\ numeral\ b)$
⟨*proof*⟩

lemma *ceiling-minus-divide-eq-div-numeral* [*simp*]:
$\lceil -\ (numeral\ a\ /\ numeral\ b :: real)\rceil = -\ (numeral\ a\ div\ numeral\ b)$
⟨*proof*⟩

The following lemmas are remnants of the erstwhile functions natfloor and natceiling.

lemma *nat-floor-neg*: $x \leq 0 \Longrightarrow nat\ \lfloor x\rfloor = 0$
for $x :: real$
⟨*proof*⟩

lemma *le-nat-floor*: $real\ x \leq a \Longrightarrow x \leq nat\ \lfloor a\rfloor$
⟨*proof*⟩

lemma *le-mult-nat-floor*: $nat\ \lfloor a\rfloor * nat\ \lfloor b\rfloor \leq nat\ \lfloor a * b\rfloor$
⟨*proof*⟩

lemma *nat-ceiling-le-eq* [*simp*]: $nat\ \lceil x\rceil \leq a \longleftrightarrow x \leq real\ a$
⟨*proof*⟩

lemma *real-nat-ceiling-ge*: $x \leq real\ (nat\ \lceil x\rceil)$
⟨*proof*⟩

lemma *Rats-no-top-le*: $\exists\, q \in \mathbb{Q}.\ x \leq q$
for $x :: real$
⟨*proof*⟩

lemma *Rats-no-bot-less*: $\exists\, q \in \mathbb{Q}.\ q < x$ **for** $x :: real$
⟨*proof*⟩

96.21 Exponentiation with floor

lemma *floor-power*:
assumes $x = of\text{-}int\ \lfloor x\rfloor$
shows $\lfloor x\ \hat{}\ n\rfloor = \lfloor x\rfloor\ \hat{}\ n$
⟨*proof*⟩

lemma *floor-numeral-power* [*simp*]: $\lfloor numeral\ x\ \hat{}\ n\rfloor = numeral\ x\ \hat{}\ n$
⟨*proof*⟩

lemma *ceiling-numeral-power* [*simp*]: $\lceil numeral\ x\ \hat{}\ n\rceil = numeral\ x\ \hat{}\ n$
⟨*proof*⟩

96.22 Implementation of rational real numbers

Formal constructor

definition *Ratreal* :: *rat* ⇒ *real*
  where [*code-abbrev*, *simp*]: *Ratreal* = *real-of-rat*

code-datatype *Ratreal*

Quasi-Numerals

lemma [*code-abbrev*]:
  real-of-rat (*numeral k*) = *numeral k*
  real-of-rat (− *numeral k*) = − *numeral k*
  real-of-rat (*rat-of-int a*) = *real-of-int a*
  ⟨*proof*⟩

lemma [*code-post*]:
  real-of-rat 0 = *0*
  real-of-rat 1 = *1*
  real-of-rat (− *1*) = − *1*
  real-of-rat (*1* / *numeral k*) = *1* / *numeral k*
  real-of-rat (*numeral k* / *numeral l*) = *numeral k* / *numeral l*
  real-of-rat (− (*1* / *numeral k*)) = − (*1* / *numeral k*)
  real-of-rat (− (*numeral k* / *numeral l*)) = − (*numeral k* / *numeral l*)
  ⟨*proof*⟩

Operations

lemma *zero-real-code* [*code*]: *0* = *Ratreal 0*
  ⟨*proof*⟩

lemma *one-real-code* [*code*]: *1* = *Ratreal 1*
  ⟨*proof*⟩

instantiation *real* :: *equal*
begin

definition *HOL.equal x y* ⟷ *x* − *y* = *0* **for** *x* :: *real*

instance ⟨*proof*⟩

lemma *real-equal-code* [*code*]: *HOL.equal* (*Ratreal x*) (*Ratreal y*) ⟷ *HOL.equal x y*
  ⟨*proof*⟩

lemma [*code nbe*]: *HOL.equal x x* ⟷ *True*
  for *x* :: *real*
  ⟨*proof*⟩

end

lemma *real-less-eq-code* [*code*]: $Ratreal\ x \leq Ratreal\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *real-less-code* [*code*]: $Ratreal\ x < Ratreal\ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *real-plus-code* [*code*]: $Ratreal\ x + Ratreal\ y = Ratreal\ (x + y)$
⟨*proof*⟩

lemma *real-times-code* [*code*]: $Ratreal\ x * Ratreal\ y = Ratreal\ (x * y)$
⟨*proof*⟩

lemma *real-uminus-code* [*code*]: $-\ Ratreal\ x = Ratreal\ (-\ x)$
⟨*proof*⟩

lemma *real-minus-code* [*code*]: $Ratreal\ x - Ratreal\ y = Ratreal\ (x - y)$
⟨*proof*⟩

lemma *real-inverse-code* [*code*]: $inverse\ (Ratreal\ x) = Ratreal\ (inverse\ x)$
⟨*proof*⟩

lemma *real-divide-code* [*code*]: $Ratreal\ x\ /\ Ratreal\ y = Ratreal\ (x\ /\ y)$
⟨*proof*⟩

lemma *real-floor-code* [*code*]: $\lfloor Ratreal\ x \rfloor = \lfloor x \rfloor$
⟨*proof*⟩

Quickcheck

definition (**in** *term-syntax*)
valterm-ratreal :: $rat \times (unit \Rightarrow Code\text{-}Evaluation.term) \Rightarrow real \times (unit \Rightarrow Code\text{-}Evaluation.term)$
where [*code-unfold*]: $valterm\text{-}ratreal\ k = Code\text{-}Evaluation.valtermify\ Ratreal\ \{\cdot\}\ k$

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

instantiation *real* :: *random*
begin

definition
$Quickcheck\text{-}Random.random\ i = Quickcheck\text{-}Random.random\ i \circ\!\!\rightarrow (\lambda r.\ Pair\ (valterm\text{-}ratreal\ r))$

instance ⟨*proof*⟩

end

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

no-notation *scomp* (**infixl** $\circ\rightarrow$ *60*)

instantiation *real* :: *exhaustive*
begin

definition
exhaustive-real f d = *Quickcheck-Exhaustive.exhaustive* (λr. *f* (*Ratreal r*)) *d*

instance ⟨*proof*⟩

end

instantiation *real* :: *full-exhaustive*
begin

definition
full-exhaustive-real f d = *Quickcheck-Exhaustive.full-exhaustive* (λr. *f* (*valterm-ratreal r*)) *d*

instance ⟨*proof*⟩

end

instantiation *real* :: *narrowing*
begin

definition
narrowing-real = *Quickcheck-Narrowing.apply* (*Quickcheck-Narrowing.cons Ratreal*) *narrowing*

instance ⟨*proof*⟩

end

96.23 Setup for Nitpick

⟨*ML*⟩

lemmas [*nitpick-unfold*] = *inverse-real-inst.inverse-real one-real-inst.one-real*
ord-real-inst.less-real ord-real-inst.less-eq-real plus-real-inst.plus-real
times-real-inst.times-real uminus-real-inst.uminus-real
zero-real-inst.zero-real

96.24 Setup for SMT

⟨*ML*⟩

lemma [*z3-rule*]:
$0 + x = x$
$x + 0 = x$

0 $*$ *x* = *0*
1 $*$ *x* = *x*
$-x = -1 * x$
$x + y = y + x$
for *x y* :: *real*
⟨*proof*⟩

96.25 Setup for Argo

⟨*ML*⟩

end

97 Topological Spaces

theory *Topological-Spaces*
imports *Main*
begin

named-theorems *continuous-intros structural introduction rules for continuity*

97.1 Topological space

class *open* =
fixes *open* :: *'a set* ⇒ *bool*

class *topological-space* = *open* +
assumes *open-UNIV* [*simp*, *intro*]: *open UNIV*
assumes *open-Int* [*intro*]: *open S* ⟹ *open T* ⟹ *open* (*S* ∩ *T*)
assumes *open-Union* [*intro*]: ∀ *S*∈*K*. *open S* ⟹ *open* (⋃ *K*)
begin

definition *closed* :: *'a set* ⇒ *bool*
where *closed S* ⟷ *open* (− *S*)

lemma *open-empty* [*continuous-intros*, *intro*, *simp*]: *open* {}
⟨*proof*⟩

lemma *open-Un* [*continuous-intros*, *intro*]: *open S* ⟹ *open T* ⟹ *open* (*S* ∪ *T*)
⟨*proof*⟩

lemma *open-UN* [*continuous-intros*, *intro*]: ∀ *x*∈*A*. *open* (*B x*) ⟹ *open* (⋃ *x*∈*A*. *B x*)
⟨*proof*⟩

lemma *open-Inter* [*continuous-intros*, *intro*]: *finite S* ⟹ ∀ *T*∈*S*. *open T* ⟹ *open* (⋂ *S*)
⟨*proof*⟩

lemma *open-INT* [*continuous-intros*, *intro*]: *finite* $A \Longrightarrow \forall x{\in}A.$ *open* $(B\ x) \Longrightarrow$ *open* $(\bigcap x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *openI*:
assumes $\bigwedge x.\ x \in S \Longrightarrow \exists\, T.$ *open* $T \wedge x \in T \wedge T \subseteq S$
shows *open* S
⟨*proof*⟩

lemma *closed-empty* [*continuous-intros*, *intro*, *simp*]: *closed* $\{\}$
⟨*proof*⟩

lemma *closed-Un* [*continuous-intros*, *intro*]: *closed* $S \Longrightarrow$ *closed* $T \Longrightarrow$ *closed* $(S \cup T)$
⟨*proof*⟩

lemma *closed-UNIV* [*continuous-intros*, *intro*, *simp*]: *closed UNIV*
⟨*proof*⟩

lemma *closed-Int* [*continuous-intros*, *intro*]: *closed* $S \Longrightarrow$ *closed* $T \Longrightarrow$ *closed* $(S \cap T)$
⟨*proof*⟩

lemma *closed-INT* [*continuous-intros*, *intro*]: $\forall x{\in}A.$ *closed* $(B\ x) \Longrightarrow$ *closed* $(\bigcap x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *closed-Inter* [*continuous-intros*, *intro*]: $\forall S{\in}K.$ *closed* $S \Longrightarrow$ *closed* $(\bigcap K)$
⟨*proof*⟩

lemma *closed-Union* [*continuous-intros*, *intro*]: *finite* $S \Longrightarrow \forall\, T{\in}S.$ *closed* $T \Longrightarrow$ *closed* $(\bigcup S)$
⟨*proof*⟩

lemma *closed-UN* [*continuous-intros*, *intro*]:
finite $A \Longrightarrow \forall x{\in}A.$ *closed* $(B\ x) \Longrightarrow$ *closed* $(\bigcup x{\in}A.\ B\ x)$
⟨*proof*⟩

lemma *open-closed*: *open* $S \longleftrightarrow$ *closed* $(-\ S)$
⟨*proof*⟩

lemma *closed-open*: *closed* $S \longleftrightarrow$ *open* $(-\ S)$
⟨*proof*⟩

lemma *open-Diff* [*continuous-intros*, *intro*]: *open* $S \Longrightarrow$ *closed* $T \Longrightarrow$ *open* $(S - T)$
⟨*proof*⟩

lemma *closed-Diff* [*continuous-intros*, *intro*]: *closed* $S \Longrightarrow$ *open* $T \Longrightarrow$ *closed* $(S$

$-\ T)$
⟨proof⟩

lemma *open-Compl* [*continuous-intros*, *intro*]: *closed* $S \Longrightarrow$ *open* $(-\ S)$
⟨proof⟩

lemma *closed-Compl* [*continuous-intros*, *intro*]: *open* $S \Longrightarrow$ *closed* $(-\ S)$
⟨proof⟩

lemma *open-Collect-neg*: *closed* $\{x.\ P\ x\} \Longrightarrow$ *open* $\{x.\ \neg\ P\ x\}$
⟨proof⟩

lemma *open-Collect-conj*:
assumes *open* $\{x.\ P\ x\}$ *open* $\{x.\ Q\ x\}$
shows *open* $\{x.\ P\ x \wedge Q\ x\}$
⟨proof⟩

lemma *open-Collect-disj*:
assumes *open* $\{x.\ P\ x\}$ *open* $\{x.\ Q\ x\}$
shows *open* $\{x.\ P\ x \vee Q\ x\}$
⟨proof⟩

lemma *open-Collect-ex*: $(\bigwedge i.$ *open* $\{x.\ P\ i\ x\}) \Longrightarrow$ *open* $\{x.\ \exists\, i.\ P\ i\ x\}$
⟨proof⟩

lemma *open-Collect-imp*: *closed* $\{x.\ P\ x\} \Longrightarrow$ *open* $\{x.\ Q\ x\} \Longrightarrow$ *open* $\{x.\ P\ x \longrightarrow Q\ x\}$
⟨proof⟩

lemma *open-Collect-const*: *open* $\{x.\ P\}$
⟨proof⟩

lemma *closed-Collect-neg*: *open* $\{x.\ P\ x\} \Longrightarrow$ *closed* $\{x.\ \neg\ P\ x\}$
⟨proof⟩

lemma *closed-Collect-conj*:
assumes *closed* $\{x.\ P\ x\}$ *closed* $\{x.\ Q\ x\}$
shows *closed* $\{x.\ P\ x \wedge Q\ x\}$
⟨proof⟩

lemma *closed-Collect-disj*:
assumes *closed* $\{x.\ P\ x\}$ *closed* $\{x.\ Q\ x\}$
shows *closed* $\{x.\ P\ x \vee Q\ x\}$
⟨proof⟩

lemma *closed-Collect-all*: $(\bigwedge i.$ *closed* $\{x.\ P\ i\ x\}) \Longrightarrow$ *closed* $\{x.\ \forall\, i.\ P\ i\ x\}$
⟨proof⟩

lemma *closed-Collect-imp*: *open* $\{x.\ P\ x\} \Longrightarrow$ *closed* $\{x.\ Q\ x\} \Longrightarrow$ *closed* $\{x.\ P\ x$

$\longrightarrow Q\ x\}$
⟨*proof*⟩

lemma *closed-Collect-const*: *closed* $\{x.\ P\}$
⟨*proof*⟩

end

97.2 Hausdorff and other separation properties

class *t0-space* = *topological-space* +
assumes *t0-space*: $x \neq y \Longrightarrow \exists\, U.\ open\ U \wedge \neg\, (x \in U \longleftrightarrow y \in U)$

class *t1-space* = *topological-space* +
assumes *t1-space*: $x \neq y \Longrightarrow \exists\, U.\ open\ U \wedge x \in U \wedge y \notin U$

instance *t1-space* ⊆ *t0-space*
⟨*proof*⟩

context *t1-space* **begin**

lemma *separation-t1*: $x \neq y \longleftrightarrow (\exists\, U.\ open\ U \wedge x \in U \wedge y \notin U)$
⟨*proof*⟩

lemma *closed-singleton* [*iff*]: *closed* $\{a\}$
⟨*proof*⟩

lemma *closed-insert* [*continuous-intros*, *simp*]:
assumes *closed S*
shows *closed* (*insert a S*)
⟨*proof*⟩

lemma *finite-imp-closed*: *finite S* $\Longrightarrow$ *closed S*
⟨*proof*⟩

end

T2 spaces are also known as Hausdorff spaces.

class *t2-space* = *topological-space* +
assumes *hausdorff*: $x \neq y \Longrightarrow \exists\, U\ V.\ open\ U \wedge open\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$

instance *t2-space* ⊆ *t1-space*
⟨*proof*⟩

lemma (**in** *t2-space*) *separation-t2*: $x \neq y \longleftrightarrow (\exists\, U\ V.\ open\ U \wedge open\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\})$
⟨*proof*⟩

lemma (**in** *t0-space*) *separation-t0*: $x \neq y \longleftrightarrow (\exists\, U.\ open\ U \wedge \neg\ (x \in U \longleftrightarrow y \in U))$
  ⟨*proof*⟩

A perfect space is a topological space with no isolated points.

class *perfect-space* = *topological-space* +
  assumes *not-open-singleton*: $\neg$ *open* $\{x\}$

lemma (**in** *perfect-space*) *UNIV-not-singleton*: *UNIV* $\neq \{x\}$
  for x::$'a$
  ⟨*proof*⟩

97.3 Generators for toplogies

inductive *generate-topology* :: $'a$ *set set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ *bool* **for** S :: $'a$ *set set*
  where
    UNIV: *generate-topology* S *UNIV*
  | *Int*: *generate-topology* S $(a \cap b)$ **if** *generate-topology* S a **and** *generate-topology* S b
  | *UN*: *generate-topology* S $(\bigcup K)$ **if** $(\bigwedge k.\ k \in K \Longrightarrow$ *generate-topology* S $k)$
  | *Basis*: *generate-topology* S s **if** $s \in S$

hide-fact (**open**) *UNIV Int UN Basis*

lemma *generate-topology-Union*:
  $(\bigwedge k.\ k \in I \Longrightarrow$ *generate-topology* S $(K\ k)) \Longrightarrow$ *generate-topology* S $(\bigcup k{\in}I.\ K\ k)$
  ⟨*proof*⟩

lemma *topological-space-generate-topology*: *class.topological-space* (*generate-topology* S)
  ⟨*proof*⟩

97.4 Order topologies

class *order-topology* = *order* + *open* +
  assumes *open-generated-order*: *open* = *generate-topology* (*range* $(\lambda a.\ \{..< a\})$ $\cup$ *range* $(\lambda a.\ \{a <..\})$)
begin

subclass *topological-space*
  ⟨*proof*⟩

lemma *open-greaterThan* [*continuous-intros*, *simp*]: *open* $\{a <..\}$
  ⟨*proof*⟩

lemma *open-lessThan* [*continuous-intros*, *simp*]: *open* $\{..< a\}$
  ⟨*proof*⟩

lemma *open-greaterThanLessThan* [*continuous-intros*, *simp*]: *open* $\{a <..< b\}$
⟨*proof*⟩

end

class *linorder-topology* = *linorder* + *order-topology*

lemma *closed-atMost* [*continuous-intros*, *simp*]: *closed* $\{..a\}$
for a :: *'a::linorder-topology*
⟨*proof*⟩

lemma *closed-atLeast* [*continuous-intros*, *simp*]: *closed* $\{a..\}$
for a :: *'a::linorder-topology*
⟨*proof*⟩

lemma *closed-atLeastAtMost* [*continuous-intros*, *simp*]: *closed* $\{a..b\}$
for $a\ b$ :: *'a::linorder-topology*
⟨*proof*⟩

lemma (**in** *linorder*) *less-separate*:
assumes $x < y$
shows $\exists\, a\ b.\ x \in \{..< a\} \wedge y \in \{b <..\} \wedge \{..< a\} \cap \{b <..\} = \{\}$
⟨*proof*⟩

instance *linorder-topology* ⊆ *t2-space*
⟨*proof*⟩

lemma (**in** *linorder-topology*) *open-right*:
assumes *open* S $x \in S$
and *gt-ex*: $x < y$
shows $\exists\, b{>}x.\ \{x ..< b\} \subseteq S$
⟨*proof*⟩

lemma (**in** *linorder-topology*) *open-left*:
assumes *open* S $x \in S$
and *lt-ex*: $y < x$
shows $\exists\, b{<}x.\ \{b <.. x\} \subseteq S$
⟨*proof*⟩

97.5 Setup some topologies

97.5.1 Boolean is an order topology

class *discrete-topology* = *topological-space* +
assumes *open-discrete*: $\bigwedge A.$ *open* A

instance *discrete-topology* < *t2-space*
⟨*proof*⟩

instantiation *bool* :: *linorder-topology*

begin

definition *open-bool* :: *bool set* ⇒ *bool*
where *open-bool* = *generate-topology* (*range* (λa. {..< a}) ∪ *range* (λa. {a <..}))

instance
⟨*proof*⟩

end

instance *bool* :: *discrete-topology*
⟨*proof*⟩

instantiation *nat* :: *linorder-topology*
begin

definition *open-nat* :: *nat set* ⇒ *bool*
where *open-nat* = *generate-topology* (*range* (λa. {..< a}) ∪ *range* (λa. {a <..}))

instance
⟨*proof*⟩

end

instance *nat* :: *discrete-topology*
⟨*proof*⟩

instantiation *int* :: *linorder-topology*
begin

definition *open-int* :: *int set* ⇒ *bool*
where *open-int* = *generate-topology* (*range* (λa. {..< a}) ∪ *range* (λa. {a <..}))

instance
⟨*proof*⟩

end

instance *int* :: *discrete-topology*
⟨*proof*⟩

97.5.2 Topological filters

definition (**in** *topological-space*) *nhds* :: *'a* ⇒ *'a filter*
where *nhds a* = (*INF S*:{*S*. *open S* ∧ *a* ∈ *S*}. *principal S*)

definition (**in** *topological-space*) *at-within* :: *'a* ⇒ *'a set* ⇒ *'a filter*
(*at* (-)/ *within* (-) [*1000*, *60*] *60*)
where *at a within s* = *inf* (*nhds a*) (*principal* (*s* − {*a*}))

abbreviation (**in** *topological-space*) *at* :: $'a \Rightarrow 'a$ *filter* (*at*)
where *at* $x \equiv$ *at* x *within* (*CONST UNIV*)

abbreviation (**in** *order-topology*) *at-right* :: $'a \Rightarrow 'a$ *filter*
where *at-right* $x \equiv$ *at* x *within* $\{x <..\}$

abbreviation (**in** *order-topology*) *at-left* :: $'a \Rightarrow 'a$ *filter*
where *at-left* $x \equiv$ *at* x *within* $\{..< x\}$

lemma (**in** *topological-space*) *nhds-generated-topology*:
open = *generate-topology* $T \Longrightarrow$ *nhds* x = (*INF* S:$\{S \in T.\ x \in S\}$. *principal* S)
⟨*proof*⟩

lemma (**in** *topological-space*) *eventually-nhds*:
eventually P (*nhds* a) $\longleftrightarrow$ ($\exists S$. *open* $S \wedge a \in S \wedge (\forall x \in S.\ P\ x)$)
⟨*proof*⟩

lemma *eventually-eventually*:
eventually (λy. *eventually* P (*nhds* y)) (*nhds* x) = *eventually* P (*nhds* x)
⟨*proof*⟩

lemma (**in** *topological-space*) *eventually-nhds-in-open*:
open $s \Longrightarrow x \in s \Longrightarrow$ *eventually* ($\lambda y.\ y \in s$) (*nhds* x)
⟨*proof*⟩

lemma (**in** *topological-space*) *eventually-nhds-x-imp-x*: *eventually* P (*nhds* x) $\Longrightarrow$ $P\ x$
⟨*proof*⟩

lemma (**in** *topological-space*) *nhds-neq-bot* [*simp*]: *nhds* $a \neq$ *bot*
⟨*proof*⟩

lemma (**in** *t1-space*) *t1-space-nhds*: $x \neq y \Longrightarrow$ ($\forall_F$ x *in nhds* x. $x \neq y$)
⟨*proof*⟩

lemma (**in** *topological-space*) *nhds-discrete-open*: *open* $\{x\} \Longrightarrow$ *nhds* x = *principal* $\{x\}$
⟨*proof*⟩

lemma (**in** *discrete-topology*) *nhds-discrete*: *nhds* x = *principal* $\{x\}$
⟨*proof*⟩

lemma (**in** *discrete-topology*) *at-discrete*: *at* x *within* S = *bot*
⟨*proof*⟩

lemma (**in** *topological-space*) *at-within-eq*:
at x *within* s = (*INF* S:$\{S.$ *open* $S \wedge x \in S\}$. *principal* ($S \cap s - \{x\}$))
⟨*proof*⟩

lemma (**in** *topological-space*) *eventually-at-filter*:
eventually P (*at* a *within* s) $\longleftrightarrow$ *eventually* ($\lambda x.\ x \neq a \longrightarrow x \in s \longrightarrow P\ x$) (*nhds* a)
⟨*proof*⟩

lemma (**in** *topological-space*) *at-le*: $s \subseteq t \Longrightarrow$ *at* x *within* $s \leq$ *at* x *within* t
⟨*proof*⟩

lemma (**in** *topological-space*) *eventually-at-topological*:
eventually P (*at* a *within* s) $\longleftrightarrow$ ($\exists S.$ *open* $S \wedge a \in S \wedge (\forall x{\in}S.\ x \neq a \longrightarrow x \in s \longrightarrow P\ x)$)
⟨*proof*⟩

lemma (**in** *topological-space*) *at-within-open*: $a \in S \Longrightarrow$ *open* $S \Longrightarrow$ *at* a *within* S $=$ *at* a
⟨*proof*⟩

lemma (**in** *topological-space*) *at-within-open-NO-MATCH*:
$a \in s \Longrightarrow$ *open* $s \Longrightarrow$ *NO-MATCH UNIV* $s \Longrightarrow$ *at* a *within* s $=$ *at* a
⟨*proof*⟩

lemma (**in** *topological-space*) *at-within-open-subset*:
$a \in S \Longrightarrow$ *open* $S \Longrightarrow S \subseteq T \Longrightarrow$ *at* a *within* T $=$ *at* a
⟨*proof*⟩

lemma (**in** *topological-space*) *at-within-nhd*:
assumes $x \in S$ *open* S $T \cap S - \{x\} = U \cap S - \{x\}$
shows *at* x *within* T $=$ *at* x *within* U
⟨*proof*⟩

lemma (**in** *topological-space*) *at-within-empty* [*simp*]: *at* a *within* $\{\}$ $=$ *bot*
⟨*proof*⟩

lemma (**in** *topological-space*) *at-within-union*:
at x *within* $(S \cup T)$ $=$ *sup* (*at* x *within* S) (*at* x *within* T)
⟨*proof*⟩

lemma (**in** *topological-space*) *at-eq-bot-iff*: *at* a $=$ *bot* $\longleftrightarrow$ *open* $\{a\}$
⟨*proof*⟩

lemma (**in** *perfect-space*) *at-neq-bot* [*simp*]: *at* $a \neq$ *bot*
⟨*proof*⟩

lemma (**in** *order-topology*) *nhds-order*:
nhds x $=$ *inf* (*INF* a:$\{x <..\}$. *principal* $\{..< a\}$) (*INF* a:$\{..< x\}$. *principal* $\{a <..\}$)
⟨*proof*⟩

lemma (**in** *topological-space*) *filterlim-at-within-If*:
assumes *filterlim f G* (*at x within* ($A \cap \{x.\ P\ x\}$))
and *filterlim g G* (*at x within* ($A \cap \{x.\ \neg P\ x\}$))
shows *filterlim* ($\lambda x.$ *if P x then f x else g x*) *G* (*at x within A*)
⟨*proof*⟩

lemma (**in** *topological-space*) *filterlim-at-If*:
assumes *filterlim f G* (*at x within* $\{x.\ P\ x\}$)
and *filterlim g G* (*at x within* $\{x.\ \neg P\ x\}$)
shows *filterlim* ($\lambda x.$ *if P x then f x else g x*) *G* (*at x*)
⟨*proof*⟩

lemma (**in** *linorder-topology*) *at-within-order*:
assumes $UNIV \neq \{x\}$
shows *at x within s* =
inf (*INF a*:$\{x <..\}$. *principal* ($\{..< a\} \cap s - \{x\}$))
(*INF a*:$\{..< x\}$. *principal* ($\{a <..\} \cap s - \{x\}$))
⟨*proof*⟩

lemma (**in** *linorder-topology*) *at-left-eq*:
$y < x \Longrightarrow$ *at-left x* = (*INF a*:$\{..< x\}$. *principal* $\{a <..< x\}$)
⟨*proof*⟩

lemma (**in** *linorder-topology*) *eventually-at-left*:
$y < x \Longrightarrow$ *eventually P* (*at-left x*) $\longleftrightarrow$ ($\exists\, b<x.\ \forall\, y>b.\ y < x \longrightarrow P\ y$)
⟨*proof*⟩

lemma (**in** *linorder-topology*) *at-right-eq*:
$x < y \Longrightarrow$ *at-right x* = (*INF a*:$\{x <..\}$. *principal* $\{x <..< a\}$)
⟨*proof*⟩

lemma (**in** *linorder-topology*) *eventually-at-right*:
$x < y \Longrightarrow$ *eventually P* (*at-right x*) $\longleftrightarrow$ ($\exists\, b>x.\ \forall\, y>x.\ y < b \longrightarrow P\ y$)
⟨*proof*⟩

lemma *eventually-at-right-less*: $\forall_F$ *y in at-right* (*x*::'*a*::{*linorder-topology*, *no-top*}). $x < y$
⟨*proof*⟩

lemma *trivial-limit-at-right-top*: *at-right* (*top*::-::{*order-top*,*linorder-topology*}) = *bot*
⟨*proof*⟩

lemma *trivial-limit-at-left-bot*: *at-left* (*bot*::-::{*order-bot*,*linorder-topology*}) = *bot*
⟨*proof*⟩

lemma *trivial-limit-at-left-real* [*simp*]: ¬ *trivial-limit* (*at-left x*)
for *x* :: '*a*::{*no-bot*,*dense-order*,*linorder-topology*}
⟨*proof*⟩

lemma *trivial-limit-at-right-real* [*simp*]: $\neg$ *trivial-limit* (*at-right x*)
for x :: $'a$::{*no-top*,*dense-order*,*linorder-topology*}
⟨*proof*⟩

lemma (**in** *linorder-topology*) *at-eq-sup-left-right*: *at x* = *sup* (*at-left x*) (*at-right x*)
⟨*proof*⟩

lemma (**in** *linorder-topology*) *eventually-at-split*:
eventually P (*at x*) $\longleftrightarrow$ *eventually P* (*at-left x*) $\wedge$ *eventually P* (*at-right x*)
⟨*proof*⟩

lemma (**in** *order-topology*) *eventually-at-leftI*:
assumes $\bigwedge x.\ x \in \{a{<}..{<}b\} \Longrightarrow P\ x\ a < b$
shows *eventually P* (*at-left b*)
⟨*proof*⟩

lemma (**in** *order-topology*) *eventually-at-rightI*:
assumes $\bigwedge x.\ x \in \{a{<}..{<}b\} \Longrightarrow P\ x\ a < b$
shows *eventually P* (*at-right a*)
⟨*proof*⟩

lemma *eventually-filtercomap-nhds*:
eventually P (*filtercomap f* (*nhds x*)) $\longleftrightarrow$ ($\exists S.$ *open S* $\wedge$ $x \in S$ $\wedge$ ($\forall x.\ f\ x \in S \longrightarrow P\ x$))
⟨*proof*⟩

lemma *eventually-filtercomap-at-topological*:
eventually P (*filtercomap f* (*at A within B*)) $\longleftrightarrow$
($\exists S.$ *open S* $\wedge$ $A \in S$ $\wedge$ ($\forall x.\ f\ x \in S \cap B - \{A\} \longrightarrow P\ x$)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

97.5.3 Tendsto

abbreviation (**in** *topological-space*)
tendsto :: ($'b \Rightarrow 'a$) $\Rightarrow$ $'a \Rightarrow 'b$ *filter* $\Rightarrow$ *bool* (**infixr** $\longrightarrow$ *55*)
where ($f \longrightarrow l$) $F \equiv$ *filterlim f* (*nhds l*) F

definition (**in** *t2-space*) *Lim* :: $'f$ *filter* $\Rightarrow$ ($'f \Rightarrow 'a$) $\Rightarrow$ $'a$
where *Lim A f* = (*THE l.* ($f \longrightarrow l$) *A*)

lemma (**in** *topological-space*) *tendsto-eq-rhs*: ($f \longrightarrow x$) $F \Longrightarrow x = y \Longrightarrow$ ($f \longrightarrow y$) F
⟨*proof*⟩

named-theorems *tendsto-intros introduction rules for tendsto*
⟨*ML*⟩

context *topological-space* **begin**

lemma *tendsto-def*:
$(f \longrightarrow l)\ F \longleftrightarrow (\forall S.\ open\ S \longrightarrow l \in S \longrightarrow eventually\ (\lambda x.\ f\ x \in S)\ F)$
⟨*proof*⟩

lemma *tendsto-cong*: $(f \longrightarrow c)\ F \longleftrightarrow (g \longrightarrow c)\ F$ **if** $eventually\ (\lambda x.\ f\ x = g\ x)\ F$
⟨*proof*⟩

lemma *tendsto-mono*: $F \leq F' \Longrightarrow (f \longrightarrow l)\ F' \Longrightarrow (f \longrightarrow l)\ F$
⟨*proof*⟩

lemma *tendsto-ident-at* [*tendsto-intros*, *simp*, *intro*]: $((\lambda x.\ x) \longrightarrow a)\ (at\ a\ within\ s)$
⟨*proof*⟩

lemma *tendsto-const* [*tendsto-intros*, *simp*, *intro*]: $((\lambda x.\ k) \longrightarrow k)\ F$
⟨*proof*⟩

lemma *filterlim-at*:
$(LIM\ x\ F.\ f\ x :> at\ b\ within\ s) \longleftrightarrow eventually\ (\lambda x.\ f\ x \in s \wedge f\ x \neq b)\ F \wedge (f \longrightarrow b)\ F$
⟨*proof*⟩

lemma *filterlim-at-withinI*:
assumes $filterlim\ f\ (nhds\ c)\ F$
assumes $eventually\ (\lambda x.\ f\ x \in A - \{c\})\ F$
shows $filterlim\ f\ (at\ c\ within\ A)\ F$
⟨*proof*⟩

lemma *filterlim-atI*:
assumes $filterlim\ f\ (nhds\ c)\ F$
assumes $eventually\ (\lambda x.\ f\ x \neq c)\ F$
shows $filterlim\ f\ (at\ c)\ F$
⟨*proof*⟩

lemma *topological-tendstoI*:
$(\bigwedge S.\ open\ S \Longrightarrow l \in S \Longrightarrow eventually\ (\lambda x.\ f\ x \in S)\ F) \Longrightarrow (f \longrightarrow l)\ F$
⟨*proof*⟩

lemma *topological-tendstoD*:
$(f \longrightarrow l)\ F \Longrightarrow open\ S \Longrightarrow l \in S \Longrightarrow eventually\ (\lambda x.\ f\ x \in S)\ F$
⟨*proof*⟩

lemma *tendsto-bot* [*simp*]: $(f \longrightarrow a)\ bot$
⟨*proof*⟩

end

lemma *tendsto-within-subset*:
$(f \longrightarrow l)$ $(at\ x\ within\ S) \Longrightarrow T \subseteq S \Longrightarrow (f \longrightarrow l)$ $(at\ x\ within\ T)$
⟨*proof*⟩

lemma (**in** *order-topology*) *order-tendsto-iff*:
$(f \longrightarrow x)\ F \longleftrightarrow (\forall\, l{<}x.\ eventually\ (\lambda x.\ l < f\,x)\ F) \wedge (\forall\, u{>}x.\ eventually\ (\lambda x.\ f\,x < u)\ F)$
⟨*proof*⟩

lemma (**in** *order-topology*) *order-tendstoI*:
$(\bigwedge a.\ a < y \Longrightarrow eventually\ (\lambda x.\ a < f\,x)\ F) \Longrightarrow (\bigwedge a.\ y < a \Longrightarrow eventually\ (\lambda x.\ f\,x < a)\ F) \Longrightarrow$
$(f \longrightarrow y)\ F$
⟨*proof*⟩

lemma (**in** *order-topology*) *order-tendstoD*:
assumes $(f \longrightarrow y)\ F$
shows $a < y \Longrightarrow eventually\ (\lambda x.\ a < f\,x)\ F$
and $y < a \Longrightarrow eventually\ (\lambda x.\ f\,x < a)\ F$
⟨*proof*⟩

lemma (**in** *linorder-topology*) *tendsto-max*:
assumes X: $(X \longrightarrow x)\ net$
and Y: $(Y \longrightarrow y)\ net$
shows $((\lambda x.\ max\ (X\ x)\ (Y\ x)) \longrightarrow max\ x\ y)\ net$
⟨*proof*⟩

lemma (**in** *linorder-topology*) *tendsto-min*:
assumes X: $(X \longrightarrow x)\ net$
and Y: $(Y \longrightarrow y)\ net$
shows $((\lambda x.\ min\ (X\ x)\ (Y\ x)) \longrightarrow min\ x\ y)\ net$
⟨*proof*⟩

lemma (**in** *order-topology*)
assumes $a < b$
shows *at-within-Icc-at-right*: $at\ a\ within\ \{a..b\} = at\text{-}right\ a$
and *at-within-Icc-at-left*: $at\ b\ within\ \{a..b\} = at\text{-}left\ b$
⟨*proof*⟩

lemma (**in** *order-topology*) *at-within-Icc-at*: $a < x \Longrightarrow x < b \Longrightarrow at\ x\ within\ \{a..b\} = at\ x$
⟨*proof*⟩

lemma (**in** *t2-space*) *tendsto-unique*:
assumes $F \neq bot$
and $(f \longrightarrow a)\ F$
and $(f \longrightarrow b)\ F$
shows $a = b$

⟨*proof*⟩

lemma (**in** *t2-space*) *tendsto-const-iff*:
  fixes $a\ b :: 'a$
  assumes $\neg$ *trivial-limit* F
  shows $((\lambda x.\ a) \longrightarrow b)\ F \longleftrightarrow a = b$
  ⟨*proof*⟩

lemma (**in** *order-topology*) *increasing-tendsto*:
  assumes *bdd*: *eventually* $(\lambda n.\ f\ n \leq l)\ F$
    and *en*: $\bigwedge x.\ x < l \Longrightarrow$ *eventually* $(\lambda n.\ x < f\ n)\ F$
  shows $(f \longrightarrow l)\ F$
  ⟨*proof*⟩

lemma (**in** *order-topology*) *decreasing-tendsto*:
  assumes *bdd*: *eventually* $(\lambda n.\ l \leq f\ n)\ F$
    and *en*: $\bigwedge x.\ l < x \Longrightarrow$ *eventually* $(\lambda n.\ f\ n < x)\ F$
  shows $(f \longrightarrow l)\ F$
  ⟨*proof*⟩

lemma (**in** *order-topology*) *tendsto-sandwich*:
  assumes *ev*: *eventually* $(\lambda n.\ f\ n \leq g\ n)$ *net* *eventually* $(\lambda n.\ g\ n \leq h\ n)$ *net*
  assumes *lim*: $(f \longrightarrow c)$ *net* $(h \longrightarrow c)$ *net*
  shows $(g \longrightarrow c)$ *net*
⟨*proof*⟩

lemma (**in** *t1-space*) *limit-frequently-eq*:
  assumes $F \neq$ *bot*
    and *frequently* $(\lambda x.\ f\ x = c)\ F$
    and $(f \longrightarrow d)\ F$
  shows $d = c$
⟨*proof*⟩

lemma (**in** *t1-space*) *tendsto-imp-eventually-ne*:
  assumes $(f \longrightarrow c)\ F\ c \neq c'$
  shows *eventually* $(\lambda z.\ f\ z \neq c')\ F$
⟨*proof*⟩

lemma (**in** *linorder-topology*) *tendsto-le*:
  assumes F: $\neg$ *trivial-limit* F
    and x: $(f \longrightarrow x)\ F$
    and y: $(g \longrightarrow y)\ F$
    and *ev*: *eventually* $(\lambda x.\ g\ x \leq f\ x)\ F$
  shows $y \leq x$
⟨*proof*⟩

lemma (**in** *linorder-topology*) *tendsto-lowerbound*:
  assumes x: $(f \longrightarrow x)\ F$
      and *ev*: *eventually* $(\lambda i.\ a \leq f\ i)\ F$

and F: $\neg$ *trivial-limit* F
shows $a \leq x$
⟨*proof*⟩

lemma (**in** *linorder-topology*) *tendsto-upperbound*:
assumes x: $(f \longrightarrow x)\ F$
and ev: *eventually* $(\lambda i.\ a \geq f\ i)\ F$
and F: $\neg$ *trivial-limit* F
shows $a \geq x$
⟨*proof*⟩

97.5.4 Rules about *Lim*

lemma *tendsto-Lim*: $\neg$ *trivial-limit net* $\Longrightarrow$ $(f \longrightarrow l)$ *net* $\Longrightarrow$ *Lim net* $f = l$
⟨*proof*⟩

lemma *Lim-ident-at*: $\neg$ *trivial-limit* (*at* x *within* s) $\Longrightarrow$ *Lim* (*at* x *within* s) $(\lambda x.\ x) = x$
⟨*proof*⟩

lemma *filterlim-at-bot-at-right*:
fixes f :: *'a::linorder-topology* $\Rightarrow$ *'b::linorder*
assumes *mono*: $\bigwedge x\ y.\ Q\ x \Longrightarrow Q\ y \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
and *bij*: $\bigwedge x.\ P\ x \Longrightarrow f\ (g\ x) = x\ \bigwedge x.\ P\ x \Longrightarrow Q\ (g\ x)$
and Q: *eventually* Q (*at-right* a)
and *bound*: $\bigwedge b.\ Q\ b \Longrightarrow a < b$
and P: *eventually* P *at-bot*
shows *filterlim* f *at-bot* (*at-right* a)
⟨*proof*⟩

lemma *filterlim-at-top-at-left*:
fixes f :: *'a::linorder-topology* $\Rightarrow$ *'b::linorder*
assumes *mono*: $\bigwedge x\ y.\ Q\ x \Longrightarrow Q\ y \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
and *bij*: $\bigwedge x.\ P\ x \Longrightarrow f\ (g\ x) = x\ \bigwedge x.\ P\ x \Longrightarrow Q\ (g\ x)$
and Q: *eventually* Q (*at-left* a)
and *bound*: $\bigwedge b.\ Q\ b \Longrightarrow b < a$
and P: *eventually* P *at-top*
shows *filterlim* f *at-top* (*at-left* a)
⟨*proof*⟩

lemma *filterlim-split-at*:
filterlim f F (*at-left* x) $\Longrightarrow$ *filterlim* f F (*at-right* x) $\Longrightarrow$
filterlim f F (*at* x)
for x :: *'a::linorder-topology*
⟨*proof*⟩

lemma *filterlim-at-split*:
filterlim f F (*at* x) $\longleftrightarrow$ *filterlim* f F (*at-left* x) $\wedge$ *filterlim* f F (*at-right* x)
for x :: *'a::linorder-topology*

⟨*proof*⟩

lemma *eventually-nhds-top*:
fixes P :: $'a$ :: {*order-top*,*linorder-topology*} ⇒ *bool*
and b :: $'a$
assumes $b < top$
shows *eventually* P (*nhds top*) $\longleftrightarrow$ ($\exists\, b{<}top.\ (\forall\, z.\ b < z \longrightarrow P\ z)$)
⟨*proof*⟩

lemma *tendsto-at-within-iff-tendsto-nhds*:
($g \longrightarrow g\ l$) (*at* l *within* S) $\longleftrightarrow$ ($g \longrightarrow g\ l$) (*inf* (*nhds* l) (*principal* S))
⟨*proof*⟩

97.6 Limits on sequences

abbreviation (**in** *topological-space*)
LIMSEQ :: [*nat* ⇒ $'a$, $'a$] ⇒ *bool* (((-)/ $\longrightarrow$ (-)) [*60*, *60*] *60*)
where $X \longrightarrow L \equiv (X \longrightarrow L)$ *sequentially*

abbreviation (**in** *t2-space*) *lim* :: (*nat* ⇒ $'a$) ⇒ $'a$
where *lim* $X \equiv$ *Lim sequentially* X

definition (**in** *topological-space*) *convergent* :: (*nat* ⇒ $'a$) ⇒ *bool*
where *convergent* $X = (\exists\, L.\ X \longrightarrow L)$

lemma *lim-def*: *lim* X = (*THE* L. $X \longrightarrow L$)
⟨*proof*⟩

97.6.1 Monotone sequences and subsequences

Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing first.

definition *monoseq* :: (*nat* ⇒ $'a$::*order*) ⇒ *bool*
where *monoseq* $X \longleftrightarrow (\forall\, m.\ \forall\, n{\geq}m.\ X\ m \leq X\ n) \vee (\forall\, m.\ \forall\, n{\geq}m.\ X\ n \leq X\ m)$

abbreviation *incseq* :: (*nat* ⇒ $'a$::*order*) ⇒ *bool*
where *incseq* $X \equiv$ *mono* X

lemma *incseq-def*: *incseq* $X \longleftrightarrow (\forall\, m.\ \forall\, n{\geq}m.\ X\ n \geq X\ m)$
⟨*proof*⟩

abbreviation *decseq* :: (*nat* ⇒ $'a$::*order*) ⇒ *bool*
where *decseq* $X \equiv$ *antimono* X

lemma *decseq-def*: *decseq* $X \longleftrightarrow (\forall\, m.\ \forall\, n{\geq}m.\ X\ n \leq X\ m)$
⟨*proof*⟩

Definition of subsequence.

lemma *strict-mono-leD*: $strict\text{-}mono\ r \Longrightarrow m \le n \Longrightarrow r\ m \le r\ n$
⟨*proof*⟩

lemma *strict-mono-id*: $strict\text{-}mono\ id$
⟨*proof*⟩

lemma *incseq-SucI*: $(\bigwedge n.\ X\ n \le X\ (Suc\ n)) \Longrightarrow incseq\ X$
⟨*proof*⟩

lemma *incseqD*: $incseq\ f \Longrightarrow i \le j \Longrightarrow f\ i \le f\ j$
⟨*proof*⟩

lemma *incseq-SucD*: $incseq\ A \Longrightarrow A\ i \le A\ (Suc\ i)$
⟨*proof*⟩

lemma *incseq-Suc-iff*: $incseq\ f \longleftrightarrow (\forall n.\ f\ n \le f\ (Suc\ n))$
⟨*proof*⟩

lemma *incseq-const*[*simp*, *intro*]: $incseq\ (\lambda x.\ k)$
⟨*proof*⟩

lemma *decseq-SucI*: $(\bigwedge n.\ X\ (Suc\ n) \le X\ n) \Longrightarrow decseq\ X$
⟨*proof*⟩

lemma *decseqD*: $decseq\ f \Longrightarrow i \le j \Longrightarrow f\ j \le f\ i$
⟨*proof*⟩

lemma *decseq-SucD*: $decseq\ A \Longrightarrow A\ (Suc\ i) \le A\ i$
⟨*proof*⟩

lemma *decseq-Suc-iff*: $decseq\ f \longleftrightarrow (\forall n.\ f\ (Suc\ n) \le f\ n)$
⟨*proof*⟩

lemma *decseq-const*[*simp*, *intro*]: $decseq\ (\lambda x.\ k)$
⟨*proof*⟩

lemma *monoseq-iff*: $monoseq\ X \longleftrightarrow incseq\ X \lor decseq\ X$
⟨*proof*⟩

lemma *monoseq-Suc*: $monoseq\ X \longleftrightarrow (\forall n.\ X\ n \le X\ (Suc\ n)) \lor (\forall n.\ X\ (Suc\ n)$
$\le X\ n)$
⟨*proof*⟩

lemma *monoI1*: $\forall m.\ \forall n \ge m.\ X\ m \le X\ n \Longrightarrow monoseq\ X$
⟨*proof*⟩

lemma *monoI2*: $\forall m.\ \forall n \ge m.\ X\ n \le X\ m \Longrightarrow monoseq\ X$
⟨*proof*⟩

lemma *mono-SucI1*: $\forall n.\ X\ n \leq X\ (Suc\ n) \Longrightarrow monoseq\ X$
⟨*proof*⟩

lemma *mono-SucI2*: $\forall n.\ X\ (Suc\ n) \leq X\ n \Longrightarrow monoseq\ X$
⟨*proof*⟩

lemma *monoseq-minus*:
fixes $a :: nat \Rightarrow 'a::ordered\text{-}ab\text{-}group\text{-}add$
assumes *monoseq a*
shows $monoseq\ (\lambda\ n.\ -\ a\ n)$
⟨*proof*⟩

Subsequence (alternative definition, (e.g. Hoskins)

lemma *strict-mono-Suc-iff*: $strict\text{-}mono\ f \longleftrightarrow (\forall n.\ f\ n < f\ (Suc\ n))$
⟨*proof*⟩

lemma *strict-mono-add*: $strict\text{-}mono\ (\lambda n::'a::linordered\text{-}semidom.\ n + k)$
⟨*proof*⟩

For any sequence, there is a monotonic subsequence.

lemma *seq-monosub*:
fixes $s :: nat \Rightarrow 'a::linorder$
shows $\exists f.\ strict\text{-}mono\ f \wedge monoseq\ (\lambda n.\ (s\ (f\ n)))$
⟨*proof*⟩

lemma *seq-suble*:
assumes *sf*: $strict\text{-}mono\ (f :: nat \Rightarrow nat)$
shows $n \leq f\ n$
⟨*proof*⟩

lemma *eventually-subseq*:
$strict\text{-}mono\ r \Longrightarrow eventually\ P\ sequentially \Longrightarrow eventually\ (\lambda n.\ P\ (r\ n))\ sequentially$
⟨*proof*⟩

lemma *not-eventually-sequentiallyD*:
assumes $\neg\ eventually\ P\ sequentially$
shows $\exists r::nat \Rightarrow nat.\ strict\text{-}mono\ r \wedge (\forall n.\ \neg\ P\ (r\ n))$
⟨*proof*⟩

lemma *filterlim-subseq*: $strict\text{-}mono\ f \Longrightarrow filterlim\ f\ sequentially\ sequentially$
⟨*proof*⟩

lemma *strict-mono-o*: $strict\text{-}mono\ r \Longrightarrow strict\text{-}mono\ s \Longrightarrow strict\text{-}mono\ (r \circ s)$
⟨*proof*⟩

lemma *incseq-imp-monoseq*: $incseq\ X \Longrightarrow monoseq\ X$
⟨*proof*⟩

lemma *decseq-imp-monoseq*: *decseq* $X \Longrightarrow$ *monoseq* X
⟨*proof*⟩

lemma *decseq-eq-incseq*: *decseq* X = *incseq* $(\lambda n.\ - X\ n)$
for X :: *nat* ⇒ *'a::ordered-ab-group-add*
⟨*proof*⟩

lemma *INT-decseq-offset*:
assumes *decseq* F
shows $(\bigcap i.\ F\ i) = (\bigcap i{\in}\{n..\}.\ F\ i)$
⟨*proof*⟩

lemma *LIMSEQ-const-iff*: $(\lambda n.\ k) \longrightarrow l \longleftrightarrow k = l$
for $k\ l$ :: *'a::t2-space*
⟨*proof*⟩

lemma *LIMSEQ-SUP*: *incseq* $X \Longrightarrow X \longrightarrow$ (*SUP* $i.\ X\ i$ :: *'a::{complete-linorder,linorder-topology}*)
⟨*proof*⟩

lemma *LIMSEQ-INF*: *decseq* $X \Longrightarrow X \longrightarrow$ (*INF* $i.\ X\ i$ :: *'a::{complete-linorder,linorder-topology}*)
⟨*proof*⟩

lemma *LIMSEQ-ignore-initial-segment*: $f \longrightarrow a \Longrightarrow (\lambda n.\ f\ (n + k)) \longrightarrow a$
⟨*proof*⟩

lemma *LIMSEQ-offset*: $(\lambda n.\ f\ (n + k)) \longrightarrow a \Longrightarrow f \longrightarrow a$
⟨*proof*⟩

lemma *LIMSEQ-Suc*: $f \longrightarrow l \Longrightarrow (\lambda n.\ f\ (Suc\ n)) \longrightarrow l$
⟨*proof*⟩

lemma *LIMSEQ-imp-Suc*: $(\lambda n.\ f\ (Suc\ n)) \longrightarrow l \Longrightarrow f \longrightarrow l$
⟨*proof*⟩

lemma *LIMSEQ-Suc-iff*: $(\lambda n.\ f\ (Suc\ n)) \longrightarrow l = f \longrightarrow l$
⟨*proof*⟩

lemma *LIMSEQ-unique*: $X \longrightarrow a \Longrightarrow X \longrightarrow b \Longrightarrow a = b$
for $a\ b$ :: *'a::t2-space*
⟨*proof*⟩

lemma *LIMSEQ-le-const*: $X \longrightarrow x \Longrightarrow \exists N.\ \forall n{\ge}N.\ a \le X\ n \Longrightarrow a \le x$
for $a\ x$ :: *'a::linorder-topology*
⟨*proof*⟩

lemma *LIMSEQ-le*: $X \longrightarrow x \Longrightarrow Y \longrightarrow y \Longrightarrow \exists N.\ \forall n{\ge}N.\ X\ n \le Y\ n \Longrightarrow x \le y$
for $x\ y$ :: *'a::linorder-topology*

⟨*proof*⟩

lemma *LIMSEQ-le-const2*: $X \longrightarrow x \Longrightarrow \exists N. \forall n \geq N. X\ n \leq a \Longrightarrow x \leq a$
for $a\ x$ :: *'a::linorder-topology*
⟨*proof*⟩

lemma *convergentD*: *convergent* $X \Longrightarrow \exists L. X \longrightarrow L$
⟨*proof*⟩

lemma *convergentI*: $X \longrightarrow L \Longrightarrow$ *convergent* X
⟨*proof*⟩

lemma *convergent-LIMSEQ-iff*: *convergent* $X \longleftrightarrow X \longrightarrow$ *lim* X
⟨*proof*⟩

lemma *convergent-const*: *convergent* $(\lambda n.\ c)$
⟨*proof*⟩

lemma *monoseq-le*:
monoseq $a \Longrightarrow a \longrightarrow x \Longrightarrow$
$(\forall n.\ a\ n \leq x) \wedge (\forall m. \forall n \geq m.\ a\ m \leq a\ n) \vee$
$(\forall n.\ x \leq a\ n) \wedge (\forall m. \forall n \geq m.\ a\ n \leq a\ m)$
for x :: *'a::linorder-topology*
⟨*proof*⟩

lemma *LIMSEQ-subseq-LIMSEQ*: $X \longrightarrow L \Longrightarrow$ *strict-mono* $f \Longrightarrow (X \circ f) \longrightarrow L$
⟨*proof*⟩

lemma *convergent-subseq-convergent*: *convergent* $X \Longrightarrow$ *strict-mono* $f \Longrightarrow$ *convergent* $(X \circ f)$
⟨*proof*⟩

lemma *limI*: $X \longrightarrow L \Longrightarrow$ *lim* $X = L$
⟨*proof*⟩

lemma *lim-le*: *convergent* $f \Longrightarrow (\bigwedge n.\ f\ n \leq x) \Longrightarrow$ *lim* $f \leq x$
for x :: *'a::linorder-topology*
⟨*proof*⟩

lemma *lim-const* [*simp*]: *lim* $(\lambda m.\ a) = a$
⟨*proof*⟩

97.6.2 Increasing and Decreasing Series

lemma *incseq-le*: *incseq* $X \Longrightarrow X \longrightarrow L \Longrightarrow X\ n \leq L$
for L :: *'a::linorder-topology*
⟨*proof*⟩

lemma *decseq-le*: *decseq* $X \Longrightarrow X \longrightarrow L \Longrightarrow L \leq X\ n$
for L :: *'a::linorder-topology*
⟨*proof*⟩

97.7 First countable topologies

class *first-countable-topology* = *topological-space* +
assumes *first-countable-basis*:
$\exists A{::}nat \Rightarrow 'a\ set.\ (\forall i.\ x \in A\ i \wedge open\ (A\ i)) \wedge (\forall S.\ open\ S \wedge x \in S \longrightarrow (\exists i.\ A\ i \subseteq S))$

lemma (**in** *first-countable-topology*) *countable-basis-at-decseq*:
obtains $A :: nat \Rightarrow 'a\ set$ **where**
$\bigwedge i.\ open\ (A\ i)\ \bigwedge i.\ x \in (A\ i)$
$\bigwedge S.\ open\ S \Longrightarrow x \in S \Longrightarrow eventually\ (\lambda i.\ A\ i \subseteq S)\ sequentially$
⟨*proof*⟩

lemma (**in** *first-countable-topology*) *nhds-countable*:
obtains $X :: nat \Rightarrow 'a\ set$
where $decseq\ X\ \bigwedge n.\ open\ (X\ n)\ \bigwedge n.\ x \in X\ n\ nhds\ x = (INF\ n.\ principal\ (X\ n))$
⟨*proof*⟩

lemma (**in** *first-countable-topology*) *countable-basis*:
obtains $A :: nat \Rightarrow 'a\ set$ **where**
$\bigwedge i.\ open\ (A\ i)\ \bigwedge i.\ x \in A\ i$
$\bigwedge F.\ (\forall n.\ F\ n \in A\ n) \Longrightarrow F \longrightarrow x$
⟨*proof*⟩

lemma (**in** *first-countable-topology*) *sequentially-imp-eventually-nhds-within*:
assumes $\forall f.\ (\forall n.\ f\ n \in s) \wedge f \longrightarrow a \longrightarrow eventually\ (\lambda n.\ P\ (f\ n))\ sequentially$
shows $eventually\ P\ (inf\ (nhds\ a)\ (principal\ s))$
⟨*proof*⟩

lemma (**in** *first-countable-topology*) *eventually-nhds-within-iff-sequentially*:
$eventually\ P\ (inf\ (nhds\ a)\ (principal\ s)) \longleftrightarrow$
$(\forall f.\ (\forall n.\ f\ n \in s) \wedge f \longrightarrow a \longrightarrow eventually\ (\lambda n.\ P\ (f\ n))\ sequentially)$
⟨*proof*⟩

lemma (**in** *first-countable-topology*) *eventually-nhds-iff-sequentially*:
$eventually\ P\ (nhds\ a) \longleftrightarrow (\forall f.\ f \longrightarrow a \longrightarrow eventually\ (\lambda n.\ P\ (f\ n))\ sequentially)$
⟨*proof*⟩

lemma *tendsto-at-iff-sequentially*:
$(f \longrightarrow a)\ (at\ x\ within\ s) \longleftrightarrow (\forall X.\ (\forall i.\ X\ i \in s - \{x\}) \longrightarrow X \longrightarrow x \longrightarrow ((f \circ X) \longrightarrow a))$
for f :: *'a::first-countable-topology* ⇒ -
⟨*proof*⟩

lemma *approx-from-above-dense-linorder*:
fixes $x::'a::\{dense\text{-}linorder, linorder\text{-}topology, first\text{-}countable\text{-}topology\}$
assumes $x < y$
shows $\exists u. (\forall n. u\ n > x) \wedge (u \longrightarrow x)$
⟨*proof*⟩

lemma *approx-from-below-dense-linorder*:
fixes $x::'a::\{dense\text{-}linorder, linorder\text{-}topology, first\text{-}countable\text{-}topology\}$
assumes $x > y$
shows $\exists u. (\forall n. u\ n < x) \wedge (u \longrightarrow x)$
⟨*proof*⟩

97.8 Function limit at a point

abbreviation $LIM :: ('a::topological\text{-}space \Rightarrow 'b::topological\text{-}space) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
$(((\text{-})/\ -(\text{-})/\!\rightarrow (\text{-}))\ [60,\ 0,\ 60]\ 60)$
where $f\ -a\!\rightarrow L \equiv (f \longrightarrow L)\ (at\ a)$

lemma *tendsto-within-open*: $a \in S \Longrightarrow open\ S \Longrightarrow (f \longrightarrow l)\ (at\ a\ within\ S) \longleftrightarrow (f\ -a\!\rightarrow l)$
⟨*proof*⟩

lemma *tendsto-within-open-NO-MATCH*:
$a \in S \Longrightarrow NO\text{-}MATCH\ UNIV\ S \Longrightarrow open\ S \Longrightarrow (f \longrightarrow l)(at\ a\ within\ S) \longleftrightarrow (f \longrightarrow l)(at\ a)$
for $f :: 'a::topological\text{-}space \Rightarrow 'b::topological\text{-}space$
⟨*proof*⟩

lemma *LIM-const-not-eq*[*tendsto-intros*]: $k \neq L \Longrightarrow \neg\ (\lambda x.\ k)\ -a\!\rightarrow L$
for $a :: 'a::perfect\text{-}space$ **and** $k\ L :: 'b::t2\text{-}space$
⟨*proof*⟩

lemmas *LIM-not-zero* = *LIM-const-not-eq* [**where** $L = 0$]

lemma *LIM-const-eq*: $(\lambda x.\ k)\ -a\!\rightarrow L \Longrightarrow k = L$
for $a :: 'a::perfect\text{-}space$ **and** $k\ L :: 'b::t2\text{-}space$
⟨*proof*⟩

lemma *LIM-unique*: $f\ -a\!\rightarrow L \Longrightarrow f\ -a\!\rightarrow M \Longrightarrow L = M$
for $a :: 'a::perfect\text{-}space$ **and** $L\ M :: 'b::t2\text{-}space$
⟨*proof*⟩

Limits are equal for functions equal except at limit point.

lemma *LIM-equal*: $\forall x.\ x \neq a \longrightarrow f\ x = g\ x \Longrightarrow (f\ -a\!\rightarrow l) \longleftrightarrow (g\ -a\!\rightarrow l)$
⟨*proof*⟩

lemma *LIM-cong*: $a = b \Longrightarrow (\bigwedge x.\ x \neq b \Longrightarrow f\ x = g\ x) \Longrightarrow l = m \Longrightarrow (f\ -a\!\rightarrow$

$l) \longleftrightarrow (g\ -b\rightarrow\ m)$
⟨proof⟩

lemma *LIM-cong-limit*: $f\ -x\rightarrow\ L \Longrightarrow K = L \Longrightarrow f\ -x\rightarrow\ K$
⟨proof⟩

lemma *tendsto-at-iff-tendsto-nhds*: $g\ -l\rightarrow\ g\ l \longleftrightarrow (g \longrightarrow g\ l)\ (nhds\ l)$
⟨proof⟩

lemma *tendsto-compose*: $g\ -l\rightarrow\ g\ l \Longrightarrow (f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ g\ (f\ x)) \longrightarrow g\ l)\ F$
⟨proof⟩

lemma *tendsto-compose-eventually*:
$g\ -l\rightarrow\ m \Longrightarrow (f \longrightarrow l)\ F \Longrightarrow eventually\ (\lambda x.\ f\ x \neq l)\ F \Longrightarrow ((\lambda x.\ g\ (f\ x)) \longrightarrow m)\ F$
⟨proof⟩

lemma *LIM-compose-eventually*:
assumes $f\ -a\rightarrow\ b$
and $g\ -b\rightarrow\ c$
and $eventually\ (\lambda x.\ f\ x \neq b)\ (at\ a)$
shows $(\lambda x.\ g\ (f\ x))\ -a\rightarrow\ c$
⟨proof⟩

lemma *tendsto-compose-filtermap*: $((g \circ f) \longrightarrow T)\ F \longleftrightarrow (g \longrightarrow T)\ (filtermap\ f\ F)$
⟨proof⟩

lemma *tendsto-compose-at*:
assumes f: $(f \longrightarrow y)\ F$ **and** g: $(g \longrightarrow z)\ (at\ y)$ **and** fg: $eventually\ (\lambda w.\ f\ w = y \longrightarrow g\ y = z)\ F$
shows $((g \circ f) \longrightarrow z)\ F$
⟨proof⟩

97.8.1 Relation of *LIM* and *LIMSEQ*

lemma (**in** *first-countable-topology*) *sequentially-imp-eventually-within*:
$(\forall f.\ (\forall n.\ f\ n \in s \wedge f\ n \neq a) \wedge f \longrightarrow a \longrightarrow eventually\ (\lambda n.\ P\ (f\ n))\ sequentially) \Longrightarrow$
$eventually\ P\ (at\ a\ within\ s)$
⟨proof⟩

lemma (**in** *first-countable-topology*) *sequentially-imp-eventually-at*:
$(\forall f.\ (\forall n.\ f\ n \neq a) \wedge f \longrightarrow a \longrightarrow eventually\ (\lambda n.\ P\ (f\ n))\ sequentially) \Longrightarrow eventually\ P\ (at\ a)$
⟨proof⟩

lemma *LIMSEQ-SEQ-conv1*:

fixes f :: $'a$::*topological-space* $\Rightarrow$ $'b$::*topological-space*
assumes f: f $-a\rightarrow$ l
shows $\forall S.\ (\forall n.\ S\ n \neq a) \wedge S \longrightarrow a \longrightarrow (\lambda n.\ f\ (S\ n)) \longrightarrow l$
⟨*proof*⟩

lemma *LIMSEQ-SEQ-conv2*:
fixes f :: $'a$::*first-countable-topology* $\Rightarrow$ $'b$::*topological-space*
assumes $\forall S.\ (\forall n.\ S\ n \neq a) \wedge S \longrightarrow a \longrightarrow (\lambda n.\ f\ (S\ n)) \longrightarrow l$
shows f $-a\rightarrow$ l
⟨*proof*⟩

lemma *LIMSEQ-SEQ-conv*: $(\forall S.\ (\forall n.\ S\ n \neq a) \wedge S \longrightarrow a \longrightarrow (\lambda n.\ X\ (S\ n)) \longrightarrow L) \longleftrightarrow X\ -a\rightarrow\ L$
for a :: $'a$::*first-countable-topology* **and** L :: $'b$::*topological-space*
⟨*proof*⟩

lemma *sequentially-imp-eventually-at-left*:
fixes a :: $'a$::{*linorder-topology*,*first-countable-topology*}
assumes $b[simp]$: $b < a$
and $*$: $\bigwedge f.\ (\bigwedge n.\ b < f\ n) \Longrightarrow (\bigwedge n.\ f\ n < a) \Longrightarrow incseq\ f \Longrightarrow f \longrightarrow a \Longrightarrow$
eventually $(\lambda n.\ P\ (f\ n))$ *sequentially*
shows *eventually* P (*at-left* a)
⟨*proof*⟩

lemma *tendsto-at-left-sequentially*:
fixes $a\ b$ :: $'b$::{*linorder-topology*,*first-countable-topology*}
assumes $b < a$
assumes $*$: $\bigwedge S.\ (\bigwedge n.\ S\ n < a) \Longrightarrow (\bigwedge n.\ b < S\ n) \Longrightarrow incseq\ S \Longrightarrow S \longrightarrow a \Longrightarrow$
$(\lambda n.\ X\ (S\ n)) \longrightarrow L$
shows $(X \longrightarrow L)$ (*at-left* a)
⟨*proof*⟩

lemma *sequentially-imp-eventually-at-right*:
fixes $a\ b$ :: $'a$::{*linorder-topology*,*first-countable-topology*}
assumes $b[simp]$: $a < b$
assumes $*$: $\bigwedge f.\ (\bigwedge n.\ a < f\ n) \Longrightarrow (\bigwedge n.\ f\ n < b) \Longrightarrow decseq\ f \Longrightarrow f \longrightarrow a \Longrightarrow$
eventually $(\lambda n.\ P\ (f\ n))$ *sequentially*
shows *eventually* P (*at-right* a)
⟨*proof*⟩

lemma *tendsto-at-right-sequentially*:
fixes a :: - :: {*linorder-topology*, *first-countable-topology*}
assumes $a < b$
and $*$: $\bigwedge S.\ (\bigwedge n.\ a < S\ n) \Longrightarrow (\bigwedge n.\ S\ n < b) \Longrightarrow decseq\ S \Longrightarrow S \longrightarrow a \Longrightarrow$
$(\lambda n.\ X\ (S\ n)) \longrightarrow L$
shows $(X \longrightarrow L)$ (*at-right* a)

⟨*proof*⟩

97.9 Continuity

97.9.1 Continuity on a set

definition *continuous-on* :: $'a$ *set* $\Rightarrow$ ($'a$::*topological-space* $\Rightarrow$ $'b$::*topological-space*) $\Rightarrow$ *bool*
where *continuous-on s f* $\longleftrightarrow$ ($\forall x{\in}s.$ ($f \longrightarrow f\ x$) (*at x within s*))

lemma *continuous-on-cong* [*cong*]:
$s = t \Longrightarrow (\bigwedge x.\ x \in t \Longrightarrow f\ x = g\ x) \Longrightarrow$ *continuous-on s f* $\longleftrightarrow$ *continuous-on t g*
⟨*proof*⟩

lemma *continuous-on-strong-cong*:
$s = t \Longrightarrow (\bigwedge x.\ x \in t$ =*simp*=> $f\ x = g\ x) \Longrightarrow$ *continuous-on s f* $\longleftrightarrow$ *continuous-on t g*
⟨*proof*⟩

lemma *continuous-on-topological*:
continuous-on s f $\longleftrightarrow$
($\forall x{\in}s.\ \forall B.$ *open* $B \longrightarrow f\ x \in B \longrightarrow (\exists A.$ *open* $A \wedge x \in A \wedge (\forall y{\in}s.\ y \in A \longrightarrow f\ y \in B)))$
⟨*proof*⟩

lemma *continuous-on-open-invariant*:
continuous-on s f $\longleftrightarrow$ ($\forall B.$ *open* $B \longrightarrow (\exists A.$ *open* $A \wedge A \cap s = f$ −' $B \cap s$))
⟨*proof*⟩

lemma *continuous-on-open-vimage*:
open s $\Longrightarrow$ *continuous-on s f* $\longleftrightarrow$ ($\forall B.$ *open* $B \longrightarrow$ *open* (f −' $B \cap s$))
⟨*proof*⟩

corollary *continuous-imp-open-vimage*:
assumes *continuous-on s f open s open B f* −' $B \subseteq s$
shows *open* (f −' B)
⟨*proof*⟩

corollary *open-vimage*[*continuous-intros*]:
assumes *open s*
and *continuous-on UNIV f*
shows *open* (f −' s)
⟨*proof*⟩

lemma *continuous-on-closed-invariant*:
continuous-on s f $\longleftrightarrow$ ($\forall B.$ *closed* $B \longrightarrow (\exists A.$ *closed* $A \wedge A \cap s = f$ −' $B \cap s$))
⟨*proof*⟩

lemma *continuous-on-closed-vimage*:
 $closed\ s \Longrightarrow continuous\text{-}on\ s\ f \longleftrightarrow (\forall B.\ closed\ B \longrightarrow closed\ (f\ -\text{`}\ B \cap s))$
 ⟨*proof*⟩

corollary *closed-vimage-Int*[*continuous-intros*]:
 assumes *closed s*
  and *continuous-on t f*
  and *t*: *closed t*
 shows $closed\ (f\ -\text{`}\ s \cap t)$
 ⟨*proof*⟩

corollary *closed-vimage*[*continuous-intros*]:
 assumes *closed s*
  and *continuous-on UNIV f*
 shows $closed\ (f\ -\text{`}\ s)$
 ⟨*proof*⟩

lemma *continuous-on-empty* [*simp*]: *continuous-on* {} *f*
 ⟨*proof*⟩

lemma *continuous-on-sing* [*simp*]: *continuous-on* $\{x\}$ *f*
 ⟨*proof*⟩

lemma *continuous-on-open-Union*:
 $(\bigwedge s.\ s \in S \Longrightarrow open\ s) \Longrightarrow (\bigwedge s.\ s \in S \Longrightarrow continuous\text{-}on\ s\ f) \Longrightarrow continuous\text{-}on$
$(\bigcup S)\ f$
 ⟨*proof*⟩

lemma *continuous-on-open-UN*:
 $(\bigwedge s.\ s \in S \Longrightarrow open\ (A\ s)) \Longrightarrow (\bigwedge s.\ s \in S \Longrightarrow continuous\text{-}on\ (A\ s)\ f) \Longrightarrow$
  $continuous\text{-}on\ (\bigcup s{\in}S.\ A\ s)\ f$
 ⟨*proof*⟩

lemma *continuous-on-open-Un*:
 $open\ s \Longrightarrow open\ t \Longrightarrow continuous\text{-}on\ s\ f \Longrightarrow continuous\text{-}on\ t\ f \Longrightarrow continuous\text{-}on$
$(s \cup t)\ f$
 ⟨*proof*⟩

lemma *continuous-on-closed-Un*:
 $closed\ s \Longrightarrow closed\ t \Longrightarrow continuous\text{-}on\ s\ f \Longrightarrow continuous\text{-}on\ t\ f \Longrightarrow continuous\text{-}on$
$(s \cup t)\ f$
 ⟨*proof*⟩

lemma *continuous-on-If*:
 assumes *closed*: *closed s closed t*
  and *cont*: *continuous-on s f continuous-on t g*
  and *P*: $\bigwedge x.\ x \in s \Longrightarrow \neg\ P\ x \Longrightarrow f\ x = g\ x\ \bigwedge x.\ x \in t \Longrightarrow P\ x \Longrightarrow f\ x = g\ x$
 shows $continuous\text{-}on\ (s \cup t)\ (\lambda x.\ if\ P\ x\ then\ f\ x\ else\ g\ x)$
  (**is** *continuous-on - ?h*)

⟨*proof*⟩

lemma *continuous-on-cases*:
$closed\ s \Longrightarrow closed\ t \Longrightarrow continuous\text{-}on\ s\ f \Longrightarrow continuous\text{-}on\ t\ g \Longrightarrow$
$\forall x.\ (x{\in}s \wedge \neg\ P\ x) \vee (x \in t \wedge P\ x) \longrightarrow f\ x = g\ x \Longrightarrow$
$continuous\text{-}on\ (s \cup t)\ (\lambda x.\ if\ P\ x\ then\ f\ x\ else\ g\ x)$
⟨*proof*⟩

lemma *continuous-on-id*[*continuous-intros*]: $continuous\text{-}on\ s\ (\lambda x.\ x)$
⟨*proof*⟩

lemma *continuous-on-id'*[*continuous-intros*]: *continuous-on s id*
⟨*proof*⟩

lemma *continuous-on-const*[*continuous-intros*]: $continuous\text{-}on\ s\ (\lambda x.\ c)$
⟨*proof*⟩

lemma *continuous-on-subset*: $continuous\text{-}on\ s\ f \Longrightarrow t \subseteq s \Longrightarrow continuous\text{-}on\ t\ f$
⟨*proof*⟩

lemma *continuous-on-compose*[*continuous-intros*]:
$continuous\text{-}on\ s\ f \Longrightarrow continuous\text{-}on\ (f\ `\ s)\ g \Longrightarrow continuous\text{-}on\ s\ (g \circ f)$
⟨*proof*⟩

lemma *continuous-on-compose2*:
$continuous\text{-}on\ t\ g \Longrightarrow continuous\text{-}on\ s\ f \Longrightarrow f\ `\ s \subseteq t \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *continuous-on-generate-topology*:
assumes $*$: $open = generate\text{-}topology\ X$
and $**$: $\bigwedge B.\ B \in X \Longrightarrow \exists\, C.\ open\ C \wedge C \cap A = f\ -`\ B \cap A$
shows *continuous-on A f*
⟨*proof*⟩

lemma *continuous-onI-mono*:
fixes f :: $'a$::*linorder-topology* ⇒ $'b$::{*dense-order*,*linorder-topology*}
assumes $open\ (f`A)$
and *mono*: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \le y \Longrightarrow f\ x \le f\ y$
shows *continuous-on A f*
⟨*proof*⟩

lemma *continuous-on-IccI*:
$[\![(f \longrightarrow f\ a)\ (at\text{-}right\ a);$
$(f \longrightarrow f\ b)\ (at\text{-}left\ b);$
$(\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow f\ {-x\rightarrow}\ f\ x);\ a < b]\!] \Longrightarrow$
$continuous\text{-}on\ \{a\ ..\ b\}\ f$
for a::$'a$::*linorder-topology*
⟨*proof*⟩

lemma
fixes *a b*::*'a*::*linorder-topology*
assumes *continuous-on* $\{a .. b\}$ *f* $a < b$
shows *continuous-on-Icc-at-rightD*: $(f \longrightarrow f\ a)\ (at\text{-}right\ a)$
and *continuous-on-Icc-at-leftD*: $(f \longrightarrow f\ b)\ (at\text{-}left\ b)$
⟨*proof*⟩

97.9.2 Continuity at a point

definition *continuous* :: *'a*::*t2-space filter* ⇒ (*'a* ⇒ *'b*::*topological-space*) ⇒ *bool*
where *continuous F f* $\longleftrightarrow$ $(f \longrightarrow f\ (Lim\ F\ (\lambda x.\ x)))\ F$

lemma *continuous-bot*[*continuous-intros*, *simp*]: *continuous bot f*
⟨*proof*⟩

lemma *continuous-trivial-limit*: *trivial-limit net* $\Longrightarrow$ *continuous net f*
⟨*proof*⟩

lemma *continuous-within*: *continuous* (*at x within s*) *f* $\longleftrightarrow$ $(f \longrightarrow f\ x)$ (*at x within s*)
⟨*proof*⟩

lemma *continuous-within-topological*:
continuous (*at x within s*) *f* $\longleftrightarrow$
$(\forall B.\ open\ B \longrightarrow f\,x \in B \longrightarrow (\exists A.\ open\ A \wedge x \in A \wedge (\forall y \in s.\ y \in A \longrightarrow f\,y \in B)))$
⟨*proof*⟩

lemma *continuous-within-compose*[*continuous-intros*]:
continuous (*at x within s*) *f* $\Longrightarrow$ *continuous* (*at* (*f x*) *within f ‘ s*) *g* $\Longrightarrow$
continuous (*at x within s*) $(g \circ f)$
⟨*proof*⟩

lemma *continuous-within-compose2*:
continuous (*at x within s*) *f* $\Longrightarrow$ *continuous* (*at* (*f x*) *within f ‘ s*) *g* $\Longrightarrow$
continuous (*at x within s*) $(\lambda x.\ g\ (f\,x))$
⟨*proof*⟩

lemma *continuous-at*: *continuous* (*at x*) *f* $\longleftrightarrow$ $f\ {-x\rightarrow}\ f\,x$
⟨*proof*⟩

lemma *continuous-ident*[*continuous-intros*, *simp*]: *continuous* (*at x within S*) $(\lambda x.\ x)$
⟨*proof*⟩

lemma *continuous-const*[*continuous-intros*, *simp*]: *continuous F* $(\lambda x.\ c)$
⟨*proof*⟩

lemma *continuous-on-eq-continuous-within*:
  continuous-on s f $\longleftrightarrow$ ($\forall x \in s$. *continuous* (*at x within s*) *f*)
  ⟨*proof*⟩

abbreviation *isCont* :: (*'a*::*t2-space* $\Rightarrow$ *'b*::*topological-space*) $\Rightarrow$ *'a* $\Rightarrow$ *bool*
  where *isCont f a* $\equiv$ *continuous* (*at a*) *f*

lemma *isCont-def*: *isCont f a* $\longleftrightarrow$ *f* $-a\rightarrow$ *f a*
  ⟨*proof*⟩

lemma *isCont-cong*:
  assumes *eventually* (λx. *f x* = *g x*) (*nhds x*)
  shows *isCont f x* $\longleftrightarrow$ *isCont g x*
⟨*proof*⟩

lemma *continuous-at-imp-continuous-at-within*: *isCont f x* $\Longrightarrow$ *continuous* (*at x within s*) *f*
  ⟨*proof*⟩

lemma *continuous-on-eq-continuous-at*: *open s* $\Longrightarrow$ *continuous-on s f* $\longleftrightarrow$ ($\forall x \in s$. *isCont f x*)
  ⟨*proof*⟩

lemma *continuous-within-open*: $a \in A \Longrightarrow$ *open A* $\Longrightarrow$ *continuous* (*at a within A*) *f* $\longleftrightarrow$ *isCont f a*
  ⟨*proof*⟩

lemma *continuous-at-imp-continuous-on*: $\forall x \in s$. *isCont f x* $\Longrightarrow$ *continuous-on s f*
  ⟨*proof*⟩

lemma *isCont-o2*: *isCont f a* $\Longrightarrow$ *isCont g* (*f a*) $\Longrightarrow$ *isCont* (λx. *g* (*f x*)) *a*
  ⟨*proof*⟩

lemma *isCont-o*[*continuous-intros*]: *isCont f a* $\Longrightarrow$ *isCont g* (*f a*) $\Longrightarrow$ *isCont* ($g \circ f$) *a*
  ⟨*proof*⟩

lemma *isCont-tendsto-compose*: *isCont g l* $\Longrightarrow$ ($f \longrightarrow l$) *F* $\Longrightarrow$ ((λx. *g* (*f x*)) $\longrightarrow$ *g l*) *F*
  ⟨*proof*⟩

lemma *continuous-on-tendsto-compose*:
  assumes *f-cont*: *continuous-on s f*
    and *g*: ($g \longrightarrow l$) *F*
    and *l*: $l \in s$
    and *ev*: $\forall_F x$ *in F*. $g\ x \in s$
  shows ((λx. *f* (*g x*)) $\longrightarrow$ *f l*) *F*
⟨*proof*⟩

lemma *continuous-within-compose3*:
$isCont\ g\ (f\ x) \Longrightarrow continuous\ (at\ x\ within\ s)\ f \Longrightarrow continuous\ (at\ x\ within\ s)\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *filtermap-nhds-open-map*:
assumes *cont*: $isCont\ f\ a$
and *open-map*: $\bigwedge S.\ open\ S \Longrightarrow open\ (f`S)$
shows $filtermap\ f\ (nhds\ a) = nhds\ (f\ a)$
⟨*proof*⟩

lemma *continuous-at-split*:
$continuous\ (at\ x)\ f \longleftrightarrow continuous\ (at\text{-}left\ x)\ f \wedge continuous\ (at\text{-}right\ x)\ f$
for $x :: 'a::linorder\text{-}topology$
⟨*proof*⟩

The following open/closed Collect lemmas are ported from Sébastien Gouëzel's *Ergodic-Theory*.

lemma *open-Collect-neq*:
fixes $f\ g :: 'a::topological\text{-}space \Rightarrow 'b::t2\text{-}space$
assumes f: *continuous-on UNIV f* **and** g: *continuous-on UNIV g*
shows $open\ \{x.\ f\ x \neq g\ x\}$
⟨*proof*⟩

lemma *closed-Collect-eq*:
fixes $f\ g :: 'a::topological\text{-}space \Rightarrow 'b::t2\text{-}space$
assumes f: *continuous-on UNIV f* **and** g: *continuous-on UNIV g*
shows $closed\ \{x.\ f\ x = g\ x\}$
⟨*proof*⟩

lemma *open-Collect-less*:
fixes $f\ g :: 'a::topological\text{-}space \Rightarrow 'b::linorder\text{-}topology$
assumes f: *continuous-on UNIV f* **and** g: *continuous-on UNIV g*
shows $open\ \{x.\ f\ x < g\ x\}$
⟨*proof*⟩

lemma *closed-Collect-le*:
fixes $f\ g :: 'a :: topological\text{-}space \Rightarrow 'b::linorder\text{-}topology$
assumes f: *continuous-on UNIV f*
and g: *continuous-on UNIV g*
shows $closed\ \{x.\ f\ x \leq g\ x\}$
⟨*proof*⟩

97.9.3 Open-cover compactness

context *topological-space*
begin

definition $compact :: 'a\ set \Rightarrow bool$

where *compact-eq-heine-borel*:
$compact\ S \longleftrightarrow (\forall\, C.\ (\forall\, c{\in}C.\ open\ c) \wedge S \subseteq \bigcup C \longrightarrow (\exists\, D{\subseteq}C.\ finite\ D \wedge S \subseteq \bigcup D))$

lemma *compactI*:
assumes $\bigwedge C.\ \forall\, t{\in}C.\ open\ t \Longrightarrow s \subseteq \bigcup C \Longrightarrow \exists\, C'.\ C' \subseteq C \wedge finite\ C' \wedge s \subseteq \bigcup C'$
shows *compact s*
⟨*proof*⟩

lemma *compact-empty*[*simp*]: *compact* {}
⟨*proof*⟩

lemma *compactE*:
assumes $compact\ S\ S \subseteq \bigcup \mathcal{T}\ \bigwedge B.\ B \in \mathcal{T} \Longrightarrow open\ B$
obtains $\mathcal{T}'$ **where** $\mathcal{T}' \subseteq \mathcal{T}\ finite\ \mathcal{T}'\ S \subseteq \bigcup \mathcal{T}'$
⟨*proof*⟩

lemma *compactE-image*:
assumes *compact S*
and *op*: $\bigwedge T.\ T \in C \Longrightarrow open\ (f\ T)$
and *S*: $S \subseteq (\bigcup c{\in}C.\ f\ c)$
obtains C' **where** $C' \subseteq C$ **and** *finite* C' **and** $S \subseteq (\bigcup c{\in}C'.\ f\ c)$
⟨*proof*⟩

lemma *compact-Int-closed* [*intro*]:
assumes *compact S*
and *closed T*
shows $compact\ (S \cap T)$
⟨*proof*⟩

lemma *compact-diff*: $[\![compact\ S;\ open\ T]\!] \Longrightarrow compact(S - T)$
⟨*proof*⟩

lemma *inj-setminus*: *inj-on uminus* $(A{::}'a\ set\ set)$
⟨*proof*⟩

97.10 Finite intersection property

lemma *compact-fip*:
$compact\ U \longleftrightarrow$
$(\forall\, A.\ (\forall\, a{\in}A.\ closed\ a) \longrightarrow (\forall\, B \subseteq A.\ finite\ B \longrightarrow U \cap \bigcap B \neq \{\}) \longrightarrow U \cap \bigcap A \neq \{\})$
(**is** - $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *compact-imp-fip*:
assumes *compact S*
and $\bigwedge T.\ T \in F \Longrightarrow closed\ T$

and $\bigwedge F'.$ *finite* $F' \Longrightarrow F' \subseteq F \Longrightarrow S \cap (\bigcap F') \neq \{\}$
shows $S \cap (\bigcap F) \neq \{\}$
⟨*proof*⟩

lemma *compact-imp-fip-image*:
assumes *compact s*
and P: $\bigwedge i.\ i \in I \Longrightarrow$ *closed* $(f\ i)$
and Q: $\bigwedge I'.$ *finite* $I' \Longrightarrow I' \subseteq I \Longrightarrow (s \cap (\bigcap i{\in}I'.\ f\ i) \neq \{\})$
shows $s \cap (\bigcap i{\in}I.\ f\ i) \neq \{\}$
⟨*proof*⟩

end

lemma (**in** *t2-space*) *compact-imp-closed*:
assumes *compact s*
shows *closed s*
⟨*proof*⟩

lemma *compact-continuous-image*:
assumes f: *continuous-on s f*
and s: *compact s*
shows *compact* $(f\ `\ s)$
⟨*proof*⟩

lemma *continuous-on-inv*:
fixes f :: *'a::topological-space* ⇒ *'b::t2-space*
assumes *continuous-on s f*
and *compact s*
and $\forall x{\in}s.\ g\ (f\ x) = x$
shows *continuous-on* $(f\ `\ s)\ g$
⟨*proof*⟩

lemma *continuous-on-inv-into*:
fixes f :: *'a::topological-space* ⇒ *'b::t2-space*
assumes s: *continuous-on s f compact s*
and f: *inj-on f s*
shows *continuous-on* $(f\ `\ s)$ (*the-inv-into s f*)
⟨*proof*⟩

lemma (**in** *linorder-topology*) *compact-attains-sup*:
assumes *compact S* $S \neq \{\}$
shows $\exists s{\in}S.\ \forall t{\in}S.\ t \leq s$
⟨*proof*⟩

lemma (**in** *linorder-topology*) *compact-attains-inf*:
assumes *compact S* $S \neq \{\}$
shows $\exists s{\in}S.\ \forall t{\in}S.\ s \leq t$
⟨*proof*⟩

lemma *continuous-attains-sup*:
fixes f :: $'a$::*topological-space* $\Rightarrow$ $'b$::*linorder-topology*
shows *compact* $s \Longrightarrow s \neq \{\} \Longrightarrow$ *continuous-on* $s\ f \Longrightarrow (\exists x{\in}s.\ \forall y{\in}s.\ \ f\ y \leq f\ x)$
⟨*proof*⟩

lemma *continuous-attains-inf*:
fixes f :: $'a$::*topological-space* $\Rightarrow$ $'b$::*linorder-topology*
shows *compact* $s \Longrightarrow s \neq \{\} \Longrightarrow$ *continuous-on* $s\ f \Longrightarrow (\exists x{\in}s.\ \forall y{\in}s.\ f\ x \leq f\ y)$
⟨*proof*⟩

97.11 Connectedness

context *topological-space*
begin

definition *connected* $S \longleftrightarrow$
$\neg (\exists A\ B.\ open\ A \wedge open\ B \wedge S \subseteq A \cup B \wedge A \cap B \cap S = \{\} \wedge A \cap S \neq \{\} \wedge B \cap S \neq \{\})$

lemma *connectedI*:
$(\bigwedge A\ B.\ open\ A \Longrightarrow open\ B \Longrightarrow A \cap U \neq \{\} \Longrightarrow B \cap U \neq \{\} \Longrightarrow A \cap B \cap U = \{\} \Longrightarrow U \subseteq A \cup B \Longrightarrow False)$
$\Longrightarrow$ *connected* U
⟨*proof*⟩

lemma *connected-empty* [*simp*]: *connected* $\{\}$
⟨*proof*⟩

lemma *connected-sing* [*simp*]: *connected* $\{x\}$
⟨*proof*⟩

lemma *connectedD*:
connected $A \Longrightarrow$ *open* $U \Longrightarrow$ *open* $V \Longrightarrow U \cap V \cap A = \{\} \Longrightarrow A \subseteq U \cup V \Longrightarrow U \cap A = \{\} \vee V \cap A = \{\}$
⟨*proof*⟩

end

lemma *connected-closed*:
connected $s \longleftrightarrow$
$\neg (\exists A\ B.\ closed\ A \wedge closed\ B \wedge s \subseteq A \cup B \wedge A \cap B \cap s = \{\} \wedge A \cap s \neq \{\} \wedge B \cap s \neq \{\})$
⟨*proof*⟩

lemma *connected-closedD*:
$[\![connected\ s;\ A \cap B \cap s = \{\};\ s \subseteq A \cup B;\ closed\ A;\ closed\ B]\!] \Longrightarrow A \cap s = \{\} \vee B \cap s = \{\}$

⟨proof⟩

lemma *connected-Union*:
assumes *cs*: $\bigwedge s.\ s \in S \Longrightarrow connected\ s$
and *ne*: $\bigcap S \neq \{\}$
shows $connected(\bigcup S)$
⟨proof⟩

lemma *connected-Un*: $connected\ s \Longrightarrow connected\ t \Longrightarrow s \cap t \neq \{\} \Longrightarrow connected\ (s \cup t)$
⟨proof⟩

lemma *connected-diff-open-from-closed*:
assumes *st*: $s \subseteq t$
and *tu*: $t \subseteq u$
and *s*: *open s*
and *t*: *closed t*
and *u*: *connected u*
and *ts*: $connected\ (t - s)$
shows $connected(u - s)$
⟨proof⟩

lemma *connected-iff-const*:
fixes $S :: 'a::topological\text{-}space\ set$
shows $connected\ S \longleftrightarrow (\forall P::'a \Rightarrow bool.\ continuous\text{-}on\ S\ P \longrightarrow (\exists c.\ \forall s\in S.\ P\ s = c))$
⟨proof⟩

lemma *connectedD-const*: $connected\ S \Longrightarrow continuous\text{-}on\ S\ P \Longrightarrow \exists c.\ \forall s\in S.\ P\ s = c$
for $P :: 'a::topological\text{-}space \Rightarrow bool$
⟨proof⟩

lemma *connectedI-const*:
$(\bigwedge P::'a::topological\text{-}space \Rightarrow bool.\ continuous\text{-}on\ S\ P \Longrightarrow \exists c.\ \forall s\in S.\ P\ s = c) \Longrightarrow connected\ S$
⟨proof⟩

lemma *connected-local-const*:
assumes $connected\ A\ \ a \in A\ \ b \in A$
and $*$: $\forall a\in A.\ eventually\ (\lambda b.\ f\ a = f\ b)\ (at\ a\ within\ A)$
shows $f\ a = f\ b$
⟨proof⟩

lemma (**in** *linorder-topology*) *connectedD-interval*:
assumes *connected U*
and *xy*: $x \in U\ y \in U$
and $x \leq z\ z \leq y$
shows $z \in U$

⟨*proof*⟩

lemma *connected-continuous-image*:
  assumes $*$: *continuous-on s f*
    and *connected s*
  shows *connected* $(f \text{ ` } s)$
⟨*proof*⟩

98 Linear Continuum Topologies

class *linear-continuum-topology* = *linorder-topology* + *linear-continuum*
begin

lemma *Inf-notin-open*:
  assumes *A*: *open A*
    and *bnd*: $\forall\, a{\in}A.\ x < a$
  shows *Inf* $A \notin A$
⟨*proof*⟩

lemma *Sup-notin-open*:
  assumes *A*: *open A*
    and *bnd*: $\forall\, a{\in}A.\ a < x$
  shows *Sup* $A \notin A$
⟨*proof*⟩

end

instance *linear-continuum-topology* $\subseteq$ *perfect-space*
⟨*proof*⟩

lemma *connectedI-interval*:
  fixes U :: $'a$ :: *linear-continuum-topology set*
  assumes $*$: $\bigwedge x\ y\ z.\ x \in U \Longrightarrow y \in U \Longrightarrow x \le z \Longrightarrow z \le y \Longrightarrow z \in U$
  shows *connected U*
⟨*proof*⟩

lemma *connected-iff-interval*: *connected U* $\longleftrightarrow$ $(\forall\, x{\in}U.\ \forall\, y{\in}U.\ \forall\, z.\ x \le z \longrightarrow z$
$\le y \longrightarrow z \in U)$
  for U :: $'a$::*linear-continuum-topology set*
  ⟨*proof*⟩

lemma *connected-UNIV*[*simp*]: *connected* (*UNIV*::$'a$::*linear-continuum-topology set*)
  ⟨*proof*⟩

lemma *connected-Ioi*[*simp*]: *connected* $\{a{<}..\}$
  for a :: $'a$::*linear-continuum-topology*
  ⟨*proof*⟩

lemma *connected-Ici*[*simp*]: *connected* $\{a..\}$

for a :: $'a$::*linear-continuum-topology*
⟨*proof*⟩

lemma *connected-Iio*[*simp*]: *connected* $\{..<a\}$
for a :: $'a$::*linear-continuum-topology*
⟨*proof*⟩

lemma *connected-Iic*[*simp*]: *connected* $\{..a\}$
for a :: $'a$::*linear-continuum-topology*
⟨*proof*⟩

lemma *connected-Ioo*[*simp*]: *connected* $\{a<..<b\}$
for a b :: $'a$::*linear-continuum-topology*
⟨*proof*⟩

lemma *connected-Ioc*[*simp*]: *connected* $\{a<..b\}$
for a b :: $'a$::*linear-continuum-topology*
⟨*proof*⟩

lemma *connected-Ico*[*simp*]: *connected* $\{a..<b\}$
for a b :: $'a$::*linear-continuum-topology*
⟨*proof*⟩

lemma *connected-Icc*[*simp*]: *connected* $\{a..b\}$
for a b :: $'a$::*linear-continuum-topology*
⟨*proof*⟩

lemma *connected-contains-Ioo*:
fixes A :: $'a$ :: *linorder-topology set*
assumes *connected* A $a \in A$ $b \in A$ **shows** $\{a <..< b\} \subseteq A$
⟨*proof*⟩

lemma *connected-contains-Icc*:
fixes A :: $'a$::*linorder-topology set*
assumes *connected* A $a \in A$ $b \in A$
shows $\{a..b\} \subseteq A$
⟨*proof*⟩

98.1 Intermediate Value Theorem

lemma *IVT'*:
fixes f :: $'a$::*linear-continuum-topology* $\Rightarrow$ $'b$::*linorder-topology*
assumes y: $f\,a \leq y$ $y \leq f\,b$ $a \leq b$
and $*$: *continuous-on* $\{a\ ..\ b\}$ f
shows $\exists\,x.\ a \leq x \wedge x \leq b \wedge f\,x = y$
⟨*proof*⟩

lemma *IVT2'*:
fixes f :: $'a$ :: *linear-continuum-topology* $\Rightarrow$ $'b$ :: *linorder-topology*

assumes y: $f\ b \le y\ y \le f\ a\ a \le b$
and $*$: *continuous-on* $\{a\ ..\ b\}\ f$
shows $\exists\, x.\ a \le x \land x \le b \land f\ x = y$
⟨*proof*⟩

lemma *IVT*:
fixes f :: $'a$::*linear-continuum-topology* $\Rightarrow$ $'b$::*linorder-topology*
shows $f\ a \le y \Longrightarrow y \le f\ b \Longrightarrow a \le b \Longrightarrow (\forall\, x.\ a \le x \land x \le b \longrightarrow isCont\ f\ x) \Longrightarrow$
$\exists\, x.\ a \le x \land x \le b \land f\ x = y$
⟨*proof*⟩

lemma *IVT2*:
fixes f :: $'a$::*linear-continuum-topology* $\Rightarrow$ $'b$::*linorder-topology*
shows $f\ b \le y \Longrightarrow y \le f\ a \Longrightarrow a \le b \Longrightarrow (\forall\, x.\ a \le x \land x \le b \longrightarrow isCont\ f\ x) \Longrightarrow$
$\exists\, x.\ a \le x \land x \le b \land f\ x = y$
⟨*proof*⟩

lemma *continuous-inj-imp-mono*:
fixes f :: $'a$::*linear-continuum-topology* $\Rightarrow$ $'b$::*linorder-topology*
assumes x: $a < x\ x < b$
and *cont*: *continuous-on* $\{a..b\}\ f$
and *inj*: *inj-on* $f\ \{a..b\}$
shows $(f\ a < f\ x \land f\ x < f\ b) \lor (f\ b < f\ x \land f\ x < f\ a)$
⟨*proof*⟩

lemma *continuous-at-Sup-mono*:
fixes f :: $'a$::{*linorder-topology*,*conditionally-complete-linorder*} $\Rightarrow$
$'b$::{*linorder-topology*,*conditionally-complete-linorder*}
assumes *mono* f
and *cont*: *continuous* (*at-left* (*Sup* S)) f
and S: $S \ne \{\}$ *bdd-above* S
shows f (*Sup* S) = (*SUP* s:S. $f\ s$)
⟨*proof*⟩

lemma *continuous-at-Sup-antimono*:
fixes f :: $'a$::{*linorder-topology*,*conditionally-complete-linorder*} $\Rightarrow$
$'b$::{*linorder-topology*,*conditionally-complete-linorder*}
assumes *antimono* f
and *cont*: *continuous* (*at-left* (*Sup* S)) f
and S: $S \ne \{\}$ *bdd-above* S
shows f (*Sup* S) = (*INF* s:S. $f\ s$)
⟨*proof*⟩

lemma *continuous-at-Inf-mono*:
fixes f :: $'a$::{*linorder-topology*,*conditionally-complete-linorder*} $\Rightarrow$
$'b$::{*linorder-topology*,*conditionally-complete-linorder*}
assumes *mono* f

and *cont*: *continuous* (*at-right* (*Inf S*)) *f*
and *S*: $S \neq \{\}$ *bdd-below S*
shows *f* (*Inf S*) = (*INF s*:*S*. *f s*)
⟨*proof*⟩

lemma *continuous-at-Inf-antimono*:
fixes *f* :: *'a*::{*linorder-topology*,*conditionally-complete-linorder*} ⇒ *'b*::{*linorder-topology*,*conditionally-complete-linorder*}
assumes *antimono f*
and *cont*: *continuous* (*at-right* (*Inf S*)) *f*
and *S*: $S \neq \{\}$ *bdd-below S*
shows *f* (*Inf S*) = (*SUP s*:*S*. *f s*)
⟨*proof*⟩

98.2 Uniform spaces

class *uniformity* =
fixes *uniformity* :: (*'a* × *'a*) *filter*
begin

abbreviation *uniformity-on* :: *'a set* ⇒ (*'a* × *'a*) *filter*
where *uniformity-on s* ≡ *inf uniformity* (*principal* (*s*×*s*))

end

lemma *uniformity-Abort*:
uniformity =
Filter.abstract-filter (λ*u*. *Code.abort* (*STR ''uniformity is not executable''*) (λ*u*. *uniformity*))
⟨*proof*⟩

class *open-uniformity* = *open* + *uniformity* +
assumes *open-uniformity*:
⋀*U*. *open U* ⟷ (∀ *x*∈*U*. *eventually* (λ(*x'*, *y*). *x'* = *x* ⟶ *y* ∈ *U*) *uniformity*)

class *uniform-space* = *open-uniformity* +
assumes *uniformity-refl*: *eventually E uniformity* ⟹ *E* (*x*, *x*)
and *uniformity-sym*: *eventually E uniformity* ⟹ *eventually* (λ(*x*, *y*). *E* (*y*, *x*)) *uniformity*
and *uniformity-trans*:
eventually E uniformity ⟹
∃ *D*. *eventually D uniformity* ∧ (∀ *x y z*. *D* (*x*, *y*) ⟶ *D* (*y*, *z*) ⟶ *E* (*x*, *z*))
begin

subclass *topological-space*
⟨*proof*⟩

lemma *uniformity-bot*: *uniformity* ≠ *bot*

⟨*proof*⟩

lemma *uniformity-trans'*:
eventually E uniformity $\Longrightarrow$
eventually $(\lambda((x, y), (y', z)).\ y = y' \longrightarrow E\ (x, z))$ (*uniformity* $\times_F$ *uniformity*)
⟨*proof*⟩

lemma *uniformity-transE*:
assumes *eventually E uniformity*
obtains *D* **where** *eventually D uniformity* $\bigwedge x\ y\ z.\ D\ (x, y) \Longrightarrow D\ (y, z) \Longrightarrow E\ (x, z)$
⟨*proof*⟩

lemma *eventually-nhds-uniformity*:
eventually P (*nhds x*) $\longleftrightarrow$ *eventually* $(\lambda(x', y).\ x' = x \longrightarrow P\ y)$ *uniformity*
(**is** - $\longleftrightarrow$ *?N P x*)
⟨*proof*⟩

98.2.1 Totally bounded sets

definition *totally-bounded* :: *'a set* $\Rightarrow$ *bool*
where *totally-bounded S* $\longleftrightarrow$
$(\forall E.$ *eventually E uniformity* $\longrightarrow (\exists X.$ *finite X* $\wedge (\forall s{\in}S.\ \exists x{\in}X.\ E\ (x, s))))$

lemma *totally-bounded-empty*[*iff*]: *totally-bounded* {}
⟨*proof*⟩

lemma *totally-bounded-subset*: *totally-bounded S* $\Longrightarrow T \subseteq S \Longrightarrow$ *totally-bounded T*
⟨*proof*⟩

lemma *totally-bounded-Union*[*intro*]:
assumes *M*: *finite M* $\bigwedge S.\ S \in M \Longrightarrow$ *totally-bounded S*
shows *totally-bounded* $(\bigcup M)$
⟨*proof*⟩

98.2.2 Cauchy filter

definition *cauchy-filter* :: *'a filter* $\Rightarrow$ *bool*
where *cauchy-filter F* $\longleftrightarrow F \times_F F \leq$ *uniformity*

definition *Cauchy* :: (*nat* $\Rightarrow$ *'a*) $\Rightarrow$ *bool*
where *Cauchy-uniform*: *Cauchy X* = *cauchy-filter* (*filtermap X sequentially*)

lemma *Cauchy-uniform-iff*:
Cauchy X $\longleftrightarrow (\forall P.$ *eventually P uniformity* $\longrightarrow (\exists N.\ \forall n{\geq}N.\ \forall m{\geq}N.\ P\ (X\ n, X\ m)))$
⟨*proof*⟩

lemma *nhds-imp-cauchy-filter*:

assumes $*$: $F \leq nhds\ x$
shows *cauchy-filter F*
⟨*proof*⟩

lemma *LIMSEQ-imp-Cauchy*: $X \longrightarrow x \Longrightarrow Cauchy\ X$
⟨*proof*⟩

lemma *Cauchy-subseq-Cauchy*:
assumes *Cauchy X strict-mono f*
shows $Cauchy\ (X \circ f)$
⟨*proof*⟩

lemma *convergent-Cauchy*: $convergent\ X \Longrightarrow Cauchy\ X$
⟨*proof*⟩

definition $complete :: 'a\ set \Rightarrow bool$
where *complete-uniform*: $complete\ S \longleftrightarrow$
$(\forall F \leq principal\ S.\ F \neq bot \longrightarrow cauchy\text{-}filter\ F \longrightarrow (\exists x{\in}S.\ F \leq nhds\ x))$

end

98.2.3 Uniformly continuous functions

definition $uniformly\text{-}continuous\text{-}on :: 'a\ set \Rightarrow ('a::uniform\text{-}space \Rightarrow 'b::uniform\text{-}space) \Rightarrow bool$
where *uniformly-continuous-on-uniformity*: $uniformly\text{-}continuous\text{-}on\ s\ f \longleftrightarrow$
$(LIM\ (x,\ y)\ (uniformity\text{-}on\ s).\ (f\ x,\ f\ y) :> uniformity)$

lemma *uniformly-continuous-onD*:
$uniformly\text{-}continuous\text{-}on\ s\ f \Longrightarrow eventually\ E\ uniformity \Longrightarrow$
$eventually\ (\lambda(x,\ y).\ x \in s \longrightarrow y \in s \longrightarrow E\ (f\ x,\ f\ y))\ uniformity$
⟨*proof*⟩

lemma *uniformly-continuous-on-const*[*continuous-intros*]: $uniformly\text{-}continuous\text{-}on\ s\ (\lambda x.\ c)$
⟨*proof*⟩

lemma *uniformly-continuous-on-id*[*continuous-intros*]: $uniformly\text{-}continuous\text{-}on\ s\ (\lambda x.\ x)$
⟨*proof*⟩

lemma *uniformly-continuous-on-compose*[*continuous-intros*]:
$uniformly\text{-}continuous\text{-}on\ s\ g \Longrightarrow uniformly\text{-}continuous\text{-}on\ (g`s)\ f \Longrightarrow$
$uniformly\text{-}continuous\text{-}on\ s\ (\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma *uniformly-continuous-imp-continuous*:
assumes *f*: *uniformly-continuous-on s f*
shows *continuous-on s f*

⟨*proof*⟩

99 Product Topology

99.1 Product is a topological space

instantiation *prod* :: (*topological-space*, *topological-space*) *topological-space*
begin

definition *open-prod-def*[*code del*]:
$open\ (S :: ('a \times 'b)\ set) \longleftrightarrow$
$(\forall x{\in}S.\ \exists A\ B.\ open\ A \wedge open\ B \wedge x \in A \times B \wedge A \times B \subseteq S)$

lemma *open-prod-elim*:
assumes *open* S **and** $x \in S$
obtains A B **where** *open* A **and** *open* B **and** $x \in A \times B$ **and** $A \times B \subseteq S$
⟨*proof*⟩

lemma *open-prod-intro*:
assumes $\bigwedge x.\ x \in S \Longrightarrow \exists A\ B.\ open\ A \wedge open\ B \wedge x \in A \times B \wedge A \times B \subseteq S$
shows *open* S
⟨*proof*⟩

instance
⟨*proof*⟩

end

declare [[*code abort*: $open :: ('a::topological\text{-}space \times 'b::topological\text{-}space)\ set \Rightarrow bool$]]

lemma *open-Times*: $open\ S \Longrightarrow open\ T \Longrightarrow open\ (S \times T)$
⟨*proof*⟩

lemma *fst-vimage-eq-Times*: $fst\ -`\ S = S \times UNIV$
⟨*proof*⟩

lemma *snd-vimage-eq-Times*: $snd\ -`\ S = UNIV \times S$
⟨*proof*⟩

lemma *open-vimage-fst*: $open\ S \Longrightarrow open\ (fst\ -`\ S)$
⟨*proof*⟩

lemma *open-vimage-snd*: $open\ S \Longrightarrow open\ (snd\ -`\ S)$
⟨*proof*⟩

lemma *closed-vimage-fst*: $closed\ S \Longrightarrow closed\ (fst\ -`\ S)$
⟨*proof*⟩

lemma *closed-vimage-snd*: $closed\ S \Longrightarrow closed\ (snd\ -`\ S)$
⟨*proof*⟩

lemma *closed-Times*: $closed\ S \Longrightarrow closed\ T \Longrightarrow closed\ (S \times T)$
⟨*proof*⟩

lemma *subset-fst-imageI*: $A \times B \subseteq S \Longrightarrow y \in B \Longrightarrow A \subseteq fst\ `\ S$
⟨*proof*⟩

lemma *subset-snd-imageI*: $A \times B \subseteq S \Longrightarrow x \in A \Longrightarrow B \subseteq snd\ `\ S$
⟨*proof*⟩

lemma *open-image-fst*:
assumes $open\ S$
shows $open\ (fst\ `\ S)$
⟨*proof*⟩

lemma *open-image-snd*:
assumes $open\ S$
shows $open\ (snd\ `\ S)$
⟨*proof*⟩

lemma *nhds-prod*: $nhds\ (a,\ b) = nhds\ a \times_F nhds\ b$
⟨*proof*⟩

99.1.1 Continuity of operations

lemma *tendsto-fst* [*tendsto-intros*]:
assumes $(f \longrightarrow a)\ F$
shows $((\lambda x.\ fst\ (f\ x)) \longrightarrow fst\ a)\ F$
⟨*proof*⟩

lemma *tendsto-snd* [*tendsto-intros*]:
assumes $(f \longrightarrow a)\ F$
shows $((\lambda x.\ snd\ (f\ x)) \longrightarrow snd\ a)\ F$
⟨*proof*⟩

lemma *tendsto-Pair* [*tendsto-intros*]:
assumes $(f \longrightarrow a)\ F$ **and** $(g \longrightarrow b)\ F$
shows $((\lambda x.\ (f\ x,\ g\ x)) \longrightarrow (a,\ b))\ F$
⟨*proof*⟩

lemma *continuous-fst*[*continuous-intros*]: $continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ fst\ (f\ x))$
⟨*proof*⟩

lemma *continuous-snd*[*continuous-intros*]: $continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ snd\ (f\ x))$
⟨*proof*⟩

lemma *continuous-Pair*[*continuous-intros*]:
continuous F $f \Longrightarrow$ *continuous* F $g \Longrightarrow$ *continuous* F $(\lambda x.\ (f\ x,\ g\ x))$
⟨*proof*⟩

lemma *continuous-on-fst*[*continuous-intros*]:
continuous-on s $f \Longrightarrow$ *continuous-on* s $(\lambda x.\ fst\ (f\ x))$
⟨*proof*⟩

lemma *continuous-on-snd*[*continuous-intros*]:
continuous-on s $f \Longrightarrow$ *continuous-on* s $(\lambda x.\ snd\ (f\ x))$
⟨*proof*⟩

lemma *continuous-on-Pair*[*continuous-intros*]:
continuous-on s $f \Longrightarrow$ *continuous-on* s $g \Longrightarrow$ *continuous-on* s $(\lambda x.\ (f\ x,\ g\ x))$
⟨*proof*⟩

lemma *continuous-on-swap*[*continuous-intros*]: *continuous-on* A *prod.swap*
⟨*proof*⟩

lemma *continuous-on-swap-args*:
assumes *continuous-on* $(A \times B)$ $(\lambda(x,y).\ d\ x\ y)$
shows *continuous-on* $(B \times A)$ $(\lambda(x,y).\ d\ y\ x)$
⟨*proof*⟩

lemma *isCont-fst* [*simp*]: *isCont* f $a \Longrightarrow$ *isCont* $(\lambda x.\ fst\ (f\ x))$ a
⟨*proof*⟩

lemma *isCont-snd* [*simp*]: *isCont* f $a \Longrightarrow$ *isCont* $(\lambda x.\ snd\ (f\ x))$ a
⟨*proof*⟩

lemma *isCont-Pair* [*simp*]: ⟦*isCont* f a; *isCont* g a⟧ $\Longrightarrow$ *isCont* $(\lambda x.\ (f\ x,\ g\ x))$ a
⟨*proof*⟩

99.1.2 Separation axioms

instance *prod* :: (*t0-space*, *t0-space*) *t0-space*
⟨*proof*⟩

instance *prod* :: (*t1-space*, *t1-space*) *t1-space*
⟨*proof*⟩

instance *prod* :: (*t2-space*, *t2-space*) *t2-space*
⟨*proof*⟩

lemma *isCont-swap*[*continuous-intros*]: *isCont prod.swap* a
⟨*proof*⟩

lemma *open-diagonal-complement*:

open $\{(x,y) \mid x\ y.\ x \neq (y::('a::t2\text{-}space))\}$
⟨*proof*⟩

lemma *closed-diagonal*:
closed $\{y.\ \exists\ x::('a::t2\text{-}space).\ y = (x,x)\}$
⟨*proof*⟩

lemma *open-superdiagonal*:
open $\{(x,y) \mid x\ y.\ x > (y::'a::\{linorder\text{-}topology\})\}$
⟨*proof*⟩

lemma *closed-subdiagonal*:
closed $\{(x,y) \mid x\ y.\ x \leq (y::'a::\{linorder\text{-}topology\})\}$
⟨*proof*⟩

lemma *open-subdiagonal*:
open $\{(x,y) \mid x\ y.\ x < (y::'a::\{linorder\text{-}topology\})\}$
⟨*proof*⟩

lemma *closed-superdiagonal*:
closed $\{(x,y) \mid x\ y.\ x \geq (y::('a::\{linorder\text{-}topology\}))\}$
⟨*proof*⟩

end

100 Vector Spaces and Algebras over the Reals

theory *Real-Vector-Spaces*
imports *Real Topological-Spaces*
begin

100.1 Locale for additive functions

locale *additive* =
fixes $f :: 'a::ab\text{-}group\text{-}add \Rightarrow 'b::ab\text{-}group\text{-}add$
assumes *add*: $f\ (x + y) = f\ x + f\ y$
begin

lemma *zero*: $f\ 0 = 0$
⟨*proof*⟩

lemma *minus*: $f\ (-\ x) = -\ f\ x$
⟨*proof*⟩

lemma *diff*: $f\ (x - y) = f\ x - f\ y$
⟨*proof*⟩

lemma *sum*: $f\ (sum\ g\ A) = (\sum x{\in}A.\ f\ (g\ x))$
⟨*proof*⟩

end

100.2 Vector spaces

locale *vector-space* =
fixes *scale* :: $'a$::*field* ⇒ $'b$::*ab-group-add* ⇒ $'b$
assumes *scale-right-distrib* [*algebra-simps*]: *scale a* $(x + y)$ = *scale a x* + *scale a y*
and *scale-left-distrib* [*algebra-simps*]: *scale* $(a + b)$ *x* = *scale a x* + *scale b x*
and *scale-scale* [*simp*]: *scale a* (*scale b x*) = *scale* $(a * b)$ *x*
and *scale-one* [*simp*]: *scale 1 x* = *x*
begin

lemma *scale-left-commute*: *scale a* (*scale b x*) = *scale b* (*scale a x*)
⟨*proof*⟩

lemma *scale-zero-left* [*simp*]: *scale 0 x* = *0*
and *scale-minus-left* [*simp*]: *scale* (− *a*) *x* = − (*scale a x*)
and *scale-left-diff-distrib* [*algebra-simps*]: *scale* $(a - b)$ *x* = *scale a x* − *scale b x*
and *scale-sum-left*: *scale* (*sum f A*) *x* = ($\sum$ *a*∈*A*. *scale* (*f a*) *x*)
⟨*proof*⟩

lemma *scale-zero-right* [*simp*]: *scale a 0* = *0*
and *scale-minus-right* [*simp*]: *scale a* (− *x*) = − (*scale a x*)
and *scale-right-diff-distrib* [*algebra-simps*]: *scale a* $(x - y)$ = *scale a x* − *scale a y*
and *scale-sum-right*: *scale a* (*sum f A*) = ($\sum$ *x*∈*A*. *scale a* (*f x*))
⟨*proof*⟩

lemma *scale-eq-0-iff* [*simp*]: *scale a x* = *0* ⟷ *a* = *0* ∨ *x* = *0*
⟨*proof*⟩

lemma *scale-left-imp-eq*:
assumes *nonzero*: *a* ≠ *0*
and *scale*: *scale a x* = *scale a y*
shows *x* = *y*
⟨*proof*⟩

lemma *scale-right-imp-eq*:
assumes *nonzero*: *x* ≠ *0*
and *scale*: *scale a x* = *scale b x*
shows *a* = *b*
⟨*proof*⟩

lemma *scale-cancel-left* [*simp*]: *scale a x* = *scale a y* ⟷ *x* = *y* ∨ *a* = *0*
⟨*proof*⟩

lemma *scale-cancel-right* [*simp*]: *scale a x* = *scale b x* ⟷ *a* = *b* ∨ *x* = *0*

⟨*proof*⟩

end

100.3 Real vector spaces

class *scaleR* =
 fixes *scaleR* :: *real* ⇒ *'a* ⇒ *'a* (**infixr** $*_R$ *75*)
begin

abbreviation *divideR* :: *'a* ⇒ *real* ⇒ *'a* (**infixl** $'/_R$ *70*)
 where $x\ /_R\ r$ ≡ *scaleR* (*inverse r*) *x*

end

class *real-vector* = *scaleR* + *ab-group-add* +
 assumes *scaleR-add-right*: *scaleR a* (*x* + *y*) = *scaleR a x* + *scaleR a y*
 and *scaleR-add-left*: *scaleR* (*a* + *b*) *x* = *scaleR a x* + *scaleR b x*
 and *scaleR-scaleR*: *scaleR a* (*scaleR b x*) = *scaleR* (*a* ∗ *b*) *x*
 and *scaleR-one*: *scaleR 1 x* = *x*

interpretation *real-vector*: *vector-space scaleR* :: *real* ⇒ *'a* ⇒ *'a*::*real-vector*
 ⟨*proof*⟩

Recover original theorem names

lemmas *scaleR-left-commute* = *real-vector.scale-left-commute*
lemmas *scaleR-zero-left* = *real-vector.scale-zero-left*
lemmas *scaleR-minus-left* = *real-vector.scale-minus-left*
lemmas *scaleR-diff-left* = *real-vector.scale-left-diff-distrib*
lemmas *scaleR-sum-left* = *real-vector.scale-sum-left*
lemmas *scaleR-zero-right* = *real-vector.scale-zero-right*
lemmas *scaleR-minus-right* = *real-vector.scale-minus-right*
lemmas *scaleR-diff-right* = *real-vector.scale-right-diff-distrib*
lemmas *scaleR-sum-right* = *real-vector.scale-sum-right*
lemmas *scaleR-eq-0-iff* = *real-vector.scale-eq-0-iff*
lemmas *scaleR-left-imp-eq* = *real-vector.scale-left-imp-eq*
lemmas *scaleR-right-imp-eq* = *real-vector.scale-right-imp-eq*
lemmas *scaleR-cancel-left* = *real-vector.scale-cancel-left*
lemmas *scaleR-cancel-right* = *real-vector.scale-cancel-right*

Legacy names

lemmas *scaleR-left-distrib* = *scaleR-add-left*
lemmas *scaleR-right-distrib* = *scaleR-add-right*
lemmas *scaleR-left-diff-distrib* = *scaleR-diff-left*
lemmas *scaleR-right-diff-distrib* = *scaleR-diff-right*

lemma *scaleR-minus1-left* [*simp*]: *scaleR* (−*1*) *x* = − *x*
 for *x* :: *'a*::*real-vector*
 ⟨*proof*⟩

lemma *scaleR-2*:
  fixes $x :: 'a::real\text{-}vector$
  shows $scaleR\ 2\ x = x + x$
  ⟨*proof*⟩

lemma *scaleR-half-double* [*simp*]:
  fixes $a :: 'a::real\text{-}vector$
  shows $(1\ /\ 2) *_R (a + a) = a$
⟨*proof*⟩

class *real-algebra* = *real-vector* + *ring* +
  assumes *mult-scaleR-left* [*simp*]: $scaleR\ a\ x * y = scaleR\ a\ (x * y)$
    and *mult-scaleR-right* [*simp*]: $x * scaleR\ a\ y = scaleR\ a\ (x * y)$

class *real-algebra-1* = *real-algebra* + *ring-1*

class *real-div-algebra* = *real-algebra-1* + *division-ring*

class *real-field* = *real-div-algebra* + *field*

instantiation *real* :: *real-field*
begin

definition *real-scaleR-def* [*simp*]: $scaleR\ a\ x = a * x$

instance
  ⟨*proof*⟩

end

interpretation *scaleR-left*: *additive* $(\lambda a.\ scaleR\ a\ x :: 'a::real\text{-}vector)$
  ⟨*proof*⟩

interpretation *scaleR-right*: *additive* $(\lambda x.\ scaleR\ a\ x :: 'a::real\text{-}vector)$
  ⟨*proof*⟩

lemma *nonzero-inverse-scaleR-distrib*:
  $a \neq 0 \Longrightarrow x \neq 0 \Longrightarrow inverse\ (scaleR\ a\ x) = scaleR\ (inverse\ a)\ (inverse\ x)$
  for $x :: 'a::real\text{-}div\text{-}algebra$
  ⟨*proof*⟩

lemma *inverse-scaleR-distrib*: $inverse\ (scaleR\ a\ x) = scaleR\ (inverse\ a)\ (inverse\ x)$
  for $x :: 'a::\{real\text{-}div\text{-}algebra,division\text{-}ring\}$
  ⟨*proof*⟩

lemma *sum-constant-scaleR*: $(\sum x \in A.\ y) = of\text{-}nat\ (card\ A) *_R y$
  for $y :: 'a::real\text{-}vector$

⟨*proof*⟩

named-theorems *vector-add-divide-simps to simplify sums of scaled vectors*

lemma [*vector-add-divide-simps*]:
$v + (b / z) *_R w = (\textit{if } z = 0 \textit{ then } v \textit{ else } (z *_R v + b *_R w) /_R z)$
$a *_R v + (b / z) *_R w = (\textit{if } z = 0 \textit{ then } a *_R v \textit{ else } ((a * z) *_R v + b *_R w) /_R z)$
$(a / z) *_R v + w = (\textit{if } z = 0 \textit{ then } w \textit{ else } (a *_R v + z *_R w) /_R z)$
$(a / z) *_R v + b *_R w = (\textit{if } z = 0 \textit{ then } b *_R w \textit{ else } (a *_R v + (b * z) *_R w) /_R z)$
$v - (b / z) *_R w = (\textit{if } z = 0 \textit{ then } v \textit{ else } (z *_R v - b *_R w) /_R z)$
$a *_R v - (b / z) *_R w = (\textit{if } z = 0 \textit{ then } a *_R v \textit{ else } ((a * z) *_R v - b *_R w) /_R z)$
$(a / z) *_R v - w = (\textit{if } z = 0 \textit{ then } -w \textit{ else } (a *_R v - z *_R w) /_R z)$
$(a / z) *_R v - b *_R w = (\textit{if } z = 0 \textit{ then } -b *_R w \textit{ else } (a *_R v - (b * z) *_R w) /_R z)$
for v :: $'a$ :: *real-vector*
⟨*proof*⟩

lemma *eq-vector-fraction-iff* [*vector-add-divide-simps*]:
fixes x :: $'a$ :: *real-vector*
shows $(x = (u / v) *_R a) \longleftrightarrow (\textit{if } v{=}0 \textit{ then } x = 0 \textit{ else } v *_R x = u *_R a)$
⟨*proof*⟩

lemma *vector-fraction-eq-iff* [*vector-add-divide-simps*]:
fixes x :: $'a$ :: *real-vector*
shows $((u / v) *_R a = x) \longleftrightarrow (\textit{if } v{=}0 \textit{ then } x = 0 \textit{ else } u *_R a = v *_R x)$
⟨*proof*⟩

lemma *real-vector-affinity-eq*:
fixes x :: $'a$ :: *real-vector*
assumes *m0*: $m \neq 0$
shows $m *_R x + c = y \longleftrightarrow x = \textit{inverse } m *_R y - (\textit{inverse } m *_R c)$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *real-vector-eq-affinity*: $m \neq 0 \Longrightarrow y = m *_R x + c \longleftrightarrow \textit{inverse } m *_R y - (\textit{inverse } m *_R c) = x$
for x :: $'a$::*real-vector*
⟨*proof*⟩

lemma *scaleR-eq-iff* [*simp*]: $b + u *_R a = a + u *_R b \longleftrightarrow a = b \lor u = 1$
for a :: $'a$::*real-vector*
⟨*proof*⟩

lemma *scaleR-collapse* [*simp*]: $(1 - u) *_R a + u *_R a = a$
for a :: $'a$::*real-vector*

⟨proof⟩

100.4 Embedding of the Reals into any *real-algebra-1*: *of-real*

definition *of-real* :: *real* ⇒ *'a*::*real-algebra-1*
where *of-real r* = *scaleR r 1*

lemma *scaleR-conv-of-real*: *scaleR r x* = *of-real r* ∗ *x*
⟨proof⟩

lemma *of-real-0* [*simp*]: *of-real 0* = *0*
⟨proof⟩

lemma *of-real-1* [*simp*]: *of-real 1* = *1*
⟨proof⟩

lemma *of-real-add* [*simp*]: *of-real* (*x* + *y*) = *of-real x* + *of-real y*
⟨proof⟩

lemma *of-real-minus* [*simp*]: *of-real* (− *x*) = − *of-real x*
⟨proof⟩

lemma *of-real-diff* [*simp*]: *of-real* (*x* − *y*) = *of-real x* − *of-real y*
⟨proof⟩

lemma *of-real-mult* [*simp*]: *of-real* (*x* ∗ *y*) = *of-real x* ∗ *of-real y*
⟨proof⟩

lemma *of-real-sum*[*simp*]: *of-real* (*sum f s*) = ($\sum$ *x*∈*s*. *of-real* (*f x*))
⟨proof⟩

lemma *of-real-prod*[*simp*]: *of-real* (*prod f s*) = ($\prod$ *x*∈*s*. *of-real* (*f x*))
⟨proof⟩

lemma *nonzero-of-real-inverse*:
x ≠ *0* ⟹ *of-real* (*inverse x*) = *inverse* (*of-real x* :: *'a*::*real-div-algebra*)
⟨proof⟩

lemma *of-real-inverse* [*simp*]:
of-real (*inverse x*) = *inverse* (*of-real x* :: *'a*::{*real-div-algebra*,*division-ring*})
⟨proof⟩

lemma *nonzero-of-real-divide*:
y ≠ *0* ⟹ *of-real* (*x* / *y*) = (*of-real x* / *of-real y* :: *'a*::*real-field*)
⟨proof⟩

lemma *of-real-divide* [*simp*]:
of-real (*x* / *y*) = (*of-real x* / *of-real y* :: *'a*::*real-div-algebra*)
⟨proof⟩

lemma *of-real-power* [*simp*]:
  of-real $(x \,\hat{}\, n) = (\textit{of-real } x :: {'}a::\{\textit{real-algebra-1}\}) \,\hat{}\, n$
  ⟨*proof*⟩

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

lemma *inj-of-real*: *inj of-real*
  ⟨*proof*⟩

lemmas *of-real-eq-0-iff* [*simp*] = *of-real-eq-iff* [*of - 0*, *simplified*]
lemmas *of-real-eq-1-iff* [*simp*] = *of-real-eq-iff* [*of - 1*, *simplified*]

lemma *of-real-eq-id* [*simp*]: *of-real* = (*id* :: *real* ⇒ *real*)
  ⟨*proof*⟩

Collapse nested embeddings.

lemma *of-real-of-nat-eq* [*simp*]: *of-real* (*of-nat* n) = *of-nat* n
  ⟨*proof*⟩

lemma *of-real-of-int-eq* [*simp*]: *of-real* (*of-int* z) = *of-int* z
  ⟨*proof*⟩

lemma *of-real-numeral* [*simp*]: *of-real* (*numeral* w) = *numeral* w
  ⟨*proof*⟩

lemma *of-real-neg-numeral* [*simp*]: *of-real* (− *numeral* w) = − *numeral* w
  ⟨*proof*⟩

Every real algebra has characteristic zero.

instance *real-algebra-1* < *ring-char-0*
⟨*proof*⟩

lemma *fraction-scaleR-times* [*simp*]:
  fixes a :: $'a$::*real-algebra-1*
  shows (*numeral* u / *numeral* v) $*_R$ (*numeral* $w * a$) = (*numeral* u * *numeral* w / *numeral* v) $*_R$ a
⟨*proof*⟩

lemma *inverse-scaleR-times* [*simp*]:
  fixes a :: $'a$::*real-algebra-1*
  shows (*1* / *numeral* v) $*_R$ (*numeral* $w * a$) = (*numeral* w / *numeral* v) $*_R$ a
⟨*proof*⟩

lemma *scaleR-times* [*simp*]:
  fixes a :: $'a$::*real-algebra-1*
  shows (*numeral* u) $*_R$ (*numeral* $w * a$) = (*numeral* u * *numeral* w) $*_R$ a
⟨*proof*⟩

instance *real-field* < *field-char-0* ⟨*proof*⟩

100.5 The Set of Real Numbers

definition *Reals* :: $'a$::*real-algebra-1 set* (ℝ)
where ℝ = *range of-real*

lemma *Reals-of-real* [*simp*]: *of-real* $r \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-of-int* [*simp*]: *of-int* $z \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-of-nat* [*simp*]: *of-nat* $n \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-numeral* [*simp*]: *numeral* $w \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-0* [*simp*]: $0 \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-1* [*simp*]: $1 \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-add* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a + b \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-minus* [*simp*]: $a \in \mathbb{R} \Longrightarrow - a \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-diff* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a - b \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-mult* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a * b \in \mathbb{R}$
⟨*proof*⟩

lemma *nonzero-Reals-inverse*: $a \in \mathbb{R} \Longrightarrow a \neq 0 \Longrightarrow$ *inverse* $a \in \mathbb{R}$
for a :: $'a$::*real-div-algebra*
⟨*proof*⟩

lemma *Reals-inverse*: $a \in \mathbb{R} \Longrightarrow$ *inverse* $a \in \mathbb{R}$
for a :: $'a$::{*real-div-algebra*,*division-ring*}
⟨*proof*⟩

lemma *Reals-inverse-iff* [*simp*]: *inverse* $x \in \mathbb{R} \longleftrightarrow x \in \mathbb{R}$
for x :: $'a$::{*real-div-algebra*,*division-ring*}
⟨*proof*⟩

lemma *nonzero-Reals-divide*: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow b \neq 0 \Longrightarrow a \mathbin{/} b \in \mathbb{R}$
for $a\ b$ :: $'a$::*real-field*
⟨*proof*⟩

lemma *Reals-divide* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a \mathbin{/} b \in \mathbb{R}$
for $a\ b$ :: $'a$::{*real-field*,*field*}
⟨*proof*⟩

lemma *Reals-power* [*simp*]: $a \in \mathbb{R} \Longrightarrow a\ \hat{}\ n \in \mathbb{R}$
for a :: $'a$::*real-algebra-1*
⟨*proof*⟩

lemma *Reals-cases* [*cases set*: *Reals*]:
assumes $q \in \mathbb{R}$
obtains (*of-real*) r **where** $q = \textit{of-real}\ r$
⟨*proof*⟩

lemma *sum-in-Reals* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{R}) \Longrightarrow \textit{sum}\ f\ s \in \mathbb{R}$
⟨*proof*⟩

lemma *prod-in-Reals* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{R}) \Longrightarrow \textit{prod}\ f\ s \in \mathbb{R}$
⟨*proof*⟩

lemma *Reals-induct* [*case-names of-real*, *induct set*: *Reals*]:
$q \in \mathbb{R} \Longrightarrow (\bigwedge r.\ P\ (\textit{of-real}\ r)) \Longrightarrow P\ q$
⟨*proof*⟩

100.6 Ordered real vector spaces

class *ordered-real-vector* = *real-vector* + *ordered-ab-group-add* +
assumes *scaleR-left-mono*: $x \le y \Longrightarrow 0 \le a \Longrightarrow a *_R x \le a *_R y$
and *scaleR-right-mono*: $a \le b \Longrightarrow 0 \le x \Longrightarrow a *_R x \le b *_R x$
begin

lemma *scaleR-mono*: $a \le b \Longrightarrow x \le y \Longrightarrow 0 \le b \Longrightarrow 0 \le x \Longrightarrow a *_R x \le b *_R y$
⟨*proof*⟩

lemma *scaleR-mono'*: $a \le b \Longrightarrow c \le d \Longrightarrow 0 \le a \Longrightarrow 0 \le c \Longrightarrow a *_R c \le b *_R d$
⟨*proof*⟩

lemma *pos-le-divideRI*:
assumes $0 < c$
and $c *_R a \le b$
shows $a \le b /_R c$
⟨*proof*⟩

lemma *pos-le-divideR-eq*:
assumes $0 < c$
shows $a \leq b /_R c \longleftrightarrow c *_R a \leq b$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *scaleR-image-atLeastAtMost*: $c > 0 \Longrightarrow scaleR\ c\ `\ \{x..y\} = \{c *_R x..c *_R y\}$
⟨*proof*⟩

end

lemma *neg-le-divideR-eq*:
fixes $a :: 'a ::$ *ordered-real-vector*
assumes $c < 0$
shows $a \leq b /_R c \longleftrightarrow b \leq c *_R a$
⟨*proof*⟩

lemma *scaleR-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq x \Longrightarrow 0 \leq a *_R x$
for $x :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-nonneg-nonpos*: $0 \leq a \Longrightarrow x \leq 0 \Longrightarrow a *_R x \leq 0$
for $x :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-nonpos-nonneg*: $a \leq 0 \Longrightarrow 0 \leq x \Longrightarrow a *_R x \leq 0$
for $x :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *split-scaleR-neg-le*: $(0 \leq a \wedge x \leq 0) \vee (a \leq 0 \wedge 0 \leq x) \Longrightarrow a *_R x \leq 0$
for $x :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *le-add-iff1*: $a *_R e + c \leq b *_R e + d \longleftrightarrow (a - b) *_R e + c \leq d$
for $c\ d\ e :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *le-add-iff2*: $a *_R e + c \leq b *_R e + d \longleftrightarrow c \leq (b - a) *_R e + d$
for $c\ d\ e :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-left-mono-neg*: $b \leq a \Longrightarrow c \leq 0 \Longrightarrow c *_R a \leq c *_R b$
for $a\ b :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-right-mono-neg*: $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a *_R c \leq b *_R c$
for $c :: 'a::$*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-nonpos-nonpos*: $a \le 0 \Longrightarrow b \le 0 \Longrightarrow 0 \le a *_R b$
for b :: $'a$::*ordered-real-vector*
⟨*proof*⟩

lemma *split-scaleR-pos-le*: $(0 \le a \wedge 0 \le b) \vee (a \le 0 \wedge b \le 0) \Longrightarrow 0 \le a *_R b$
for b :: $'a$::*ordered-real-vector*
⟨*proof*⟩

lemma *zero-le-scaleR-iff*:
fixes b :: $'a$::*ordered-real-vector*
shows $0 \le a *_R b \longleftrightarrow 0 < a \wedge 0 \le b \vee a < 0 \wedge b \le 0 \vee a = 0$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *scaleR-le-0-iff*: $a *_R b \le 0 \longleftrightarrow 0 < a \wedge b \le 0 \vee a < 0 \wedge 0 \le b \vee a = 0$
for b::$'a$::*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-le-cancel-left*: $c *_R a \le c *_R b \longleftrightarrow (0 < c \longrightarrow a \le b) \wedge (c < 0 \longrightarrow b \le a)$
for b :: $'a$::*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-le-cancel-left-pos*: $0 < c \Longrightarrow c *_R a \le c *_R b \longleftrightarrow a \le b$
for b :: $'a$::*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-le-cancel-left-neg*: $c < 0 \Longrightarrow c *_R a \le c *_R b \longleftrightarrow b \le a$
for b :: $'a$::*ordered-real-vector*
⟨*proof*⟩

lemma *scaleR-left-le-one-le*: $0 \le x \Longrightarrow a \le 1 \Longrightarrow a *_R x \le x$
for x :: $'a$::*ordered-real-vector* **and** a :: *real*
⟨*proof*⟩

100.7 Real normed vector spaces

class *dist* =
fixes *dist* :: $'a \Rightarrow 'a \Rightarrow$ *real*

class *norm* =
fixes *norm* :: $'a \Rightarrow$ *real*

class *sgn-div-norm* = *scaleR* + *norm* + *sgn* +
assumes *sgn-div-norm*: *sgn* $x = x /_R$ *norm* x

class *dist-norm* = *dist* + *norm* + *minus* +

assumes *dist-norm*: $dist\ x\ y = norm\ (x - y)$

class *uniformity-dist* = *dist* + *uniformity* +
assumes *uniformity-dist*: $uniformity = (INF\ e{:}\{0 <..\}.\ principal\ \{(x, y).\ dist\ x\ y < e\})$
begin

lemma *eventually-uniformity-metric*:
$eventually\ P\ uniformity \longleftrightarrow (\exists\, e{>}0.\ \forall\, x\ y.\ dist\ x\ y < e \longrightarrow P\ (x, y))$
$\langle proof \rangle$

end

class *real-normed-vector* = *real-vector* + *sgn-div-norm* + *dist-norm* + *uniformity-dist* + *open-uniformity* +
assumes *norm-eq-zero* [*simp*]: $norm\ x = 0 \longleftrightarrow x = 0$
and *norm-triangle-ineq*: $norm\ (x + y) \leq norm\ x + norm\ y$
and *norm-scaleR* [*simp*]: $norm\ (scaleR\ a\ x) = |a| * norm\ x$
begin

lemma *norm-ge-zero* [*simp*]: $0 \leq norm\ x$
$\langle proof \rangle$

end

class *real-normed-algebra* = *real-algebra* + *real-normed-vector* +
assumes *norm-mult-ineq*: $norm\ (x * y) \leq norm\ x * norm\ y$

class *real-normed-algebra-1* = *real-algebra-1* + *real-normed-algebra* +
assumes *norm-one* [*simp*]: $norm\ 1 = 1$

lemma (**in** *real-normed-algebra-1*) *scaleR-power* [*simp*]: $(scaleR\ x\ y)\ \hat{}\ n = scaleR\ (x\hat{}n)\ (y\hat{}n)$
$\langle proof \rangle$

class *real-normed-div-algebra* = *real-div-algebra* + *real-normed-vector* +
assumes *norm-mult*: $norm\ (x * y) = norm\ x * norm\ y$

class *real-normed-field* = *real-field* + *real-normed-div-algebra*

instance *real-normed-div-algebra* < *real-normed-algebra-1*
$\langle proof \rangle$

lemma *norm-zero* [*simp*]: $norm\ (0{::}'a{::}real\text{-}normed\text{-}vector) = 0$
$\langle proof \rangle$

lemma *zero-less-norm-iff* [*simp*]: $norm\ x > 0 \longleftrightarrow x \neq 0$
for $x :: 'a{::}real\text{-}normed\text{-}vector$
$\langle proof \rangle$

lemma *norm-not-less-zero* [*simp*]: $\neg$ *norm* $x < 0$
for x :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *norm-le-zero-iff* [*simp*]: *norm* $x \leq 0 \longleftrightarrow x = 0$
for x :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *norm-minus-cancel* [*simp*]: *norm* $(- x)$ = *norm* x
for x :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *norm-minus-commute*: *norm* $(a - b)$ = *norm* $(b - a)$
for a b :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *dist-add-cancel* [*simp*]: *dist* $(a + b)$ $(a + c)$ = *dist* b c
for a :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *dist-add-cancel2* [*simp*]: *dist* $(b + a)$ $(c + a)$ = *dist* b c
for a :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *dist-scaleR* [*simp*]: *dist* $(x *_R a)$ $(y *_R a) = |x - y| *$ *norm* a
for a :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *norm-uminus-minus*: *norm* $(- x - y$ :: $'a$ :: *real-normed-vector*$)$ = *norm* $(x + y)$
⟨*proof*⟩

lemma *norm-triangle-ineq2*: *norm* a − *norm* $b \leq$ *norm* $(a - b)$
for a b :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *norm-triangle-ineq3*: $|$*norm* a − *norm* $b| \leq$ *norm* $(a - b)$
for a b :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *norm-triangle-ineq4*: *norm* $(a - b) \leq$ *norm* a + *norm* b
for a b :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *norm-triangle-le-diff*:
fixes x y :: $'a$::*real-normed-vector*
shows *norm* x + *norm* $y \leq e \Longrightarrow$ *norm* $(x - y) \leq e$
⟨*proof*⟩

lemma *norm-diff-ineq*: *norm a* − *norm b* ≤ *norm* (*a* + *b*)
  for *a b* :: *'a::real-normed-vector*
⟨*proof*⟩

lemma *norm-add-leD*: *norm* (*a* + *b*) ≤ *c* ⟹ *norm b* ≤ *norm a* + *c*
  for *a b* :: *'a::real-normed-vector*
  ⟨*proof*⟩

lemma *norm-diff-triangle-ineq*: *norm* ((*a* + *b*) − (*c* + *d*)) ≤ *norm* (*a* − *c*) + *norm* (*b* − *d*)
  for *a b c d* :: *'a::real-normed-vector*
⟨*proof*⟩

lemma *norm-diff-triangle-le*:
  fixes *x y z* :: *'a::real-normed-vector*
  assumes *norm* (*x* − *y*) ≤ *e1* *norm* (*y* − *z*) ≤ *e2*
  shows *norm* (*x* − *z*) ≤ *e1* + *e2*
  ⟨*proof*⟩

lemma *norm-diff-triangle-less*:
  fixes *x y z* :: *'a::real-normed-vector*
  assumes *norm* (*x* − *y*) < *e1* *norm* (*y* − *z*) < *e2*
  shows *norm* (*x* − *z*) < *e1* + *e2*
  ⟨*proof*⟩

lemma *norm-triangle-mono*:
  fixes *a b* :: *'a::real-normed-vector*
  shows *norm a* ≤ *r* ⟹ *norm b* ≤ *s* ⟹ *norm* (*a* + *b*) ≤ *r* + *s*
  ⟨*proof*⟩

lemma *norm-sum*:
  fixes *f* :: *'a* ⇒ *'b::real-normed-vector*
  shows *norm* (*sum f A*) ≤ (∑ *i*∈*A*. *norm* (*f i*))
  ⟨*proof*⟩

lemma *sum-norm-le*:
  fixes *f* :: *'a* ⇒ *'b::real-normed-vector*
  assumes *fg*: ⋀*x*. *x* ∈ *S* ⟹ *norm* (*f x*) ≤ *g x*
  shows *norm* (*sum f S*) ≤ *sum g S*
  ⟨*proof*⟩

lemma *abs-norm-cancel* [*simp*]: |*norm a*| = *norm a*
  for *a* :: *'a::real-normed-vector*
  ⟨*proof*⟩

lemma *norm-add-less*: *norm x* < *r* ⟹ *norm y* < *s* ⟹ *norm* (*x* + *y*) < *r* + *s*
  for *x y* :: *'a::real-normed-vector*
  ⟨*proof*⟩

lemma *norm-mult-less*: $norm\ x < r \Longrightarrow norm\ y < s \Longrightarrow norm\ (x * y) < r * s$
for $x\ y$ :: *'a::real-normed-algebra*
⟨*proof*⟩

lemma *norm-of-real* [*simp*]: *norm* (*of-real* r :: *'a::real-normed-algebra-1*) = $|r|$
⟨*proof*⟩

lemma *norm-numeral* [*simp*]: *norm* (*numeral* w::*'a::real-normed-algebra-1*) = *numeral* w
⟨*proof*⟩

lemma *norm-neg-numeral* [*simp*]: *norm* ($-$ *numeral* w::*'a::real-normed-algebra-1*) = *numeral* w
⟨*proof*⟩

lemma *norm-of-real-add1* [*simp*]: *norm* (*of-real* $x + 1$ :: *'a :: real-normed-div-algebra*) = $|x + 1|$
⟨*proof*⟩

lemma *norm-of-real-addn* [*simp*]:
norm (*of-real* x + *numeral* b :: *'a :: real-normed-div-algebra*) = $|x$ + *numeral* $b|$
⟨*proof*⟩

lemma *norm-of-int* [*simp*]: *norm* (*of-int* z::*'a::real-normed-algebra-1*) = |*of-int* z|
⟨*proof*⟩

lemma *norm-of-nat* [*simp*]: *norm* (*of-nat* n::*'a::real-normed-algebra-1*) = *of-nat* n
⟨*proof*⟩

lemma *nonzero-norm-inverse*: $a \neq 0 \Longrightarrow norm\ (inverse\ a) = inverse\ (norm\ a)$
for a :: *'a::real-normed-div-algebra*
⟨*proof*⟩

lemma *norm-inverse*: $norm\ (inverse\ a) = inverse\ (norm\ a)$
for a :: *'a::{real-normed-div-algebra,division-ring}*
⟨*proof*⟩

lemma *nonzero-norm-divide*: $b \neq 0 \Longrightarrow norm\ (a\ /\ b) = norm\ a\ /\ norm\ b$
for $a\ b$ :: *'a::real-normed-field*
⟨*proof*⟩

lemma *norm-divide*: $norm\ (a\ /\ b) = norm\ a\ /\ norm\ b$
for $a\ b$ :: *'a::{real-normed-field,field}*
⟨*proof*⟩

lemma *norm-power-ineq*: $norm\ (x\ \hat{}\ n) \leq norm\ x\ \hat{}\ n$
for x :: *'a::real-normed-algebra-1*
⟨*proof*⟩

lemma *norm-power*: *norm* $(x \,\hat{}\, n)$ = *norm* $x \,\hat{}\, n$
for x :: *'a::real-normed-div-algebra*
⟨*proof*⟩

lemma *power-eq-imp-eq-norm*:
fixes w :: *'a::real-normed-div-algebra*
assumes *eq*: $w \,\hat{}\, n = z \,\hat{}\, n$ **and** $n > 0$
shows *norm* w = *norm* z
⟨*proof*⟩

lemma *norm-mult-numeral1* [*simp*]: *norm* (*numeral* $w * a$) = *numeral* $w *$ *norm* a
for a b :: *'a::{real-normed-field,field}*
⟨*proof*⟩

lemma *norm-mult-numeral2* [*simp*]: *norm* ($a *$ *numeral* w) = *norm* $a *$ *numeral* w
for a b :: *'a::{real-normed-field,field}*
⟨*proof*⟩

lemma *norm-divide-numeral* [*simp*]: *norm* (a / *numeral* w) = *norm* a / *numeral* w
for a b :: *'a::{real-normed-field,field}*
⟨*proof*⟩

lemma *norm-of-real-diff* [*simp*]:
norm (*of-real* b − *of-real* a :: *'a::real-normed-algebra-1*) $\leq |b - a|$
⟨*proof*⟩

Despite a superficial resemblance, *norm-eq-1* is not relevant.

lemma *square-norm-one*:
fixes x :: *'a::real-normed-div-algebra*
assumes $x^2 = 1$
shows *norm* $x = 1$
⟨*proof*⟩

lemma *norm-less-p1*: *norm* x < *norm* (*of-real* (*norm* x) + *1* :: *'a*)
for x :: *'a::real-normed-algebra-1*
⟨*proof*⟩

lemma *prod-norm*: *prod* (λx. *norm* (f x)) A = *norm* (*prod* f A)
for f :: *'a* ⇒ *'b::{comm-semiring-1,real-normed-div-algebra}*
⟨*proof*⟩

lemma *norm-prod-le*:
norm (*prod* f A) $\leq$ ($\prod a \in A$. *norm* (f a :: *'a* :: {*real-normed-algebra-1,comm-monoid-mult*}))
⟨*proof*⟩

lemma *norm-prod-diff*:
  fixes $z\ w :: 'i \Rightarrow 'a::\{real\text{-}normed\text{-}algebra\text{-}1,\ comm\text{-}monoid\text{-}mult\}$
  shows $(\bigwedge i.\ i \in I \Longrightarrow norm\ (z\ i) \leq 1) \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow norm\ (w\ i) \leq 1) \Longrightarrow$
    $norm\ ((\prod i{\in}I.\ z\ i) - (\prod i{\in}I.\ w\ i)) \leq (\sum i{\in}I.\ norm\ (z\ i - w\ i))$
⟨*proof*⟩

lemma *norm-power-diff*:
  fixes $z\ w :: 'a::\{real\text{-}normed\text{-}algebra\text{-}1,\ comm\text{-}monoid\text{-}mult\}$
  assumes $norm\ z \leq 1\ norm\ w \leq 1$
  shows $norm\ (z\hat{}m - w\hat{}m) \leq m * norm\ (z - w)$
⟨*proof*⟩

100.8 Metric spaces

class *metric-space* = *uniformity-dist* + *open-uniformity* +
  assumes *dist-eq-0-iff* [*simp*]: $dist\ x\ y = 0 \longleftrightarrow x = y$
    and *dist-triangle2*: $dist\ x\ y \leq dist\ x\ z + dist\ y\ z$
begin

lemma *dist-self* [*simp*]: $dist\ x\ x = 0$
  ⟨*proof*⟩

lemma *zero-le-dist* [*simp*]: $0 \leq dist\ x\ y$
  ⟨*proof*⟩

lemma *zero-less-dist-iff*: $0 < dist\ x\ y \longleftrightarrow x \neq y$
  ⟨*proof*⟩

lemma *dist-not-less-zero* [*simp*]: $\neg\ dist\ x\ y < 0$
  ⟨*proof*⟩

lemma *dist-le-zero-iff* [*simp*]: $dist\ x\ y \leq 0 \longleftrightarrow x = y$
  ⟨*proof*⟩

lemma *dist-commute*: $dist\ x\ y = dist\ y\ x$
⟨*proof*⟩

lemma *dist-commute-lessI*: $dist\ y\ x < e \Longrightarrow dist\ x\ y < e$
  ⟨*proof*⟩

lemma *dist-triangle*: $dist\ x\ z \leq dist\ x\ y + dist\ y\ z$
  ⟨*proof*⟩

lemma *dist-triangle3*: $dist\ x\ y \leq dist\ a\ x + dist\ a\ y$
  ⟨*proof*⟩

lemma *dist-pos-lt*: $x \neq y \Longrightarrow 0 < dist\ x\ y$
  ⟨*proof*⟩

lemma *dist-nz*: $x \neq y \longleftrightarrow 0 < dist\ x\ y$
⟨*proof*⟩

declare *dist-nz* [*symmetric*, *simp*]

lemma *dist-triangle-le*: $dist\ x\ z + dist\ y\ z \leq e \Longrightarrow dist\ x\ y \leq e$
⟨*proof*⟩

lemma *dist-triangle-lt*: $dist\ x\ z + dist\ y\ z < e \Longrightarrow dist\ x\ y < e$
⟨*proof*⟩

lemma *dist-triangle-less-add*: $dist\ x1\ y < e1 \Longrightarrow dist\ x2\ y < e2 \Longrightarrow dist\ x1\ x2 < e1 + e2$
⟨*proof*⟩

lemma *dist-triangle-half-l*: $dist\ x1\ y < e\ /\ 2 \Longrightarrow dist\ x2\ y < e\ /\ 2 \Longrightarrow dist\ x1\ x2 < e$
⟨*proof*⟩

lemma *dist-triangle-half-r*: $dist\ y\ x1 < e\ /\ 2 \Longrightarrow dist\ y\ x2 < e\ /\ 2 \Longrightarrow dist\ x1\ x2 < e$
⟨*proof*⟩

lemma *dist-triangle-third*:
assumes $dist\ x1\ x2 < e/3$ $dist\ x2\ x3 < e/3$ $dist\ x3\ x4 < e/3$
shows $dist\ x1\ x4 < e$
⟨*proof*⟩

subclass *uniform-space*
⟨*proof*⟩

lemma *open-dist*: $open\ S \longleftrightarrow (\forall x \in S.\ \exists e>0.\ \forall y.\ dist\ y\ x < e \longrightarrow y \in S)$
⟨*proof*⟩

lemma *open-ball*: $open\ \{y.\ dist\ x\ y < d\}$
⟨*proof*⟩

subclass *first-countable-topology*
⟨*proof*⟩

end

instance *metric-space* ⊆ *t2-space*
⟨*proof*⟩

Every normed vector space is a metric space.

instance *real-normed-vector* < *metric-space*
⟨*proof*⟩

100.9 Class instances for real numbers

instantiation *real* :: *real-normed-field*
begin

definition *dist-real-def*: *dist x y* = $|x - y|$

definition *uniformity-real-def* [*code del*]:
(*uniformity* :: (*real* × *real*) *filter*) = (*INF e*:{*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})

definition *open-real-def* [*code del*]:
open (*U* :: *real set*) ⟷ (∀ *x*∈*U*. *eventually* (λ(*x′*, *y*). *x′* = *x* ⟶ *y* ∈ *U*) *uniformity*)

definition *real-norm-def* [*simp*]: *norm r* = $|r|$

instance
⟨*proof*⟩

end

declare *uniformity-Abort*[**where** *′a*=*real*, *code*]

lemma *dist-of-real* [*simp*]: *dist* (*of-real x* :: *′a*) (*of-real y*) = *dist x y*
for *a* :: *′a*::*real-normed-div-algebra*
⟨*proof*⟩

declare [[*code abort*: *open* :: *real set* ⇒ *bool*]]

instance *real* :: *linorder-topology*
⟨*proof*⟩

instance *real* :: *linear-continuum-topology* ⟨*proof*⟩

lemmas *open-real-greaterThan* = *open-greaterThan*[**where** *′a*=*real*]
lemmas *open-real-lessThan* = *open-lessThan*[**where** *′a*=*real*]
lemmas *open-real-greaterThanLessThan* = *open-greaterThanLessThan*[**where** *′a*=*real*]
lemmas *closed-real-atMost* = *closed-atMost*[**where** *′a*=*real*]
lemmas *closed-real-atLeast* = *closed-atLeast*[**where** *′a*=*real*]
lemmas *closed-real-atLeastAtMost* = *closed-atLeastAtMost*[**where** *′a*=*real*]

100.10 Extra type constraints

Only allow *open* in class *topological-space*.

⟨*ML*⟩

Only allow *uniformity* in class *uniform-space*.

⟨*ML*⟩

Only allow *dist* in class *metric-space*.

⟨*ML*⟩

Only allow *norm* in class *real-normed-vector*.

⟨*ML*⟩

100.11 Sign function

lemma *norm-sgn*: *norm* (*sgn x*) = (*if x* = *0 then 0 else 1*)
for *x* :: *'a*::*real-normed-vector*
⟨*proof*⟩

lemma *sgn-zero* [*simp*]: *sgn* (*0*::*'a*::*real-normed-vector*) = *0*
⟨*proof*⟩

lemma *sgn-zero-iff*: *sgn x* = *0* ⟷ *x* = *0*
for *x* :: *'a*::*real-normed-vector*
⟨*proof*⟩

lemma *sgn-minus*: *sgn* (− *x*) = − *sgn x*
for *x* :: *'a*::*real-normed-vector*
⟨*proof*⟩

lemma *sgn-scaleR*: *sgn* (*scaleR r x*) = *scaleR* (*sgn r*) (*sgn x*)
for *x* :: *'a*::*real-normed-vector*
⟨*proof*⟩

lemma *sgn-one* [*simp*]: *sgn* (*1*::*'a*::*real-normed-algebra-1*) = *1*
⟨*proof*⟩

lemma *sgn-of-real*: *sgn* (*of-real r* :: *'a*::*real-normed-algebra-1*) = *of-real* (*sgn r*)
⟨*proof*⟩

lemma *sgn-mult*: *sgn* (*x* ∗ *y*) = *sgn x* ∗ *sgn y*
for *x y* :: *'a*::*real-normed-div-algebra*
⟨*proof*⟩

hide-fact (**open**) *sgn-mult*

lemma *real-sgn-eq*: *sgn x* = *x* / |*x*|
for *x* :: *real*
⟨*proof*⟩

lemma *zero-le-sgn-iff* [*simp*]: *0* ≤ *sgn x* ⟷ *0* ≤ *x*
for *x* :: *real*
⟨*proof*⟩

lemma *sgn-le-0-iff* [*simp*]: *sgn x* ≤ *0* ⟷ *x* ≤ *0*
for *x* :: *real*

⟨*proof*⟩

lemma *norm-conv-dist*: *norm* x = *dist* x *0*
⟨*proof*⟩

declare *norm-conv-dist* [*symmetric*, *simp*]

lemma *dist-0-norm* [*simp*]: *dist 0* x = *norm* x
for x :: *'a::real-normed-vector*
⟨*proof*⟩

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

lemma *dist-of-int*: *dist* (*of-int* m) (*of-int* n :: *'a* :: *real-normed-algebra-1*) = *of-int* $|m - n|$
⟨*proof*⟩

lemma *dist-of-nat*:
dist (*of-nat* m) (*of-nat* n :: *'a* :: *real-normed-algebra-1*) = *of-int* |*int* m − *int* n|
⟨*proof*⟩

100.12 Bounded Linear and Bilinear Operators

locale *linear* = *additive* f **for** f :: *'a::real-vector* ⇒ *'b::real-vector* +
assumes *scaleR*: f (*scaleR* r x) = *scaleR* r (f x)

lemma *linear-imp-scaleR*:
assumes *linear* D
obtains d **where** $D = (\lambda x.\ x *_R d)$
⟨*proof*⟩

corollary *real-linearD*:
fixes f :: *real* ⇒ *real*
assumes *linear* f **obtains** c **where** f = *op∗* c
⟨*proof*⟩

lemma *linear-times-of-real*: *linear* ($\lambda x.$ a ∗ *of-real* x)
⟨*proof*⟩

lemma *linearI*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
and $\bigwedge c\ x.\ f\ (c *_R x) = c *_R f\ x$
shows *linear* f
⟨*proof*⟩

locale *bounded-linear* = *linear* f **for** f :: *'a::real-normed-vector* ⇒ *'b::real-normed-vector* +
assumes *bounded*: $\exists K.\ \forall x.$ *norm* (f x) ≤ *norm* x ∗ K

begin

lemma *pos-bounded*: $\exists K>0.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$
⟨*proof*⟩

lemma *nonneg-bounded*: $\exists K\geq 0.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$
⟨*proof*⟩

lemma *linear*: *linear f*
⟨*proof*⟩

end

lemma *bounded-linear-intro*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
and $\bigwedge r\ x.\ f\ (scaleR\ r\ x) = scaleR\ r\ (f\ x)$
and $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * K$
shows *bounded-linear f*
⟨*proof*⟩

locale *bounded-bilinear* =
fixes *prod* :: $'a$::*real-normed-vector* ⇒ $'b$::*real-normed-vector* ⇒ $'c$::*real-normed-vector*
(**infixl** $**$ *70*)
assumes *add-left*: $prod\ (a + a')\ b = prod\ a\ b + prod\ a'\ b$
and *add-right*: $prod\ a\ (b + b') = prod\ a\ b + prod\ a\ b'$
and *scaleR-left*: $prod\ (scaleR\ r\ a)\ b = scaleR\ r\ (prod\ a\ b)$
and *scaleR-right*: $prod\ a\ (scaleR\ r\ b) = scaleR\ r\ (prod\ a\ b)$
and *bounded*: $\exists K.\ \forall a\ b.\ norm\ (prod\ a\ b) \leq norm\ a * norm\ b * K$
begin

lemma *pos-bounded*: $\exists K>0.\ \forall a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$
⟨*proof*⟩

lemma *nonneg-bounded*: $\exists K\geq 0.\ \forall a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$
⟨*proof*⟩

lemma *additive-right*: *additive* $(\lambda b.\ prod\ a\ b)$
⟨*proof*⟩

lemma *additive-left*: *additive* $(\lambda a.\ prod\ a\ b)$
⟨*proof*⟩

lemma *zero-left*: $prod\ 0\ b = 0$
⟨*proof*⟩

lemma *zero-right*: $prod\ a\ 0 = 0$
⟨*proof*⟩

lemma *minus-left*: $prod\ (-\ a)\ b = -\ prod\ a\ b$

⟨*proof*⟩

lemma *minus-right*: $prod\ a\ (-\ b) = -\ prod\ a\ b$
⟨*proof*⟩

lemma *diff-left*: $prod\ (a - a')\ b = prod\ a\ b - prod\ a'\ b$
⟨*proof*⟩

lemma *diff-right*: $prod\ a\ (b - b') = prod\ a\ b - prod\ a\ b'$
⟨*proof*⟩

lemma *sum-left*: $prod\ (sum\ g\ S)\ x = sum\ ((\lambda i.\ prod\ (g\ i)\ x))\ S$
⟨*proof*⟩

lemma *sum-right*: $prod\ x\ (sum\ g\ S) = sum\ ((\lambda i.\ (prod\ x\ (g\ i))))\ S$
⟨*proof*⟩

lemma *bounded-linear-left*: $bounded\text{-}linear\ (\lambda a.\ a\ {*}{*}\ b)$
⟨*proof*⟩

lemma *bounded-linear-right*: $bounded\text{-}linear\ (\lambda b.\ a\ {*}{*}\ b)$
⟨*proof*⟩

lemma *prod-diff-prod*: $(x\ {*}{*}\ y - a\ {*}{*}\ b) = (x - a)\ {*}{*}\ (y - b) + (x - a)\ {*}{*}\ b + a\ {*}{*}\ (y - b)$
⟨*proof*⟩

lemma *flip*: $bounded\text{-}bilinear\ (\lambda x\ y.\ y\ {*}{*}\ x)$
⟨*proof*⟩

lemma *comp1*:
assumes $bounded\text{-}linear\ g$
shows $bounded\text{-}bilinear\ (\lambda x.\ op\ {*}{*}\ (g\ x))$
⟨*proof*⟩

lemma *comp*: $bounded\text{-}linear\ f \Longrightarrow bounded\text{-}linear\ g \Longrightarrow bounded\text{-}bilinear\ (\lambda x\ y.\ f\ x\ {*}{*}\ g\ y)$
⟨*proof*⟩

end

lemma *bounded-linear-ident*[*simp*]: $bounded\text{-}linear\ (\lambda x.\ x)$
⟨*proof*⟩

lemma *bounded-linear-zero*[*simp*]: $bounded\text{-}linear\ (\lambda x.\ 0)$
⟨*proof*⟩

lemma *bounded-linear-add*:

assumes *bounded-linear f*
and *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *bounded-linear-minus*:
assumes *bounded-linear f*
shows *bounded-linear* $(\lambda x.\ - f\ x)$
⟨*proof*⟩

lemma *bounded-linear-sub*: *bounded-linear f* $\Longrightarrow$ *bounded-linear g* $\Longrightarrow$ *bounded-linear* $(\lambda x.\ f\ x - g\ x)$
⟨*proof*⟩

lemma *bounded-linear-sum*:
fixes f :: $'i \Rightarrow 'a$::*real-normed-vector* $\Rightarrow 'b$::*real-normed-vector*
shows $(\bigwedge i.\ i \in I \Longrightarrow$ *bounded-linear* $(f\ i)) \Longrightarrow$ *bounded-linear* $(\lambda x.\ \sum i{\in}I.\ f\ i\ x)$
⟨*proof*⟩

lemma *bounded-linear-compose*:
assumes *bounded-linear f*
and *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma *bounded-bilinear-mult*: *bounded-bilinear* $(op * :: 'a \Rightarrow 'a \Rightarrow 'a$::*real-normed-algebra*)
⟨*proof*⟩

lemma *bounded-linear-mult-left*: *bounded-linear* $(\lambda x{::}'a$::*real-normed-algebra*. $x * y)$
⟨*proof*⟩

lemma *bounded-linear-mult-right*: *bounded-linear* $(\lambda y{::}'a$::*real-normed-algebra*. $x * y)$
⟨*proof*⟩

lemmas *bounded-linear-mult-const* =
bounded-linear-mult-left [*THEN bounded-linear-compose*]

lemmas *bounded-linear-const-mult* =
bounded-linear-mult-right [*THEN bounded-linear-compose*]

lemma *bounded-linear-divide*: *bounded-linear* $(\lambda x.\ x\ /\ y)$
for y :: $'a$::*real-normed-field*
⟨*proof*⟩

lemma *bounded-bilinear-scaleR*: *bounded-bilinear scaleR*
⟨*proof*⟩

lemma *bounded-linear-scaleR-left*: *bounded-linear* $(\lambda r.\ scaleR\ r\ x)$
⟨*proof*⟩

lemma *bounded-linear-scaleR-right*: *bounded-linear* $(\lambda x.\ scaleR\ r\ x)$
⟨*proof*⟩

lemmas *bounded-linear-scaleR-const* =
bounded-linear-scaleR-left[*THEN bounded-linear-compose*]

lemmas *bounded-linear-const-scaleR* =
bounded-linear-scaleR-right[*THEN bounded-linear-compose*]

lemma *bounded-linear-of-real*: *bounded-linear* $(\lambda r.\ of\text{-}real\ r)$
⟨*proof*⟩

lemma *real-bounded-linear*: *bounded-linear* $f \longleftrightarrow (\exists\, c{::}real.\ f = (\lambda x.\ x * c))$
for $f :: real \Rightarrow real$
⟨*proof*⟩

lemma *bij-linear-imp-inv-linear*: *linear* $f \Longrightarrow bij\ f \Longrightarrow linear\ (inv\ f)$
⟨*proof*⟩

instance *real-normed-algebra-1* $\subseteq$ *perfect-space*
⟨*proof*⟩

100.13 Filters and Limits on Metric Space

lemma (**in** *metric-space*) *nhds-metric*: $nhds\ x = (INF\ e{:}\{0 <..\}.\ principal\ \{y.\ dist\ y\ x < e\})$
⟨*proof*⟩

lemma (**in** *metric-space*) *tendsto-iff*: $(f \longrightarrow l)\ F \longleftrightarrow (\forall\, e{>}0.\ eventually\ (\lambda x.\ dist\ (f\ x)\ l < e)\ F)$
⟨*proof*⟩

lemma (**in** *metric-space*) *tendstoI* [*intro?*]:
$(\bigwedge e.\ 0 < e \Longrightarrow eventually\ (\lambda x.\ dist\ (f\ x)\ l < e)\ F) \Longrightarrow (f \longrightarrow l)\ F$
⟨*proof*⟩

lemma (**in** *metric-space*) *tendstoD*: $(f \longrightarrow l)\ F \Longrightarrow 0 < e \Longrightarrow eventually\ (\lambda x.\ dist\ (f\ x)\ l < e)\ F$
⟨*proof*⟩

lemma (**in** *metric-space*) *eventually-nhds-metric*:
$eventually\ P\ (nhds\ a) \longleftrightarrow (\exists\, d{>}0.\ \forall\, x.\ dist\ x\ a < d \longrightarrow P\ x)$
⟨*proof*⟩

lemma *eventually-at*: $eventually\ P\ (at\ a\ within\ S) \longleftrightarrow (\exists\, d{>}0.\ \forall\, x{\in}S.\ x \neq a\ \wedge$

dist x a $<$ *d* $\longrightarrow$ *P x*)
for *a* :: *'a* :: *metric-space*
⟨*proof*⟩

lemma *eventually-at-le*: *eventually P* (*at a within S*) $\longleftrightarrow$ ($\exists$ *d*>*0*. $\forall$ *x*$\in$*S*. *x* $\neq$ *a* $\wedge$ *dist x a* $\leq$ *d* $\longrightarrow$ *P x*)
for *a* :: *'a*::*metric-space*
⟨*proof*⟩

lemma *eventually-at-left-real*: *a* > (*b* :: *real*) $\Longrightarrow$ *eventually* (λ*x*. *x* $\in$ {*b*<..<*a*}) (*at-left a*)
⟨*proof*⟩

lemma *eventually-at-right-real*: *a* < (*b* :: *real*) $\Longrightarrow$ *eventually* (λ*x*. *x* $\in$ {*a*<..<*b*}) (*at-right a*)
⟨*proof*⟩

lemma *metric-tendsto-imp-tendsto*:
fixes *a* :: *'a* :: *metric-space*
and *b* :: *'b* :: *metric-space*
assumes *f*: (*f* $\longrightarrow$ *a*) *F*
and *le*: *eventually* (λ*x*. *dist* (*g x*) *b* $\leq$ *dist* (*f x*) *a*) *F*
shows (*g* $\longrightarrow$ *b*) *F*
⟨*proof*⟩

lemma *filterlim-real-sequentially*: *LIM x sequentially*. *real x* :> *at-top*
⟨*proof*⟩

lemma *filterlim-nat-sequentially*: *filterlim nat sequentially at-top*
⟨*proof*⟩

lemma *filterlim-floor-sequentially*: *filterlim floor at-top at-top*
⟨*proof*⟩

lemma *filterlim-sequentially-iff-filterlim-real*:
filterlim f sequentially F $\longleftrightarrow$ *filterlim* (λ*x*. *real* (*f x*)) *at-top F*
⟨*proof*⟩

100.13.1 Limits of Sequences

lemma *lim-sequentially*: *X* $\longrightarrow$ *L* $\longleftrightarrow$ ($\forall$ *r*>*0*. $\exists$ *no*. $\forall$ *n*$\geq$*no*. *dist* (*X n*) *L* < *r*)
for *L* :: *'a*::*metric-space*
⟨*proof*⟩

lemmas *LIMSEQ-def* = *lim-sequentially*

lemma *LIMSEQ-iff-nz*: *X* $\longrightarrow$ *L* $\longleftrightarrow$ ($\forall$ *r*>*0*. $\exists$ *no*>*0*. $\forall$ *n*$\geq$*no*. *dist* (*X n*) *L* < *r*)

for L :: $'a$::*metric-space*
⟨*proof*⟩

lemma *metric-LIMSEQ-I*: $(\bigwedge r.\ 0 < r \Longrightarrow \exists\, no.\ \forall\, n{\ge}no.\ dist\ (X\ n)\ L < r) \Longrightarrow X \longrightarrow L$
for L :: $'a$::*metric-space*
⟨*proof*⟩

lemma *metric-LIMSEQ-D*: $X \longrightarrow L \Longrightarrow 0 < r \Longrightarrow \exists\, no.\ \forall\, n{\ge}no.\ dist\ (X\ n)\ L < r$
for L :: $'a$::*metric-space*
⟨*proof*⟩

100.13.2 Limits of Functions

lemma *LIM-def*: $f\ -a\rightarrow\ L \longleftrightarrow (\forall\, r > 0.\ \exists\, s > 0.\ \forall\, x.\ x \ne a \wedge dist\ x\ a < s \longrightarrow dist\ (f\ x)\ L < r)$
for a :: $'a$::*metric-space* **and** L :: $'b$::*metric-space*
⟨*proof*⟩

lemma *metric-LIM-I*:
$(\bigwedge r.\ 0 < r \Longrightarrow \exists\, s{>}0.\ \forall\, x.\ x \ne a \wedge dist\ x\ a < s \longrightarrow dist\ (f\ x)\ L < r) \Longrightarrow f\ -a\rightarrow\ L$
for a :: $'a$::*metric-space* **and** L :: $'b$::*metric-space*
⟨*proof*⟩

lemma *metric-LIM-D*: $f\ -a\rightarrow\ L \Longrightarrow 0 < r \Longrightarrow \exists\, s{>}0.\ \forall\, x.\ x \ne a \wedge dist\ x\ a < s \longrightarrow dist\ (f\ x)\ L < r$
for a :: $'a$::*metric-space* **and** L :: $'b$::*metric-space*
⟨*proof*⟩

lemma *metric-LIM-imp-LIM*:
fixes l :: $'a$::*metric-space*
and m :: $'b$::*metric-space*
assumes f: $f\ -a\rightarrow\ l$
and *le*: $\bigwedge x.\ x \ne a \Longrightarrow dist\ (g\ x)\ m \le dist\ (f\ x)\ l$
shows $g\ -a\rightarrow\ m$
⟨*proof*⟩

lemma *metric-LIM-equal2*:
fixes a :: $'a$::*metric-space*
assumes $0 < R$
and $\bigwedge x.\ x \ne a \Longrightarrow dist\ x\ a < R \Longrightarrow f\ x = g\ x$
shows $g\ -a\rightarrow\ l \Longrightarrow f\ -a\rightarrow\ l$
⟨*proof*⟩

lemma *metric-LIM-compose2*:
fixes a :: $'a$::*metric-space*
assumes f: $f\ -a\rightarrow\ b$

and *g*: $g \ -b\rightarrow\ c$
and *inj*: $\exists\, d>0.\ \forall\, x.\ x \neq a \wedge dist\ x\ a < d \longrightarrow f\ x \neq b$
shows $(\lambda x.\ g\ (f\ x))\ -a\rightarrow\ c$
⟨*proof*⟩

lemma *metric-isCont-LIM-compose2*:
fixes f :: $'a$ :: *metric-space* $\Rightarrow$ -
assumes f [*unfolded isCont-def*]: *isCont f a*
and *g*: $g \ -f\ a\rightarrow\ l$
and *inj*: $\exists\, d>0.\ \forall\, x.\ x \neq a \wedge dist\ x\ a < d \longrightarrow f\ x \neq f\ a$
shows $(\lambda x.\ g\ (f\ x))\ -a\rightarrow\ l$
⟨*proof*⟩

100.14 Complete metric spaces

100.15 Cauchy sequences

lemma (**in** *metric-space*) *Cauchy-def*: $Cauchy\ X = (\forall\, e>0.\ \exists\, M.\ \forall\, m\geq M.\ \forall\, n\geq M.\ dist\ (X\ m)\ (X\ n) < e)$
⟨*proof*⟩

lemma (**in** *metric-space*) *Cauchy-altdef*: $Cauchy\ f \longleftrightarrow (\forall\, e>0.\ \exists\, M.\ \forall\, m\geq M.\ \forall\, n>m.\ dist\ (f\ m)\ (f\ n) < e)$
(**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

lemma (**in** *metric-space*) *Cauchy-altdef2*: $Cauchy\ s \longleftrightarrow (\forall\, e>0.\ \exists\, N{::}nat.\ \forall\, n\geq N.\ dist(s\ n)(s\ N) < e)$ (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma (**in** *metric-space*) *metric-CauchyI*:
$(\bigwedge e.\ 0 < e \Longrightarrow \exists\, M.\ \forall\, m\geq M.\ \forall\, n\geq M.\ dist\ (X\ m)\ (X\ n) < e) \Longrightarrow Cauchy\ X$
⟨*proof*⟩

lemma (**in** *metric-space*) *CauchyI'*:
$(\bigwedge e.\ 0 < e \Longrightarrow \exists\, M.\ \forall\, m\geq M.\ \forall\, n>m.\ dist\ (X\ m)\ (X\ n) < e) \Longrightarrow Cauchy\ X$
⟨*proof*⟩

lemma (**in** *metric-space*) *metric-CauchyD*:
$Cauchy\ X \Longrightarrow 0 < e \Longrightarrow \exists\, M.\ \forall\, m\geq M.\ \forall\, n\geq M.\ dist\ (X\ m)\ (X\ n) < e$
⟨*proof*⟩

lemma (**in** *metric-space*) *metric-Cauchy-iff2*:
$Cauchy\ X = (\forall\, j.\ (\exists\, M.\ \forall\, m \geq M.\ \forall\, n \geq M.\ dist\ (X\ m)\ (X\ n) < inverse(real\ (Suc\ j))))$
⟨*proof*⟩

lemma *Cauchy-iff2*: $Cauchy\ X \longleftrightarrow (\forall\, j.\ (\exists\, M.\ \forall\, m \geq M.\ \forall\, n \geq M.\ |X\ m - X\ n| < inverse\ (real\ (Suc\ j))))$
⟨*proof*⟩

lemma *lim-1-over-n*: $((\lambda n.\ 1\ /\ \textit{of-nat}\ n) \longrightarrow (0::'a::\textit{real-normed-field}))$ *sequentially*
⟨*proof*⟩

lemma (**in** *metric-space*) *complete-def*:
shows $\textit{complete}\ S = (\forall f.\ (\forall n.\ f\ n \in S) \wedge \textit{Cauchy}\ f \longrightarrow (\exists l{\in}S.\ f \longrightarrow l))$
⟨*proof*⟩

lemma (**in** *metric-space*) *totally-bounded-metric*:
$\textit{totally-bounded}\ S \longleftrightarrow (\forall e{>}0.\ \exists k.\ \textit{finite}\ k \wedge S \subseteq (\bigcup x{\in}k.\ \{y.\ \textit{dist}\ x\ y < e\}))$
⟨*proof*⟩

100.15.1 Cauchy Sequences are Convergent

class *complete-space* = *metric-space* +
assumes *Cauchy-convergent*: $\textit{Cauchy}\ X \Longrightarrow \textit{convergent}\ X$

lemma *Cauchy-convergent-iff*: $\textit{Cauchy}\ X \longleftrightarrow \textit{convergent}\ X$
for $X :: \textit{nat} \Rightarrow {'a}::\textit{complete-space}$
⟨*proof*⟩

100.16 The set of real numbers is a complete metric space

Proof that Cauchy sequences converge based on the one from http://pirate.shu.edu/~wachsmut/ira/numseq/proofs/cauconv.html

If sequence X is Cauchy, then its limit is the lub of $\{r.\ \exists N.\ \forall n{\geq}N.\ r < X\ n\}$

lemma *increasing-LIMSEQ*:
fixes $f :: \textit{nat} \Rightarrow \textit{real}$
assumes *inc*: $\bigwedge n.\ f\ n \leq f\ (\textit{Suc}\ n)$
and *bdd*: $\bigwedge n.\ f\ n \leq l$
and *en*: $\bigwedge e.\ 0 < e \Longrightarrow \exists n.\ l \leq f\ n + e$
shows $f \longrightarrow l$
⟨*proof*⟩

lemma *real-Cauchy-convergent*:
fixes $X :: \textit{nat} \Rightarrow \textit{real}$
assumes X: *Cauchy X*
shows *convergent X*
⟨*proof*⟩

instance *real* :: *complete-space*
⟨*proof*⟩

class *banach* = *real-normed-vector* + *complete-space*

instance *real* :: *banach* ⟨*proof*⟩

lemma *tendsto-at-topI-sequentially*:
  fixes f :: *real* $\Rightarrow$ $'b$::*first-countable-topology*
  assumes $*$: $\bigwedge X$. *filterlim* X *at-top sequentially* $\Longrightarrow$ $(\lambda n.\ f\ (X\ n)) \longrightarrow y$
  shows $(f \longrightarrow y)$ *at-top*
$\langle proof \rangle$

lemma *tendsto-at-topI-sequentially-real*:
  fixes f :: *real* $\Rightarrow$ *real*
  assumes *mono*: *mono f*
    and *limseq*: $(\lambda n.\ f\ (real\ n)) \longrightarrow y$
  shows $(f \longrightarrow y)$ *at-top*
$\langle proof \rangle$

end

101 Limits on Real Vector Spaces

theory *Limits*
  imports *Real-Vector-Spaces*
begin

101.1 Filter going to infinity norm

definition *at-infinity* :: $'a$::*real-normed-vector filter*
  where *at-infinity* = (*INF* r. *principal* $\{x.\ r \leq norm\ x\}$)

lemma *eventually-at-infinity*: *eventually* P *at-infinity* $\longleftrightarrow$ $(\exists b.\ \forall x.\ b \leq norm\ x \longrightarrow P\ x)$
  $\langle proof \rangle$

corollary *eventually-at-infinity-pos*:
  eventually p *at-infinity* $\longleftrightarrow$ $(\exists b.\ 0 < b \wedge (\forall x.\ norm\ x \geq b \longrightarrow p\ x))$
  $\langle proof \rangle$

lemma *at-infinity-eq-at-top-bot*: (*at-infinity* :: *real filter*) = *sup at-top at-bot*
  $\langle proof \rangle$

lemma *at-top-le-at-infinity*: *at-top* $\leq$ (*at-infinity* :: *real filter*)
  $\langle proof \rangle$

lemma *at-bot-le-at-infinity*: *at-bot* $\leq$ (*at-infinity* :: *real filter*)
  $\langle proof \rangle$

lemma *filterlim-at-top-imp-at-infinity*: *filterlim f at-top F* $\Longrightarrow$ *filterlim f at-infinity F*
  for f :: - $\Rightarrow$ *real*
  $\langle proof \rangle$

lemma *lim-infinity-imp-sequentially*: $(f \longrightarrow l)$ *at-infinity* $\Longrightarrow$ $((\lambda n.\ f(n)) \longrightarrow$

l) sequentially
⟨*proof*⟩

101.1.1 Boundedness

definition *Bfun* :: $('a \Rightarrow 'b{::}metric\text{-}space) \Rightarrow 'a\ filter \Rightarrow bool$
where *Bfun-metric-def*: $Bfun\ f\ F = (\exists\, y.\ \exists\, K{>}0.\ eventually\ (\lambda x.\ dist\ (f\ x)\ y \le K)\ F)$

abbreviation *Bseq* :: $(nat \Rightarrow 'a{::}metric\text{-}space) \Rightarrow bool$
where $Bseq\ X \equiv Bfun\ X\ sequentially$

lemma *Bseq-conv-Bfun*: $Bseq\ X \longleftrightarrow Bfun\ X\ sequentially$ ⟨*proof*⟩

lemma *Bseq-ignore-initial-segment*: $Bseq\ X \Longrightarrow Bseq\ (\lambda n.\ X\ (n + k))$
⟨*proof*⟩

lemma *Bseq-offset*: $Bseq\ (\lambda n.\ X\ (n + k)) \Longrightarrow Bseq\ X$
⟨*proof*⟩

lemma *Bfun-def*: $Bfun\ f\ F \longleftrightarrow (\exists\, K{>}0.\ eventually\ (\lambda x.\ norm\ (f\ x) \le K)\ F)$
⟨*proof*⟩

lemma *BfunI*:
assumes K: $eventually\ (\lambda x.\ norm\ (f\ x) \le K)\ F$
shows $Bfun\ f\ F$
⟨*proof*⟩

lemma *BfunE*:
assumes $Bfun\ f\ F$
obtains B **where** $0 < B$ **and** $eventually\ (\lambda x.\ norm\ (f\ x) \le B)\ F$
⟨*proof*⟩

lemma *Cauchy-Bseq*: $Cauchy\ X \Longrightarrow Bseq\ X$
⟨*proof*⟩

101.1.2 Bounded Sequences

lemma *BseqI'*: $(\bigwedge n.\ norm\ (X\ n) \le K) \Longrightarrow Bseq\ X$
⟨*proof*⟩

lemma *BseqI2'*: $\forall\, n{\ge}N.\ norm\ (X\ n) \le K \Longrightarrow Bseq\ X$
⟨*proof*⟩

lemma *Bseq-def*: $Bseq\ X \longleftrightarrow (\exists\, K{>}0.\ \forall\, n.\ norm\ (X\ n) \le K)$
⟨*proof*⟩

lemma *BseqE*: $Bseq\ X \Longrightarrow (\bigwedge K.\ 0 < K \Longrightarrow \forall\, n.\ norm\ (X\ n) \le K \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *BseqD*: $Bseq\ X \Longrightarrow \exists K.\ 0 < K \land (\forall n.\ norm\ (X\ n) \le K)$
⟨*proof*⟩

lemma *BseqI*: $0 < K \Longrightarrow \forall n.\ norm\ (X\ n) \le K \Longrightarrow Bseq\ X$
⟨*proof*⟩

lemma *Bseq-bdd-above*: $Bseq\ X \Longrightarrow bdd\text{-}above\ (range\ X)$
for $X :: nat \Rightarrow real$
⟨*proof*⟩

lemma *Bseq-bdd-above'*: $Bseq\ X \Longrightarrow bdd\text{-}above\ (range\ (\lambda n.\ norm\ (X\ n)))$
for $X :: nat \Rightarrow 'a :: real\text{-}normed\text{-}vector$
⟨*proof*⟩

lemma *Bseq-bdd-below*: $Bseq\ X \Longrightarrow bdd\text{-}below\ (range\ X)$
for $X :: nat \Rightarrow real$
⟨*proof*⟩

lemma *Bseq-eventually-mono*:
assumes $eventually\ (\lambda n.\ norm\ (f\ n) \le norm\ (g\ n))\ sequentially\ Bseq\ g$
shows $Bseq\ f$
⟨*proof*⟩

lemma *lemma-NBseq-def*: $(\exists K > 0.\ \forall n.\ norm\ (X\ n) \le K) \longleftrightarrow (\exists N.\ \forall n.\ norm\ (X\ n) \le real(Suc\ N))$
⟨*proof*⟩

Alternative definition for *Bseq*.

lemma *Bseq-iff*: $Bseq\ X \longleftrightarrow (\exists N.\ \forall n.\ norm\ (X\ n) \le real(Suc\ N))$
⟨*proof*⟩

lemma *lemma-NBseq-def2*: $(\exists K > 0.\ \forall n.\ norm\ (X\ n) \le K) = (\exists N.\ \forall n.\ norm\ (X\ n) < real(Suc\ N))$
⟨*proof*⟩

Yet another definition for Bseq.

lemma *Bseq-iff1a*: $Bseq\ X \longleftrightarrow (\exists N.\ \forall n.\ norm\ (X\ n) < real\ (Suc\ N))$
⟨*proof*⟩

101.1.3 A Few More Equivalence Theorems for Boundedness

Alternative formulation for boundedness.

lemma *Bseq-iff2*: $Bseq\ X \longleftrightarrow (\exists k > 0.\ \exists x.\ \forall n.\ norm\ (X\ n + -\ x) \le k)$
⟨*proof*⟩

Alternative formulation for boundedness.

lemma *Bseq-iff3*: $Bseq\ X \longleftrightarrow (\exists k{>}0.\ \exists N.\ \forall n.\ norm\ (X\ n + -\ X\ N) \le k)$

(**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *BseqI2*: $\forall n.\ k \leq f\ n \wedge f\ n \leq K \Longrightarrow$ *Bseq f*
for $k\ K$ :: *real*
⟨*proof*⟩

101.1.4 Upper Bounds and Lubs of Bounded Sequences

lemma *Bseq-minus-iff*: *Bseq* ($\lambda n.\ -\ (X\ n)$ :: *'a::real-normed-vector*) $\longleftrightarrow$ *Bseq X*
⟨*proof*⟩

lemma *Bseq-add*:
fixes f :: *nat* $\Rightarrow$ *'a::real-normed-vector*
assumes *Bseq f*
shows *Bseq* ($\lambda x.\ f\ x + c$)
⟨*proof*⟩

lemma *Bseq-add-iff*: *Bseq* ($\lambda x.\ f\ x + c$) $\longleftrightarrow$ *Bseq f*
for f :: *nat* $\Rightarrow$ *'a::real-normed-vector*
⟨*proof*⟩

lemma *Bseq-mult*:
fixes $f\ g$ :: *nat* $\Rightarrow$ *'a::real-normed-field*
assumes *Bseq f* **and** *Bseq g*
shows *Bseq* ($\lambda x.\ f\ x * g\ x$)
⟨*proof*⟩

lemma *Bfun-const* [*simp*]: *Bfun* (λ-. c) F
⟨*proof*⟩

lemma *Bseq-cmult-iff*:
fixes c :: *'a::real-normed-field*
assumes $c \neq 0$
shows *Bseq* ($\lambda x.\ c * f\ x$) $\longleftrightarrow$ *Bseq f*
⟨*proof*⟩

lemma *Bseq-subseq*: *Bseq f* $\Longrightarrow$ *Bseq* ($\lambda x.\ f\ (g\ x)$)
for f :: *nat* $\Rightarrow$ *'a::real-normed-vector*
⟨*proof*⟩

lemma *Bseq-Suc-iff*: *Bseq* ($\lambda n.\ f\ (Suc\ n)$) $\longleftrightarrow$ *Bseq f*
for f :: *nat* $\Rightarrow$ *'a::real-normed-vector*
⟨*proof*⟩

lemma *increasing-Bseq-subseq-iff*:
assumes $\bigwedge x\ y.\ x \leq y \Longrightarrow$ *norm* ($f\ x$ :: *'a::real-normed-vector*) $\leq$ *norm* ($f\ y$)
strict-mono g
shows *Bseq* ($\lambda x.\ f\ (g\ x)$) $\longleftrightarrow$ *Bseq f*

⟨*proof*⟩

lemma *nonneg-incseq-Bseq-subseq-iff*:
fixes f :: *nat* ⇒ *real*
and g :: *nat* ⇒ *nat*
assumes $\bigwedge x.\ f\ x \geq 0$ *incseq f strict-mono g*
shows *Bseq* $(\lambda x.\ f\ (g\ x)) \longleftrightarrow$ *Bseq f*
⟨*proof*⟩

lemma *Bseq-eq-bounded*: *range f* $\subseteq \{a..b\} \Longrightarrow$ *Bseq f*
for $a\ b$ :: *real*
⟨*proof*⟩

lemma *incseq-bounded*: *incseq X* $\Longrightarrow \forall\, i.\ X\ i \leq B \Longrightarrow$ *Bseq X*
for B :: *real*
⟨*proof*⟩

lemma *decseq-bounded*: *decseq X* $\Longrightarrow \forall\, i.\ B \leq X\ i \Longrightarrow$ *Bseq X*
for B :: *real*
⟨*proof*⟩

101.2 Bounded Monotonic Sequences

101.2.1 A Bounded and Monotonic Sequence Converges

lemma *Bmonoseq-LIMSEQ*: $\forall\, n.\ m \leq n \longrightarrow X\ n = X\ m \Longrightarrow \exists\, L.\ X \longrightarrow L$
⟨*proof*⟩

101.3 Convergence to Zero

definition *Zfun* :: $('a \Rightarrow 'b$::*real-normed-vector*$) \Rightarrow$ *'a filter* ⇒ *bool*
where *Zfun f F* = $(\forall\, r{>}0.$ *eventually* $(\lambda x.$ *norm* $(f\ x) < r)\ F)$

lemma *ZfunI*: $(\bigwedge r.\ 0 < r \Longrightarrow$ *eventually* $(\lambda x.$ *norm* $(f\ x) < r)\ F) \Longrightarrow$ *Zfun f F*
⟨*proof*⟩

lemma *ZfunD*: *Zfun f F* $\Longrightarrow 0 < r \Longrightarrow$ *eventually* $(\lambda x.$ *norm* $(f\ x) < r)\ F$
⟨*proof*⟩

lemma *Zfun-ssubst*: *eventually* $(\lambda x.\ f\ x = g\ x)\ F \Longrightarrow$ *Zfun g F* $\Longrightarrow$ *Zfun f F*
⟨*proof*⟩

lemma *Zfun-zero*: *Zfun* $(\lambda x.\ 0)\ F$
⟨*proof*⟩

lemma *Zfun-norm-iff*: *Zfun* $(\lambda x.$ *norm* $(f\ x))\ F =$ *Zfun* $(\lambda x.\ f\ x)\ F$
⟨*proof*⟩

lemma *Zfun-imp-Zfun*:
assumes f: *Zfun f F*

and g: *eventually* $(\lambda x.\ norm\ (g\ x) \leq norm\ (f\ x) * K)\ F$
shows *Zfun* $(\lambda x.\ g\ x)\ F$
⟨*proof*⟩

lemma *Zfun-le*: *Zfun* $g\ F \Longrightarrow \forall x.\ norm\ (f\ x) \leq norm\ (g\ x) \Longrightarrow Zfun\ f\ F$
⟨*proof*⟩

lemma *Zfun-add*:
assumes f: *Zfun* $f\ F$
and g: *Zfun* $g\ F$
shows *Zfun* $(\lambda x.\ f\ x + g\ x)\ F$
⟨*proof*⟩

lemma *Zfun-minus*: *Zfun* $f\ F \Longrightarrow Zfun\ (\lambda x.\ - f\ x)\ F$
⟨*proof*⟩

lemma *Zfun-diff*: *Zfun* $f\ F \Longrightarrow Zfun\ g\ F \Longrightarrow Zfun\ (\lambda x.\ f\ x - g\ x)\ F$
⟨*proof*⟩

lemma (**in** *bounded-linear*) *Zfun*:
assumes g: *Zfun* $g\ F$
shows *Zfun* $(\lambda x.\ f\ (g\ x))\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *Zfun*:
assumes f: *Zfun* $f\ F$
and g: *Zfun* $g\ F$
shows *Zfun* $(\lambda x.\ f\ x ** g\ x)\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *Zfun-left*: *Zfun* $f\ F \Longrightarrow Zfun\ (\lambda x.\ f\ x ** a)\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *Zfun-right*: *Zfun* $f\ F \Longrightarrow Zfun\ (\lambda x.\ a ** f\ x)\ F$
⟨*proof*⟩

lemmas *Zfun-mult* = *bounded-bilinear.Zfun* [*OF bounded-bilinear-mult*]
lemmas *Zfun-mult-right* = *bounded-bilinear.Zfun-right* [*OF bounded-bilinear-mult*]
lemmas *Zfun-mult-left* = *bounded-bilinear.Zfun-left* [*OF bounded-bilinear-mult*]

lemma *tendsto-Zfun-iff*: $(f \longrightarrow a)\ F = Zfun\ (\lambda x.\ f\ x - a)\ F$
⟨*proof*⟩

lemma *tendsto-0-le*:
$(f \longrightarrow 0)\ F \Longrightarrow eventually\ (\lambda x.\ norm\ (g\ x) \leq norm\ (f\ x) * K)\ F \Longrightarrow (g \longrightarrow 0)\ F$
⟨*proof*⟩

101.3.1 Distance and norms

lemma *tendsto-dist* [*tendsto-intros*]:
fixes *l m* :: *'a::metric-space*
assumes *f*: $(f \longrightarrow l)\ F$
and *g*: $(g \longrightarrow m)\ F$
shows $((\lambda x.\ dist\ (f\ x)\ (g\ x)) \longrightarrow dist\ l\ m)\ F$
⟨*proof*⟩

lemma *continuous-dist*[*continuous-intros*]:
fixes *f g* :: *- ⇒ 'a :: metric-space*
shows $continuous\ F\ f \Longrightarrow continuous\ F\ g \Longrightarrow continuous\ F\ (\lambda x.\ dist\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *continuous-on-dist*[*continuous-intros*]:
fixes *f g* :: *- ⇒ 'a :: metric-space*
shows $continuous\text{-}on\ s\ f \Longrightarrow continuous\text{-}on\ s\ g \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ dist\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *tendsto-norm* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ norm\ (f\ x)) \longrightarrow norm\ a)\ F$
⟨*proof*⟩

lemma *continuous-norm* [*continuous-intros*]: $continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ norm\ (f\ x))$
⟨*proof*⟩

lemma *continuous-on-norm* [*continuous-intros*]:
$continuous\text{-}on\ s\ f \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ norm\ (f\ x))$
⟨*proof*⟩

lemma *tendsto-norm-zero*: $(f \longrightarrow 0)\ F \Longrightarrow ((\lambda x.\ norm\ (f\ x)) \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-norm-zero-cancel*: $((\lambda x.\ norm\ (f\ x)) \longrightarrow 0)\ F \Longrightarrow (f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-norm-zero-iff*: $((\lambda x.\ norm\ (f\ x)) \longrightarrow 0)\ F \longleftrightarrow (f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-rabs* [*tendsto-intros*]: $(f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ |f\ x|) \longrightarrow |l|)\ F$
for *l* :: *real*
⟨*proof*⟩

lemma *continuous-rabs* [*continuous-intros*]:
$continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ |f\ x :: real|)$
⟨*proof*⟩

lemma *continuous-on-rabs* [*continuous-intros*]:
continuous-on s f $\Longrightarrow$ *continuous-on s* ($\lambda x.\ |f\ x :: real|$)
⟨*proof*⟩

lemma *tendsto-rabs-zero*: ($f \longrightarrow (0::real)$) $F \Longrightarrow ((\lambda x.\ |f\ x|) \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-rabs-zero-cancel*: $((\lambda x.\ |f\ x|) \longrightarrow (0::real))\ F \Longrightarrow (f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-rabs-zero-iff*: $((\lambda x.\ |f\ x|) \longrightarrow (0::real))\ F \longleftrightarrow (f \longrightarrow 0)\ F$
⟨*proof*⟩

101.4 Topological Monoid

class *topological-monoid-add* = *topological-space* + *monoid-add* +
assumes *tendsto-add-Pair*: *LIM x* (*nhds a* $\times_F$ *nhds b*). *fst x* + *snd x* :> *nhds* ($a + b$)

class *topological-comm-monoid-add* = *topological-monoid-add* + *comm-monoid-add*

lemma *tendsto-add* [*tendsto-intros*]:
fixes $a\ b :: 'a$::*topological-monoid-add*
shows $(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow ((\lambda x.\ f\ x + g\ x) \longrightarrow a + b)\ F$
⟨*proof*⟩

lemma *continuous-add* [*continuous-intros*]:
fixes $f\ g$ :: - ⇒ $'b$::*topological-monoid-add*
shows *continuous F f* $\Longrightarrow$ *continuous F g* $\Longrightarrow$ *continuous F* ($\lambda x.\ f\ x + g\ x$)
⟨*proof*⟩

lemma *continuous-on-add* [*continuous-intros*]:
fixes $f\ g$ :: - ⇒ $'b$::*topological-monoid-add*
shows *continuous-on s f* $\Longrightarrow$ *continuous-on s g* $\Longrightarrow$ *continuous-on s* ($\lambda x.\ f\ x + g\ x$)
⟨*proof*⟩

lemma *tendsto-add-zero*:
fixes $f\ g$ :: - ⇒ $'b$::*topological-monoid-add*
shows $(f \longrightarrow 0)\ F \Longrightarrow (g \longrightarrow 0)\ F \Longrightarrow ((\lambda x.\ f\ x + g\ x) \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-sum* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$::*topological-comm-monoid-add*
shows $(\bigwedge i.\ i \in I \Longrightarrow (f\ i \longrightarrow a\ i)\ F) \Longrightarrow ((\lambda x.\ \sum i{\in}I.\ f\ i\ x) \longrightarrow (\sum i{\in}I.\ a\ i))\ F$
⟨*proof*⟩

lemma *continuous-sum* [*continuous-intros*]:
fixes f :: $'a \Rightarrow 'b$::*t2-space* $\Rightarrow 'c$::*topological-comm-monoid-add*
shows $(\bigwedge i.\ i \in I \Longrightarrow$ *continuous* $F\ (f\ i)) \Longrightarrow$ *continuous* $F\ (\lambda x.\ \sum i \in I.\ f\ i\ x)$
⟨*proof*⟩

lemma *continuous-on-sum* [*continuous-intros*]:
fixes f :: $'a \Rightarrow 'b$::*topological-space* $\Rightarrow 'c$::*topological-comm-monoid-add*
shows $(\bigwedge i.\ i \in I \Longrightarrow$ *continuous-on* $S\ (f\ i)) \Longrightarrow$ *continuous-on* $S\ (\lambda x.\ \sum i \in I.\ f\ i\ x)$
⟨*proof*⟩

instance *nat* :: *topological-comm-monoid-add*
⟨*proof*⟩

instance *int* :: *topological-comm-monoid-add*
⟨*proof*⟩

101.4.1 Topological group

class *topological-group-add* = *topological-monoid-add* + *group-add* +
assumes *tendsto-uminus-nhds*: (*uminus* $\longrightarrow -\ a$) (*nhds* a)
begin

lemma *tendsto-minus* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ -\ f\ x) \longrightarrow -\ a)\ F$
⟨*proof*⟩

end

class *topological-ab-group-add* = *topological-group-add* + *ab-group-add*

instance *topological-ab-group-add* < *topological-comm-monoid-add* ⟨*proof*⟩

lemma *continuous-minus* [*continuous-intros*]: *continuous* $F\ f \Longrightarrow$ *continuous* $F\ (\lambda x.\ -\ f\ x)$
for f :: $'a$::*t2-space* $\Rightarrow 'b$::*topological-group-add*
⟨*proof*⟩

lemma *continuous-on-minus* [*continuous-intros*]: *continuous-on* $s\ f \Longrightarrow$ *continuous-on* $s\ (\lambda x.\ -\ f\ x)$
for f :: - $\Rightarrow 'b$::*topological-group-add*
⟨*proof*⟩

lemma *tendsto-minus-cancel*: $((\lambda x.\ -\ f\ x) \longrightarrow -\ a)\ F \Longrightarrow (f \longrightarrow a)\ F$
for a :: $'a$::*topological-group-add*
⟨*proof*⟩

lemma *tendsto-minus-cancel-left*:
$(f \longrightarrow -\ (y$::-::*topological-group-add*$))\ F \longleftrightarrow ((\lambda x.\ -\ f\ x) \longrightarrow y)\ F$

⟨*proof*⟩

lemma *tendsto-diff* [*tendsto-intros*]:
fixes *a b* :: *'a::topological-group-add*
shows $(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow ((\lambda x.\ f\ x - g\ x) \longrightarrow a - b)\ F$
⟨*proof*⟩

lemma *continuous-diff* [*continuous-intros*]:
fixes *f g* :: *'a::t2-space* ⇒ *'b::topological-group-add*
shows *continuous F f* ⟹ *continuous F g* ⟹ *continuous F* $(\lambda x.\ f\ x - g\ x)$
⟨*proof*⟩

lemma *continuous-on-diff* [*continuous-intros*]:
fixes *f g* :: *-* ⇒ *'b::topological-group-add*
shows *continuous-on s f* ⟹ *continuous-on s g* ⟹ *continuous-on s* $(\lambda x.\ f\ x - g\ x)$
⟨*proof*⟩

lemma *continuous-on-op-minus*: *continuous-on* (*s::'a::topological-group-add set*) (*op* − *x*)
⟨*proof*⟩

instance *real-normed-vector* < *topological-ab-group-add*
⟨*proof*⟩

lemmas *real-tendsto-sandwich* = *tendsto-sandwich*[**where** *'a*=*real*]

101.4.2 Linear operators and multiplication

lemma *linear-times*: *linear* $(\lambda x.\ c * x)$
for *c* :: *'a::real-algebra*
⟨*proof*⟩

lemma (**in** *bounded-linear*) *tendsto*: $(g \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ f\ (g\ x)) \longrightarrow f\ a)\ F$
⟨*proof*⟩

lemma (**in** *bounded-linear*) *continuous*: *continuous F g* ⟹ *continuous F* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma (**in** *bounded-linear*) *continuous-on*: *continuous-on s g* ⟹ *continuous-on s* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma (**in** *bounded-linear*) *tendsto-zero*: $(g \longrightarrow 0)\ F \Longrightarrow ((\lambda x.\ f\ (g\ x)) \longrightarrow 0)\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto*:
$(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow ((\lambda x.\ f\ x\ {**}\ g\ x) \longrightarrow a\ {**}\ b)\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *continuous*:
continuous $F\ f \Longrightarrow$ *continuous* $F\ g \Longrightarrow$ *continuous* $F\ (\lambda x.\ f\ x\ {**}\ g\ x)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *continuous-on*:
continuous-on $s\ f \Longrightarrow$ *continuous-on* $s\ g \Longrightarrow$ *continuous-on* $s\ (\lambda x.\ f\ x\ {**}\ g\ x)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto-zero*:
assumes f: $(f \longrightarrow 0)\ F$
and g: $(g \longrightarrow 0)\ F$
shows $((\lambda x.\ f\ x\ {**}\ g\ x) \longrightarrow 0)\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto-left-zero*:
$(f \longrightarrow 0)\ F \Longrightarrow ((\lambda x.\ f\ x\ {**}\ c) \longrightarrow 0)\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *tendsto-right-zero*:
$(f \longrightarrow 0)\ F \Longrightarrow ((\lambda x.\ c\ {**}\ f\ x) \longrightarrow 0)\ F$
⟨*proof*⟩

lemmas *tendsto-of-real* [*tendsto-intros*] =
bounded-linear.tendsto [*OF bounded-linear-of-real*]

lemmas *tendsto-scaleR* [*tendsto-intros*] =
bounded-bilinear.tendsto [*OF bounded-bilinear-scaleR*]

lemmas *tendsto-mult* [*tendsto-intros*] =
bounded-bilinear.tendsto [*OF bounded-bilinear-mult*]

lemma *tendsto-mult-left*: $(f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ c * (f\ x)) \longrightarrow c * l)\ F$
for c :: *'a::real-normed-algebra*
⟨*proof*⟩

lemma *tendsto-mult-right*: $(f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ (f\ x) * c) \longrightarrow l * c)\ F$
for c :: *'a::real-normed-algebra*
⟨*proof*⟩

lemmas *continuous-of-real* [*continuous-intros*] =
bounded-linear.continuous [*OF bounded-linear-of-real*]

lemmas *continuous-scaleR* [*continuous-intros*] =
bounded-bilinear.continuous [*OF bounded-bilinear-scaleR*]

lemmas *continuous-mult* [*continuous-intros*] =
bounded-bilinear.continuous [*OF bounded-bilinear-mult*]

lemmas *continuous-on-of-real* [*continuous-intros*] =
bounded-linear.continuous-on [*OF bounded-linear-of-real*]

lemmas *continuous-on-scaleR* [*continuous-intros*] =
bounded-bilinear.continuous-on [*OF bounded-bilinear-scaleR*]

lemmas *continuous-on-mult* [*continuous-intros*] =
bounded-bilinear.continuous-on [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-zero* =
bounded-bilinear.tendsto-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-left-zero* =
bounded-bilinear.tendsto-left-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-right-zero* =
bounded-bilinear.tendsto-right-zero [*OF bounded-bilinear-mult*]

lemma *tendsto-power* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ f\ x\ \hat{}\ n) \longrightarrow a\ \hat{}\ n)\ F$
for f :: $'a \Rightarrow\ 'b$::{*power*,*real-normed-algebra*}
⟨*proof*⟩

lemma *tendsto-null-power*: $[\![(f \longrightarrow 0)\ F;\ 0 < n]\!] \Longrightarrow ((\lambda x.\ f\ x\ \hat{}\ n) \longrightarrow 0)\ F$
for f :: $'a \Rightarrow\ 'b$::{*power*,*real-normed-algebra-1*}
⟨*proof*⟩

lemma *continuous-power* [*continuous-intros*]: *continuous* $F\ f \Longrightarrow$ *continuous* F $(\lambda x.\ (f\ x)\hat{}n)$
for f :: $'a$::*t2-space* $\Rightarrow\ 'b$::{*power*,*real-normed-algebra*}
⟨*proof*⟩

lemma *continuous-on-power* [*continuous-intros*]:
fixes f :: $- \Rightarrow\ 'b$::{*power*,*real-normed-algebra*}
shows *continuous-on* $s\ f \Longrightarrow$ *continuous-on* $s\ (\lambda x.\ (f\ x)\hat{}n)$
⟨*proof*⟩

lemma *tendsto-prod* [*tendsto-intros*]:
fixes f :: $'a \Rightarrow\ 'b \Rightarrow\ 'c$::{*real-normed-algebra*,*comm-ring-1*}
shows $(\bigwedge i.\ i \in S \Longrightarrow (f\ i \longrightarrow L\ i)\ F) \Longrightarrow ((\lambda x.\ \prod i{\in}S.\ f\ i\ x) \longrightarrow (\prod i{\in}S.\ L\ i))\ F$
⟨*proof*⟩

lemma *continuous-prod* [*continuous-intros*]:
fixes f :: $'a \Rightarrow\ 'b$::*t2-space* $\Rightarrow\ 'c$::{*real-normed-algebra*,*comm-ring-1*}
shows $(\bigwedge i.\ i \in S \Longrightarrow$ *continuous* $F\ (f\ i)) \Longrightarrow$ *continuous* $F\ (\lambda x.\ \prod i{\in}S.\ f\ i\ x)$

⟨*proof*⟩

lemma *continuous-on-prod* [*continuous-intros*]:
fixes f :: $'a \Rightarrow$ - $\Rightarrow$ $'c$::{*real-normed-algebra*,*comm-ring-1*}
shows $(\bigwedge i.\ i \in S \Longrightarrow \textit{continuous-on}\ s\ (f\ i)) \Longrightarrow \textit{continuous-on}\ s\ (\lambda x.\ \prod i{\in}S.\ f\ i\ x)$
⟨*proof*⟩

lemma *tendsto-of-real-iff*:
$((\lambda x.\ \textit{of-real}\ (f\ x) :: 'a{::}\textit{real-normed-div-algebra}) \longrightarrow \textit{of-real}\ c)\ F \longleftrightarrow (f \longrightarrow c)\ F$
⟨*proof*⟩

lemma *tendsto-add-const-iff*:
$((\lambda x.\ c + f\ x :: 'a{::}\textit{real-normed-vector}) \longrightarrow c + d)\ F \longleftrightarrow (f \longrightarrow d)\ F$
⟨*proof*⟩

101.4.3 Inverse and division

lemma (**in** *bounded-bilinear*) *Zfun-prod-Bfun*:
assumes f: *Zfun f F*
and g: *Bfun g F*
shows $\textit{Zfun}\ (\lambda x.\ f\ x ** g\ x)\ F$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *Bfun-prod-Zfun*:
assumes f: *Bfun f F*
and g: *Zfun g F*
shows $\textit{Zfun}\ (\lambda x.\ f\ x ** g\ x)\ F$
⟨*proof*⟩

lemma *Bfun-inverse-lemma*:
fixes x :: $'a$::*real-normed-div-algebra*
shows $r \le \textit{norm}\ x \Longrightarrow 0 < r \Longrightarrow \textit{norm}\ (\textit{inverse}\ x) \le \textit{inverse}\ r$
⟨*proof*⟩

lemma *Bfun-inverse*:
fixes a :: $'a$::*real-normed-div-algebra*
assumes f: $(f \longrightarrow a)\ F$
assumes a: $a \neq 0$
shows $\textit{Bfun}\ (\lambda x.\ \textit{inverse}\ (f\ x))\ F$
⟨*proof*⟩

lemma *tendsto-inverse* [*tendsto-intros*]:
fixes a :: $'a$::*real-normed-div-algebra*
assumes f: $(f \longrightarrow a)\ F$
and a: $a \neq 0$
shows $((\lambda x.\ \textit{inverse}\ (f\ x)) \longrightarrow \textit{inverse}\ a)\ F$
⟨*proof*⟩

lemma *continuous-inverse*:
fixes f :: $'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-div-algebra*
assumes *continuous* F f
and f $(Lim\ F\ (\lambda x.\ x)) \neq 0$
shows *continuous* F $(\lambda x.\ inverse\ (f\ x))$
⟨*proof*⟩

lemma *continuous-at-within-inverse*[*continuous-intros*]:
fixes f :: $'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-div-algebra*
assumes *continuous* $(at\ a\ within\ s)$ f
and $f\ a \neq 0$
shows *continuous* $(at\ a\ within\ s)$ $(\lambda x.\ inverse\ (f\ x))$
⟨*proof*⟩

lemma *isCont-inverse*[*continuous-intros*, *simp*]:
fixes f :: $'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-div-algebra*
assumes *isCont* f a
and $f\ a \neq 0$
shows *isCont* $(\lambda x.\ inverse\ (f\ x))$ a
⟨*proof*⟩

lemma *continuous-on-inverse*[*continuous-intros*]:
fixes f :: $'a$::*topological-space* $\Rightarrow$ $'b$::*real-normed-div-algebra*
assumes *continuous-on* s f
and $\forall x \in s.\ f\ x \neq 0$
shows *continuous-on* s $(\lambda x.\ inverse\ (f\ x))$
⟨*proof*⟩

lemma *tendsto-divide* [*tendsto-intros*]:
fixes $a\ b$ :: $'a$::*real-normed-field*
shows $(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow b \neq 0 \Longrightarrow ((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow a\ /\ b)\ F$
⟨*proof*⟩

lemma *continuous-divide*:
fixes $f\ g$ :: $'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-field*
assumes *continuous* F f
and *continuous* F g
and g $(Lim\ F\ (\lambda x.\ x)) \neq 0$
shows *continuous* F $(\lambda x.\ (f\ x)\ /\ (g\ x))$
⟨*proof*⟩

lemma *continuous-at-within-divide*[*continuous-intros*]:
fixes $f\ g$ :: $'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-field*
assumes *continuous* $(at\ a\ within\ s)$ f *continuous* $(at\ a\ within\ s)$ g
and $g\ a \neq 0$
shows *continuous* $(at\ a\ within\ s)$ $(\lambda x.\ (f\ x)\ /\ (g\ x))$
⟨*proof*⟩

lemma *isCont-divide*[*continuous-intros*, *simp*]:
fixes $f\ g$ :: $'a$::*t2-space* ⇒ $'b$::*real-normed-field*
assumes *isCont f a isCont g a g a* $\neq 0$
shows *isCont* $(\lambda x.\ (f\ x)\ /\ g\ x)\ a$
⟨*proof*⟩

lemma *continuous-on-divide*[*continuous-intros*]:
fixes f :: $'a$::*topological-space* ⇒ $'b$::*real-normed-field*
assumes *continuous-on s f continuous-on s g*
and $\forall x \in s.\ g\ x \neq 0$
shows *continuous-on s* $(\lambda x.\ (f\ x)\ /\ (g\ x))$
⟨*proof*⟩

lemma *tendsto-sgn* [*tendsto-intros*]: $(f \longrightarrow l)\ F \Longrightarrow l \neq 0 \Longrightarrow ((\lambda x.\ sgn\ (f\ x)) \longrightarrow sgn\ l)\ F$
for l :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *continuous-sgn*:
fixes f :: $'a$::*t2-space* ⇒ $'b$::*real-normed-vector*
assumes *continuous F f*
and $f\ (Lim\ F\ (\lambda x.\ x)) \neq 0$
shows *continuous F* $(\lambda x.\ sgn\ (f\ x))$
⟨*proof*⟩

lemma *continuous-at-within-sgn*[*continuous-intros*]:
fixes f :: $'a$::*t2-space* ⇒ $'b$::*real-normed-vector*
assumes *continuous* (*at a within s*) *f*
and $f\ a \neq 0$
shows *continuous* (*at a within s*) $(\lambda x.\ sgn\ (f\ x))$
⟨*proof*⟩

lemma *isCont-sgn*[*continuous-intros*]:
fixes f :: $'a$::*t2-space* ⇒ $'b$::*real-normed-vector*
assumes *isCont f a*
and $f\ a \neq 0$
shows *isCont* $(\lambda x.\ sgn\ (f\ x))\ a$
⟨*proof*⟩

lemma *continuous-on-sgn*[*continuous-intros*]:
fixes f :: $'a$::*topological-space* ⇒ $'b$::*real-normed-vector*
assumes *continuous-on s f*
and $\forall x \in s.\ f\ x \neq 0$
shows *continuous-on s* $(\lambda x.\ sgn\ (f\ x))$
⟨*proof*⟩

lemma *filterlim-at-infinity*:
fixes f :: - ⇒ $'a$::*real-normed-vector*

assumes $0 \leq c$
shows $(LIM\ x\ F.\ f\ x :> at\text{-}infinity) \longleftrightarrow (\forall r>c.\ eventually\ (\lambda x.\ r \leq norm\ (f\ x))\ F)$
⟨*proof*⟩

lemma *not-tendsto-and-filterlim-at-infinity*:
fixes $c :: 'a::real\text{-}normed\text{-}vector$
assumes $F \neq bot$
and $(f \longrightarrow c)\ F$
and *filterlim f at-infinity F*
shows *False*
⟨*proof*⟩

lemma *filterlim-at-infinity-imp-not-convergent*:
assumes *filterlim f at-infinity sequentially*
shows $\neg$ *convergent f*
⟨*proof*⟩

lemma *filterlim-at-infinity-imp-eventually-ne*:
assumes *filterlim f at-infinity F*
shows $eventually\ (\lambda z.\ f\ z \neq c)\ F$
⟨*proof*⟩

lemma *tendsto-of-nat* [*tendsto-intros*]:
$filterlim\ (of\text{-}nat :: nat \Rightarrow 'a::real\text{-}normed\text{-}algebra\text{-}1)\ at\text{-}infinity\ sequentially$
⟨*proof*⟩

101.5 Relate *at*, *at-left* and *at-right*

This lemmas are useful for conversion between *at x* to *at-left x* and *at-right x* and also *at-right* $(0::'a)$.

lemmas *filterlim-split-at-real* = *filterlim-split-at*[**where** *'a=real*]

lemma *filtermap-nhds-shift*: $filtermap\ (\lambda x.\ x - d)\ (nhds\ a) = nhds\ (a - d)$
for $a\ d :: 'a::real\text{-}normed\text{-}vector$
⟨*proof*⟩

lemma *filtermap-nhds-minus*: $filtermap\ (\lambda x.\ - x)\ (nhds\ a) = nhds\ (- a)$
for $a :: 'a::real\text{-}normed\text{-}vector$
⟨*proof*⟩

lemma *filtermap-at-shift*: $filtermap\ (\lambda x.\ x - d)\ (at\ a) = at\ (a - d)$
for $a\ d :: 'a::real\text{-}normed\text{-}vector$
⟨*proof*⟩

lemma *filtermap-at-right-shift*: $filtermap\ (\lambda x.\ x - d)\ (at\text{-}right\ a) = at\text{-}right\ (a - d)$
for $a\ d :: real$
⟨*proof*⟩

lemma *at-right-to-0*: *at-right a* = *filtermap* (λ*x*. *x* + *a*) (*at-right 0*)
  for *a* :: *real*
  ⟨*proof*⟩

lemma *filterlim-at-right-to-0*:
  filterlim f F (*at-right a*) ⟷ *filterlim* (λ*x*. *f* (*x* + *a*)) *F* (*at-right 0*)
  for *a* :: *real*
  ⟨*proof*⟩

lemma *eventually-at-right-to-0*:
  eventually P (*at-right a*) ⟷ *eventually* (λ*x*. *P* (*x* + *a*)) (*at-right 0*)
  for *a* :: *real*
  ⟨*proof*⟩

lemma *filtermap-at-minus*: *filtermap* (λ*x*. − *x*) (*at a*) = *at* (− *a*)
  for *a* :: *'a::real-normed-vector*
  ⟨*proof*⟩

lemma *at-left-minus*: *at-left a* = *filtermap* (λ*x*. − *x*) (*at-right* (− *a*))
  for *a* :: *real*
  ⟨*proof*⟩

lemma *at-right-minus*: *at-right a* = *filtermap* (λ*x*. − *x*) (*at-left* (− *a*))
  for *a* :: *real*
  ⟨*proof*⟩

lemma *filterlim-at-left-to-right*:
  filterlim f F (*at-left a*) ⟷ *filterlim* (λ*x*. *f* (− *x*)) *F* (*at-right* (−*a*))
  for *a* :: *real*
  ⟨*proof*⟩

lemma *eventually-at-left-to-right*:
  eventually P (*at-left a*) ⟷ *eventually* (λ*x*. *P* (− *x*)) (*at-right* (−*a*))
  for *a* :: *real*
  ⟨*proof*⟩

lemma *filterlim-uminus-at-top-at-bot*: *LIM x at-bot*. − *x* :: *real* :> *at-top*
  ⟨*proof*⟩

lemma *filterlim-uminus-at-bot-at-top*: *LIM x at-top*. − *x* :: *real* :> *at-bot*
  ⟨*proof*⟩

lemma *at-top-mirror*: *at-top* = *filtermap uminus* (*at-bot* :: *real filter*)
  ⟨*proof*⟩

lemma *at-bot-mirror*: *at-bot* = *filtermap uminus* (*at-top* :: *real filter*)
  ⟨*proof*⟩

lemma *filterlim-at-top-mirror*: $(LIM\ x\ at\text{-}top.\ f\ x :> F) \longleftrightarrow (LIM\ x\ at\text{-}bot.\ f\ (-x::real) :> F)$
⟨*proof*⟩

lemma *filterlim-at-bot-mirror*: $(LIM\ x\ at\text{-}bot.\ f\ x :> F) \longleftrightarrow (LIM\ x\ at\text{-}top.\ f\ (-x::real) :> F)$
⟨*proof*⟩

lemma *filterlim-uminus-at-top*: $(LIM\ x\ F.\ f\ x :> at\text{-}top) \longleftrightarrow (LIM\ x\ F.\ -\ (f\ x) :: real :> at\text{-}bot)$
⟨*proof*⟩

lemma *filterlim-uminus-at-bot*: $(LIM\ x\ F.\ f\ x :> at\text{-}bot) \longleftrightarrow (LIM\ x\ F.\ -\ (f\ x) :: real :> at\text{-}top)$
⟨*proof*⟩

lemma *filterlim-inverse-at-top-right*: $LIM\ x\ at\text{-}right\ (0::real).\ inverse\ x :> at\text{-}top$
⟨*proof*⟩

lemma *tendsto-inverse-0*:
fixes $x :: \text{-} \Rightarrow 'a::real\text{-}normed\text{-}div\text{-}algebra$
shows $(inverse \longrightarrow (0::'a))\ at\text{-}infinity$
⟨*proof*⟩

lemma *tendsto-add-filterlim-at-infinity*:
fixes $c :: 'b::real\text{-}normed\text{-}vector$
and $F :: 'a\ filter$
assumes $(f \longrightarrow c)\ F$
and *filterlim g at-infinity F*
shows *filterlim* $(\lambda x.\ f\ x + g\ x)$ *at-infinity F*
⟨*proof*⟩

lemma *tendsto-add-filterlim-at-infinity'*:
fixes $c :: 'b::real\text{-}normed\text{-}vector$
and $F :: 'a\ filter$
assumes *filterlim f at-infinity F*
and $(g \longrightarrow c)\ F$
shows *filterlim* $(\lambda x.\ f\ x + g\ x)$ *at-infinity F*
⟨*proof*⟩

lemma *filterlim-inverse-at-right-top*: $LIM\ x\ at\text{-}top.\ inverse\ x :> at\text{-}right\ (0::real)$
⟨*proof*⟩

lemma *filterlim-inverse-at-top*:
$(f \longrightarrow (0 :: real))\ F \Longrightarrow eventually\ (\lambda x.\ 0 < f\ x)\ F \Longrightarrow LIM\ x\ F.\ inverse\ (f\ x) :> at\text{-}top$
⟨*proof*⟩

lemma *filterlim-inverse-at-bot-neg*:

$LIM\ x\ (at\text{-}left\ (0::real)).\ inverse\ x :> at\text{-}bot$
⟨proof⟩

lemma *filterlim-inverse-at-bot*:
$(f \longrightarrow (0 :: real))\ F \Longrightarrow eventually\ (\lambda x.\ f\ x < 0)\ F \Longrightarrow LIM\ x\ F.\ inverse\ (f\ x) :> at\text{-}bot$
⟨proof⟩

lemma *at-right-to-top*: $(at\text{-}right\ (0::real)) = filtermap\ inverse\ at\text{-}top$
⟨proof⟩

lemma *eventually-at-right-to-top*:
$eventually\ P\ (at\text{-}right\ (0::real)) \longleftrightarrow eventually\ (\lambda x.\ P\ (inverse\ x))\ at\text{-}top$
⟨proof⟩

lemma *filterlim-at-right-to-top*:
$filterlim\ f\ F\ (at\text{-}right\ (0::real)) \longleftrightarrow (LIM\ x\ at\text{-}top.\ f\ (inverse\ x) :> F)$
⟨proof⟩

lemma *at-top-to-right*: $at\text{-}top = filtermap\ inverse\ (at\text{-}right\ (0::real))$
⟨proof⟩

lemma *eventually-at-top-to-right*:
$eventually\ P\ at\text{-}top \longleftrightarrow eventually\ (\lambda x.\ P\ (inverse\ x))\ (at\text{-}right\ (0::real))$
⟨proof⟩

lemma *filterlim-at-top-to-right*:
$filterlim\ f\ F\ at\text{-}top \longleftrightarrow (LIM\ x\ (at\text{-}right\ (0::real)).\ f\ (inverse\ x) :> F)$
⟨proof⟩

lemma *filterlim-inverse-at-infinity*:
fixes $x :: - \Rightarrow 'a::\{real\text{-}normed\text{-}div\text{-}algebra,\ division\text{-}ring\}$
shows $filterlim\ inverse\ at\text{-}infinity\ (at\ (0::'a))$
⟨proof⟩

lemma *filterlim-inverse-at-iff*:
fixes $g :: 'a \Rightarrow 'b::\{real\text{-}normed\text{-}div\text{-}algebra,\ division\text{-}ring\}$
shows $(LIM\ x\ F.\ inverse\ (g\ x) :> at\ 0) \longleftrightarrow (LIM\ x\ F.\ g\ x :> at\text{-}infinity)$
⟨proof⟩

lemma *tendsto-mult-filterlim-at-infinity*:
fixes $c :: 'a::real\text{-}normed\text{-}field$
assumes $(f \longrightarrow c)\ F\ c \neq 0$
assumes $filterlim\ g\ at\text{-}infinity\ F$
shows $filterlim\ (\lambda x.\ f\ x * g\ x)\ at\text{-}infinity\ F$
⟨proof⟩

lemma *tendsto-inverse-0-at-top*: $LIM\ x\ F.\ f\ x :> at\text{-}top \Longrightarrow ((\lambda x.\ inverse\ (f\ x) :: real) \longrightarrow 0)\ F$

⟨*proof*⟩

lemma *real-tendsto-divide-at-top*:
fixes *c*::*real*
assumes $(f \longrightarrow c)\ F$
assumes *filterlim g at-top F*
shows $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *mult-nat-left-at-top*: $c > 0 \Longrightarrow$ *filterlim* $(\lambda x.\ c * x)$ *at-top sequentially*
for *c* :: *nat*
⟨*proof*⟩

lemma *mult-nat-right-at-top*: $c > 0 \Longrightarrow$ *filterlim* $(\lambda x.\ x * c)$ *at-top sequentially*
for *c* :: *nat*
⟨*proof*⟩

lemma *at-to-infinity*: (*at* (*0*::'*a*::{*real-normed-field*,*field*})) = *filtermap inverse at-infinity*
⟨*proof*⟩

lemma *lim-at-infinity-0*:
fixes *l* :: '*a*::{*real-normed-field*,*field*}
shows $(f \longrightarrow l)$ *at-infinity* $\longleftrightarrow$ $((f \circ inverse) \longrightarrow l)$ (*at* (*0*::'*a*))
⟨*proof*⟩

lemma *lim-zero-infinity*:
fixes *l* :: '*a*::{*real-normed-field*,*field*}
shows $((\lambda x.\ f(1\ /\ x)) \longrightarrow l)$ (*at* (*0*::'*a*)) $\Longrightarrow$ $(f \longrightarrow l)$ *at-infinity*
⟨*proof*⟩

We only show rules for multiplication and addition when the functions are either against a real value or against infinity. Further rules are easy to derive by using *filterlim ?f at-top ?F* = (*LIM x ?F*. − *?f x* :> *at-bot*).

lemma *filterlim-tendsto-pos-mult-at-top*:
assumes *f*: $(f \longrightarrow c)\ F$
and *c*: $0 < c$
and *g*: *LIM x F*. *g x* :> *at-top*
shows *LIM x F*. (*f x* ∗ *g x* :: *real*) :> *at-top*
⟨*proof*⟩

lemma *filterlim-at-top-mult-at-top*:
assumes *f*: *LIM x F*. *f x* :> *at-top*
and *g*: *LIM x F*. *g x* :> *at-top*
shows *LIM x F*. (*f x* ∗ *g x* :: *real*) :> *at-top*
⟨*proof*⟩

lemma *filterlim-at-top-mult-tendsto-pos*:
assumes *f*: $(f \longrightarrow c)\ F$
and *c*: $0 < c$

and g: $LIM\ x\ F.\ g\ x :> at\text{-}top$
shows $LIM\ x\ F.\ (g\ x * f\ x{::}\ real) :> at\text{-}top$
⟨*proof*⟩

lemma *filterlim-tendsto-pos-mult-at-bot*:
fixes $c :: real$
assumes $(f \longrightarrow c)\ F\ 0 < c\ filterlim\ g\ at\text{-}bot\ F$
shows $LIM\ x\ F.\ f\ x * g\ x :> at\text{-}bot$
⟨*proof*⟩

lemma *filterlim-tendsto-neg-mult-at-bot*:
fixes $c :: real$
assumes c: $(f \longrightarrow c)\ F\ c < 0$ **and** g: $filterlim\ g\ at\text{-}top\ F$
shows $LIM\ x\ F.\ f\ x * g\ x :> at\text{-}bot$
⟨*proof*⟩

lemma *filterlim-pow-at-top*:
fixes $f :: 'a \Rightarrow real$
assumes $0 < n$
and f: $LIM\ x\ F.\ f\ x :> at\text{-}top$
shows $LIM\ x\ F.\ (f\ x)\hat{}n :: real :> at\text{-}top$
⟨*proof*⟩

lemma *filterlim-pow-at-bot-even*:
fixes $f :: real \Rightarrow real$
shows $0 < n \Longrightarrow LIM\ x\ F.\ f\ x :> at\text{-}bot \Longrightarrow even\ n \Longrightarrow LIM\ x\ F.\ (f\ x)\hat{}n :> at\text{-}top$
⟨*proof*⟩

lemma *filterlim-pow-at-bot-odd*:
fixes $f :: real \Rightarrow real$
shows $0 < n \Longrightarrow LIM\ x\ F.\ f\ x :> at\text{-}bot \Longrightarrow odd\ n \Longrightarrow LIM\ x\ F.\ (f\ x)\hat{}n :> at\text{-}bot$
⟨*proof*⟩

lemma *filterlim-tendsto-add-at-top*:
assumes f: $(f \longrightarrow c)\ F$
and g: $LIM\ x\ F.\ g\ x :> at\text{-}top$
shows $LIM\ x\ F.\ (f\ x + g\ x :: real) :> at\text{-}top$
⟨*proof*⟩

lemma *LIM-at-top-divide*:
fixes $f\ g :: 'a \Rightarrow real$
assumes f: $(f \longrightarrow a)\ F\ 0 < a$
and g: $(g \longrightarrow 0)\ F\ eventually\ (\lambda x.\ 0 < g\ x)\ F$
shows $LIM\ x\ F.\ f\ x\ /\ g\ x :> at\text{-}top$
⟨*proof*⟩

lemma *filterlim-at-top-add-at-top*:

assumes f: *LIM* x F. f x :> *at-top*
and g: *LIM* x F. g x :> *at-top*
shows *LIM* x F. (f x + g x :: *real*) :> *at-top*
⟨*proof*⟩

lemma *tendsto-divide-0*:
fixes f :: - ⇒ 'a::{*real-normed-div-algebra*, *division-ring*}
assumes f: (f $\longrightarrow$ c) F
and g: *LIM* x F. g x :> *at-infinity*
shows ((λx. f x / g x) $\longrightarrow$ *0*) F
⟨*proof*⟩

lemma *linear-plus-1-le-power*:
fixes x :: *real*
assumes x: $0 \leq x$
shows *real* $n * x + 1 \leq (x + 1)$ ^ n
⟨*proof*⟩

lemma *filterlim-realpow-sequentially-gt1*:
fixes x :: 'a :: *real-normed-div-algebra*
assumes x[*arith*]: $1 <$ *norm* x
shows *LIM* n *sequentially*. x ^ n :> *at-infinity*
⟨*proof*⟩

lemma *filterlim-divide-at-infinity*:
fixes f g :: 'a ⇒ 'a :: *real-normed-field*
assumes *filterlim* f (*nhds* c) F *filterlim* g (*at* *0*) F $c \neq 0$
shows *filterlim* (λx. f x / g x) *at-infinity* F
⟨*proof*⟩

101.6 Floor and Ceiling

lemma *eventually-floor-less*:
fixes f :: 'a ⇒ 'b::{*order-topology*,*floor-ceiling*}
assumes f: (f $\longrightarrow$ l) F
and l: $l \notin \mathbb{Z}$
shows $\forall_F$ x *in* F. *of-int* (*floor* l) $<$ f x
⟨*proof*⟩

lemma *eventually-less-ceiling*:
fixes f :: 'a ⇒ 'b::{*order-topology*,*floor-ceiling*}
assumes f: (f $\longrightarrow$ l) F
and l: $l \notin \mathbb{Z}$
shows $\forall_F$ x *in* F. f x $<$ *of-int* (*ceiling* l)
⟨*proof*⟩

lemma *eventually-floor-eq*:
fixes f::'a ⇒ 'b::{*order-topology*,*floor-ceiling*}

assumes f: $(f \longrightarrow l)$ F
and l: $l \notin \mathbb{Z}$
shows $\forall_F$ x *in* F. *floor* $(f\ x)$ = *floor* l
⟨*proof*⟩

lemma *eventually-ceiling-eq*:
fixes f::$'a \Rightarrow$ $'b$::{*order-topology*,*floor-ceiling*}
assumes f: $(f \longrightarrow l)$ F
and l: $l \notin \mathbb{Z}$
shows $\forall_F$ x *in* F. *ceiling* $(f\ x)$ = *ceiling* l
⟨*proof*⟩

lemma *tendsto-of-int-floor*:
fixes f::$'a \Rightarrow$ $'b$::{*order-topology*,*floor-ceiling*}
assumes $(f \longrightarrow l)$ F
and $l \notin \mathbb{Z}$
shows ((λx. *of-int* (*floor* $(f\ x)$) :: $'c$::{*ring-1*,*topological-space*}) $\longrightarrow$ *of-int* (*floor* l)) F
⟨*proof*⟩

lemma *tendsto-of-int-ceiling*:
fixes f::$'a \Rightarrow$ $'b$::{*order-topology*,*floor-ceiling*}
assumes $(f \longrightarrow l)$ F
and $l \notin \mathbb{Z}$
shows ((λx. *of-int* (*ceiling* $(f\ x)$):: $'c$::{*ring-1*,*topological-space*}) $\longrightarrow$ *of-int* (*ceiling* l)) F
⟨*proof*⟩

lemma *continuous-on-of-int-floor*:
continuous-on (*UNIV* $-$ $\mathbb{Z}$::$'a$::{*order-topology*, *floor-ceiling*} *set*)
(λx. *of-int* (*floor* x)::$'b$::{*ring-1*, *topological-space*})
⟨*proof*⟩

lemma *continuous-on-of-int-ceiling*:
continuous-on (*UNIV* $-$ $\mathbb{Z}$::$'a$::{*order-topology*, *floor-ceiling*} *set*)
(λx. *of-int* (*ceiling* x)::$'b$::{*ring-1*, *topological-space*})
⟨*proof*⟩

101.7 Limits of Sequences

lemma [*trans*]: $X = Y \Longrightarrow Y \longrightarrow z \Longrightarrow X \longrightarrow z$
⟨*proof*⟩

lemma *LIMSEQ-iff*:
fixes L :: $'a$::*real-normed-vector*
shows $(X \longrightarrow L) = (\forall r>0.\ \exists no.\ \forall n \geq no.\ norm\ (X\ n - L) < r)$
⟨*proof*⟩

lemma *LIMSEQ-I*: $(\bigwedge r.\ 0 < r \Longrightarrow \exists no.\ \forall n \geq no.\ norm\ (X\ n - L) < r) \Longrightarrow X$

$\longrightarrow L$
for L :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *LIMSEQ-D*: $X \longrightarrow L \Longrightarrow 0 < r \Longrightarrow \exists no. \forall n \geq no.\ norm\ (X\ n - L) < r$
for L :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *LIMSEQ-linear*: $X \longrightarrow x \Longrightarrow l > 0 \Longrightarrow (\lambda\ n.\ X\ (n * l)) \longrightarrow x$
⟨*proof*⟩

lemma *norm-inverse-le-norm*: $r \leq norm\ x \Longrightarrow 0 < r \Longrightarrow norm\ (inverse\ x) \leq inverse\ r$
for x :: $'a$::*real-normed-div-algebra*
⟨*proof*⟩

lemma *Bseq-inverse*: $X \longrightarrow a \Longrightarrow a \neq 0 \Longrightarrow Bseq\ (\lambda n.\ inverse\ (X\ n))$
for a :: $'a$::*real-normed-div-algebra*
⟨*proof*⟩

Transformation of limit.

lemma *Lim-transform*: $(g \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ f\ x - g\ x) \longrightarrow 0)\ F \Longrightarrow (f \longrightarrow a)\ F$
for $a\ b$ :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *Lim-transform2*: $(f \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ f\ x - g\ x) \longrightarrow 0)\ F \Longrightarrow (g \longrightarrow a)\ F$
for $a\ b$ :: $'a$::*real-normed-vector*
⟨*proof*⟩

proposition *Lim-transform-eq*: $((\lambda x.\ f\ x - g\ x) \longrightarrow 0)\ F \Longrightarrow (f \longrightarrow a)\ F \longleftrightarrow (g \longrightarrow a)\ F$
for a :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *Lim-transform-eventually*:
eventually $(\lambda x.\ f\ x = g\ x)\ net \Longrightarrow (f \longrightarrow l)\ net \Longrightarrow (g \longrightarrow l)\ net$
⟨*proof*⟩

lemma *Lim-transform-within*:
assumes $(f \longrightarrow l)$ (*at x within S*)
and $0 < d$
and $\bigwedge x'.\ x' \in S \Longrightarrow 0 < dist\ x'\ x \Longrightarrow dist\ x'\ x < d \Longrightarrow f\ x' = g\ x'$
shows $(g \longrightarrow l)$ (*at x within S*)
⟨*proof*⟩

Common case assuming being away from some crucial point like 0.

lemma *Lim-transform-away-within*:
fixes a b :: $'a$::*t1-space*
assumes $a \neq b$
and $\forall x \in S.\ x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$
and $(f \longrightarrow l)$ (*at* a *within* S)
shows $(g \longrightarrow l)$ (*at* a *within* S)
⟨*proof*⟩

lemma *Lim-transform-away-at*:
fixes a b :: $'a$::*t1-space*
assumes *ab*: $a \neq b$
and *fg*: $\forall x.\ x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$
and *fl*: $(f \longrightarrow l)$ (*at* a)
shows $(g \longrightarrow l)$ (*at* a)
⟨*proof*⟩

Alternatively, within an open set.

lemma *Lim-transform-within-open*:
assumes $(f \longrightarrow l)$ (*at* a *within* T)
and *open* s **and** $a \in s$
and $\bigwedge x.\ x \in s \Longrightarrow x \neq a \Longrightarrow f\ x = g\ x$
shows $(g \longrightarrow l)$ (*at* a *within* T)
⟨*proof*⟩

A congruence rule allowing us to transform limits assuming not at point.

lemma *Lim-cong-within*:
assumes $a = b$
and $x = y$
and $S = T$
and $\bigwedge x.\ x \neq b \Longrightarrow x \in T \Longrightarrow f\ x = g\ x$
shows $(f \longrightarrow x)$ (*at* a *within* S) $\longleftrightarrow$ $(g \longrightarrow y)$ (*at* b *within* T)
⟨*proof*⟩

lemma *Lim-cong-at*:
assumes $a = b$ $x = y$
and $\bigwedge x.\ x \neq a \Longrightarrow f\ x = g\ x$
shows $((\lambda x.\ f\ x) \longrightarrow x)$ (*at* a) $\longleftrightarrow$ $((g \longrightarrow y)$ (*at* a))
⟨*proof*⟩

An unbounded sequence's inverse tends to 0.

lemma *LIMSEQ-inverse-zero*:
assumes $\bigwedge r$::*real*. $\exists N.\ \forall n \geq N.\ r < X\ n$
shows $(\lambda n.\ inverse\ (X\ n)) \longrightarrow 0$
⟨*proof*⟩

The sequence $(1::'a)$ / n tends to 0 as n tends to infinity.

lemma *LIMSEQ-inverse-real-of-nat*: $(\lambda n.\ inverse\ (real\ (Suc\ n))) \longrightarrow 0$
⟨*proof*⟩

The sequence $r + (1::'a) / n$ tends to r as n tends to infinity is now easily proved.

lemma *LIMSEQ-inverse-real-of-nat-add*: $(\lambda n.\ r + inverse\ (real\ (Suc\ n))) \longrightarrow r$
⟨*proof*⟩

lemma *LIMSEQ-inverse-real-of-nat-add-minus*: $(\lambda n.\ r + -inverse\ (real\ (Suc\ n))) \longrightarrow r$
⟨*proof*⟩

lemma *LIMSEQ-inverse-real-of-nat-add-minus-mult*: $(\lambda n.\ r * (1 + - inverse\ (real\ (Suc\ n)))) \longrightarrow r$
⟨*proof*⟩

lemma *lim-inverse-n*: $((\lambda n.\ inverse(of\text{-}nat\ n)) \longrightarrow (0::'a::real\text{-}normed\text{-}field))$ *sequentially*
⟨*proof*⟩

lemma *LIMSEQ-Suc-n-over-n*: $(\lambda n.\ of\text{-}nat\ (Suc\ n)\ /\ of\text{-}nat\ n :: 'a :: real\text{-}normed\text{-}field) \longrightarrow 1$
⟨*proof*⟩

lemma *LIMSEQ-n-over-Suc-n*: $(\lambda n.\ of\text{-}nat\ n\ /\ of\text{-}nat\ (Suc\ n) :: 'a :: real\text{-}normed\text{-}field) \longrightarrow 1$
⟨*proof*⟩

101.8 Convergence on sequences

lemma *convergent-cong*:
assumes *eventually* $(\lambda x.\ f\ x = g\ x)$ *sequentially*
shows *convergent f* $\longleftrightarrow$ *convergent g*
⟨*proof*⟩

lemma *convergent-Suc-iff*: *convergent* $(\lambda n.\ f\ (Suc\ n)) \longleftrightarrow$ *convergent f*
⟨*proof*⟩

lemma *convergent-ignore-initial-segment*: *convergent* $(\lambda n.\ f\ (n + m))$ = *convergent f*
⟨*proof*⟩

lemma *convergent-add*:
fixes $X\ Y :: nat \Rightarrow 'a::real\text{-}normed\text{-}vector$
assumes *convergent* $(\lambda n.\ X\ n)$
and *convergent* $(\lambda n.\ Y\ n)$
shows *convergent* $(\lambda n.\ X\ n + Y\ n)$
⟨*proof*⟩

lemma *convergent-sum*:
fixes $X :: 'a \Rightarrow nat \Rightarrow 'b::real\text{-}normed\text{-}vector$

shows ($\bigwedge i.\ i \in A \Longrightarrow$ *convergent* $(\lambda n.\ X\ i\ n)) \Longrightarrow$ *convergent* $(\lambda n.\ \sum i{\in}A.\ X\ i$ $n)$
⟨*proof*⟩

lemma (**in** *bounded-linear*) *convergent*:
assumes *convergent* $(\lambda n.\ X\ n)$
shows *convergent* $(\lambda n.\ f\ (X\ n))$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *convergent*:
assumes *convergent* $(\lambda n.\ X\ n)$
and *convergent* $(\lambda n.\ Y\ n)$
shows *convergent* $(\lambda n.\ X\ n\ {*}{*}\ Y\ n)$
⟨*proof*⟩

lemma *convergent-minus-iff*: *convergent* $X \longleftrightarrow$ *convergent* $(\lambda n.\ -\ X\ n)$
for X :: *nat* $\Rightarrow$ *'a::real-normed-vector*
⟨*proof*⟩

lemma *convergent-diff*:
fixes $X\ Y$ :: *nat* $\Rightarrow$ *'a::real-normed-vector*
assumes *convergent* $(\lambda n.\ X\ n)$
assumes *convergent* $(\lambda n.\ Y\ n)$
shows *convergent* $(\lambda n.\ X\ n\ -\ Y\ n)$
⟨*proof*⟩

lemma *convergent-norm*:
assumes *convergent f*
shows *convergent* $(\lambda n.$ *norm* $(f\ n))$
⟨*proof*⟩

lemma *convergent-of-real*:
convergent f $\Longrightarrow$ *convergent* $(\lambda n.$ *of-real* $(f\ n)$ :: *'a::real-normed-algebra-1*$)$
⟨*proof*⟩

lemma *convergent-add-const-iff*:
convergent $(\lambda n.\ c + f\ n$ :: *'a::real-normed-vector*$) \longleftrightarrow$ *convergent f*
⟨*proof*⟩

lemma *convergent-add-const-right-iff*:
convergent $(\lambda n.\ f\ n + c$ :: *'a::real-normed-vector*$) \longleftrightarrow$ *convergent f*
⟨*proof*⟩

lemma *convergent-diff-const-right-iff*:
convergent $(\lambda n.\ f\ n - c$ :: *'a::real-normed-vector*$) \longleftrightarrow$ *convergent f*
⟨*proof*⟩

lemma *convergent-mult*:
fixes $X\ Y$ :: *nat* $\Rightarrow$ *'a::real-normed-field*

assumes *convergent* $(\lambda n.\ X\ n)$
and *convergent* $(\lambda n.\ Y\ n)$
shows *convergent* $(\lambda n.\ X\ n * Y\ n)$
⟨*proof*⟩

lemma *convergent-mult-const-iff*:
assumes $c \neq 0$
shows *convergent* $(\lambda n.\ c * f\ n :: 'a{::}real\text{-}normed\text{-}field) \longleftrightarrow$ *convergent f*
⟨*proof*⟩

lemma *convergent-mult-const-right-iff*:
fixes $c :: 'a{::}real\text{-}normed\text{-}field$
assumes $c \neq 0$
shows *convergent* $(\lambda n.\ f\ n * c) \longleftrightarrow$ *convergent f*
⟨*proof*⟩

lemma *convergent-imp-Bseq*: *convergent f* $\Longrightarrow$ *Bseq f*
⟨*proof*⟩

A monotone sequence converges to its least upper bound.

lemma *LIMSEQ-incseq-SUP*:
fixes $X :: nat \Rightarrow 'a{::}\{conditionally\text{-}complete\text{-}linorder,linorder\text{-}topology\}$
assumes *u*: *bdd-above* (*range X*)
and *X*: *incseq X*
shows $X \longrightarrow (SUP\ i.\ X\ i)$
⟨*proof*⟩

lemma *LIMSEQ-decseq-INF*:
fixes $X :: nat \Rightarrow 'a{::}\{conditionally\text{-}complete\text{-}linorder,\ linorder\text{-}topology\}$
assumes *u*: *bdd-below* (*range X*)
and *X*: *decseq X*
shows $X \longrightarrow (INF\ i.\ X\ i)$
⟨*proof*⟩

Main monotonicity theorem.

lemma *Bseq-monoseq-convergent*: *Bseq X* $\Longrightarrow$ *monoseq X* $\Longrightarrow$ *convergent X*
for $X :: nat \Rightarrow real$
⟨*proof*⟩

lemma *Bseq-mono-convergent*: *Bseq X* $\Longrightarrow (\forall\, m\ n.\ m \leq n \longrightarrow X\ m \leq X\ n) \Longrightarrow$ *convergent X*
for $X :: nat \Rightarrow real$
⟨*proof*⟩

lemma *monoseq-imp-convergent-iff-Bseq*: *monoseq f* $\Longrightarrow$ *convergent f* $\longleftrightarrow$ *Bseq f*
for $f :: nat \Rightarrow real$
⟨*proof*⟩

lemma *Bseq-monoseq-convergent'-inc*:

fixes f :: *nat* $\Rightarrow$ *real*
shows *Bseq* $(\lambda n.\ f\ (n + M)) \Longrightarrow (\bigwedge m\ n.\ M \leq m \Longrightarrow m \leq n \Longrightarrow f\ m \leq f\ n) \Longrightarrow$ *convergent* f
⟨*proof*⟩

lemma *Bseq-monoseq-convergent′-dec*:
fixes f :: *nat* $\Rightarrow$ *real*
shows *Bseq* $(\lambda n.\ f\ (n + M)) \Longrightarrow (\bigwedge m\ n.\ M \leq m \Longrightarrow m \leq n \Longrightarrow f\ m \geq f\ n) \Longrightarrow$ *convergent* f
⟨*proof*⟩

lemma *Cauchy-iff*: *Cauchy* $X \longleftrightarrow (\forall\, e{>}0.\ \exists\, M.\ \forall\, m{\geq}M.\ \forall\, n{\geq}M.\ norm\ (X\ m - X\ n) < e)$
for X :: *nat* $\Rightarrow$ *′a::real-normed-vector*
⟨*proof*⟩

lemma *CauchyI*: $(\bigwedge e.\ 0 < e \Longrightarrow \exists\, M.\ \forall\, m{\geq}M.\ \forall\, n{\geq}M.\ norm\ (X\ m - X\ n) < e) \Longrightarrow$ *Cauchy* X
for X :: *nat* $\Rightarrow$ *′a::real-normed-vector*
⟨*proof*⟩

lemma *CauchyD*: *Cauchy* $X \Longrightarrow 0 < e \Longrightarrow \exists\, M.\ \forall\, m{\geq}M.\ \forall\, n{\geq}M.\ norm\ (X\ m - X\ n) < e$
for X :: *nat* $\Rightarrow$ *′a::real-normed-vector*
⟨*proof*⟩

lemma *incseq-convergent*:
fixes X :: *nat* $\Rightarrow$ *real*
assumes *incseq* X
and $\forall\, i.\ X\ i \leq B$
obtains L **where** $X \longrightarrow L\ \forall\, i.\ X\ i \leq L$
⟨*proof*⟩

lemma *decseq-convergent*:
fixes X :: *nat* $\Rightarrow$ *real*
assumes *decseq* X
and $\forall\, i.\ B \leq X\ i$
obtains L **where** $X \longrightarrow L\ \forall\, i.\ L \leq X\ i$
⟨*proof*⟩

101.9 Power Sequences

The sequence x^n tends to 0 if $(0::'a) \leq x$ and $x < (1::'a)$. Proof will use (NS) Cauchy equivalence for convergence and also fact that bounded and monotonic sequence converges.

lemma *Bseq-realpow*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow$ *Bseq* $(\lambda n.\ x\ \hat{}\ n)$
for x :: *real*
⟨*proof*⟩

lemma *monoseq-realpow*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow monoseq\ (\lambda n.\ x\ \hat{}\ n)$
for $x :: real$
⟨*proof*⟩

lemma *convergent-realpow*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow convergent\ (\lambda n.\ x\ \hat{}\ n)$
for $x :: real$
⟨*proof*⟩

lemma *LIMSEQ-inverse-realpow-zero*: $1 < x \Longrightarrow (\lambda n.\ inverse\ (x\ \hat{}\ n)) \longrightarrow 0$
for $x :: real$
⟨*proof*⟩

lemma *LIMSEQ-realpow-zero*:
fixes $x :: real$
assumes $0 \leq x\ x < 1$
shows $(\lambda n.\ x\ \hat{}\ n) \longrightarrow 0$
⟨*proof*⟩

lemma *LIMSEQ-power-zero*: $norm\ x < 1 \Longrightarrow (\lambda n.\ x\ \hat{}\ n) \longrightarrow 0$
for $x :: 'a::real\text{-}normed\text{-}algebra\text{-}1$
⟨*proof*⟩

lemma *LIMSEQ-divide-realpow-zero*: $1 < x \Longrightarrow (\lambda n.\ a\ /\ (x\ \hat{}\ n) :: real) \longrightarrow 0$
⟨*proof*⟩

lemma
tendsto-power-zero:
fixes $x::'a::real\text{-}normed\text{-}algebra\text{-}1$
assumes *filterlim f at-top F*
assumes $norm\ x < 1$
shows $((\lambda y.\ x\ \hat{}\ (f\ y)) \longrightarrow 0)\ F$
⟨*proof*⟩

Limit of c^n for $|c| < (1::'a)$.

lemma *LIMSEQ-rabs-realpow-zero*: $|c| < 1 \Longrightarrow (\lambda n.\ |c|\ \hat{}\ n :: real) \longrightarrow 0$
⟨*proof*⟩

lemma *LIMSEQ-rabs-realpow-zero2*: $|c| < 1 \Longrightarrow (\lambda n.\ c\ \hat{}\ n :: real) \longrightarrow 0$
⟨*proof*⟩

101.10 Limits of Functions

lemma *LIM-eq*: $f\ -a\rightarrow\ L = (\forall r>0.\ \exists s>0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\ x - L) < r)$
for $a :: 'a::real\text{-}normed\text{-}vector$ **and** $L :: 'b::real\text{-}normed\text{-}vector$
⟨*proof*⟩

lemma *LIM-I*:

($\bigwedge r.\ 0 < r \Longrightarrow \exists s{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\,x - L) < r) \Longrightarrow f\ {-a\rightarrow}\ L$
for a :: $'a$::*real-normed-vector* **and** L :: $'b$::*real-normed-vector*
⟨*proof*⟩

lemma *LIM-D*: $f\ {-a\rightarrow}\ L \Longrightarrow 0 < r \Longrightarrow \exists s{>}0.\forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\,x - L) < r$
for a :: $'a$::*real-normed-vector* **and** L :: $'b$::*real-normed-vector*
⟨*proof*⟩

lemma *LIM-offset*: $f\ {-a\rightarrow}\ L \Longrightarrow (\lambda x.\ f\ (x + k))\ {-(a - k)\rightarrow}\ L$
for a :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *LIM-offset-zero*: $f\ {-a\rightarrow}\ L \Longrightarrow (\lambda h.\ f\ (a + h))\ {-0\rightarrow}\ L$
for a :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *LIM-offset-zero-cancel*: $(\lambda h.\ f\ (a + h))\ {-0\rightarrow}\ L \Longrightarrow f\ {-a\rightarrow}\ L$
for a :: $'a$::*real-normed-vector*
⟨*proof*⟩

lemma *LIM-offset-zero-iff*: $f\ {-a\rightarrow}\ L \longleftrightarrow (\lambda h.\ f\ (a + h))\ {-0\rightarrow}\ L$
for f :: $'a$ :: *real-normed-vector* $\Rightarrow$ -
⟨*proof*⟩

lemma *LIM-zero*: $(f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ f\,x - l) \longrightarrow 0)\ F$
for f :: $'a \Rightarrow 'b$::*real-normed-vector*
⟨*proof*⟩

lemma *LIM-zero-cancel*:
fixes f :: $'a \Rightarrow 'b$::*real-normed-vector*
shows $((\lambda x.\ f\,x - l) \longrightarrow 0)\ F \Longrightarrow (f \longrightarrow l)\ F$
⟨*proof*⟩

lemma *LIM-zero-iff*: $((\lambda x.\ f\,x - l) \longrightarrow 0)\ F = (f \longrightarrow l)\ F$
for f :: $'a \Rightarrow 'b$::*real-normed-vector*
⟨*proof*⟩

lemma *LIM-imp-LIM*:
fixes f :: $'a$::*topological-space* $\Rightarrow$ $'b$::*real-normed-vector*
fixes g :: $'a$::*topological-space* $\Rightarrow$ $'c$::*real-normed-vector*
assumes f: $f\ {-a\rightarrow}\ l$
and *le*: $\bigwedge x.\ x \neq a \Longrightarrow norm\ (g\ x - m) \leq norm\ (f\,x - l)$
shows $g\ {-a\rightarrow}\ m$
⟨*proof*⟩

lemma *LIM-equal2*:
fixes $f\ g$ :: $'a$::*real-normed-vector* $\Rightarrow$ $'b$::*topological-space*

assumes $0 < R$
and $\bigwedge x.\ x \neq a \Longrightarrow norm\ (x - a) < R \Longrightarrow f\ x = g\ x$
shows $g\ -a\rightarrow\ l \Longrightarrow f\ -a\rightarrow\ l$
⟨*proof*⟩

lemma *LIM-compose2*:
fixes $a :: 'a::real\text{-}normed\text{-}vector$
assumes f: $f\ -a\rightarrow\ b$
and g: $g\ -b\rightarrow\ c$
and *inj*: $\exists\, d>0.\ \forall\, x.\ x \neq a \wedge norm\ (x - a) < d \longrightarrow f\ x \neq b$
shows $(\lambda x.\ g\ (f\ x))\ -a\rightarrow\ c$
⟨*proof*⟩

lemma *real-LIM-sandwich-zero*:
fixes $f\ g :: 'a::topological\text{-}space \Rightarrow real$
assumes f: $f\ -a\rightarrow\ 0$
and *1*: $\bigwedge x.\ x \neq a \Longrightarrow 0 \leq g\ x$
and *2*: $\bigwedge x.\ x \neq a \Longrightarrow g\ x \leq f\ x$
shows $g\ -a\rightarrow\ 0$
⟨*proof*⟩

101.11 Continuity

lemma *LIM-isCont-iff*: $(f\ -a\rightarrow\ f\ a) = ((\lambda h.\ f\ (a + h))\ -0\rightarrow\ f\ a)$
for $f :: 'a::real\text{-}normed\text{-}vector \Rightarrow 'b::topological\text{-}space$
⟨*proof*⟩

lemma *isCont-iff*: $isCont\ f\ x = (\lambda h.\ f\ (x + h))\ -0\rightarrow\ f\ x$
for $f :: 'a::real\text{-}normed\text{-}vector \Rightarrow 'b::topological\text{-}space$
⟨*proof*⟩

lemma *isCont-LIM-compose2*:
fixes $a :: 'a::real\text{-}normed\text{-}vector$
assumes f [*unfolded isCont-def*]: $isCont\ f\ a$
and g: $g\ -f\ a\rightarrow\ l$
and *inj*: $\exists\, d>0.\ \forall\, x.\ x \neq a \wedge norm\ (x - a) < d \longrightarrow f\ x \neq f\ a$
shows $(\lambda x.\ g\ (f\ x))\ -a\rightarrow\ l$
⟨*proof*⟩

lemma *isCont-norm* [*simp*]: $isCont\ f\ a \Longrightarrow isCont\ (\lambda x.\ norm\ (f\ x))\ a$
for $f :: 'a::t2\text{-}space \Rightarrow 'b::real\text{-}normed\text{-}vector$
⟨*proof*⟩

lemma *isCont-rabs* [*simp*]: $isCont\ f\ a \Longrightarrow isCont\ (\lambda x.\ |f\ x|)\ a$
for $f :: 'a::t2\text{-}space \Rightarrow real$
⟨*proof*⟩

lemma *isCont-add* [*simp*]: $isCont\ f\ a \Longrightarrow isCont\ g\ a \Longrightarrow isCont\ (\lambda x.\ f\ x + g\ x)$
a

for *f* :: *'a::t2-space* ⇒ *'b::topological-monoid-add*
⟨*proof*⟩

lemma *isCont-minus* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. − *f x*) *a*
for *f* :: *'a::t2-space* ⇒ *'b::real-normed-vector*
⟨*proof*⟩

lemma *isCont-diff* [*simp*]: *isCont f a* ⟹ *isCont g a* ⟹ *isCont* (λ*x*. *f x* − *g x*) *a*
for *f* :: *'a::t2-space* ⇒ *'b::real-normed-vector*
⟨*proof*⟩

lemma *isCont-mult* [*simp*]: *isCont f a* ⟹ *isCont g a* ⟹ *isCont* (λ*x*. *f x* ∗ *g x*) *a*
for *f g* :: *'a::t2-space* ⇒ *'b::real-normed-algebra*
⟨*proof*⟩

lemma (**in** *bounded-linear*) *isCont*: *isCont g a* ⟹ *isCont* (λ*x*. *f* (*g x*)) *a*
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *isCont*: *isCont f a* ⟹ *isCont g a* ⟹ *isCont* (λ*x*. *f x* ∗∗ *g x*) *a*
⟨*proof*⟩

lemmas *isCont-scaleR* [*simp*] =
bounded-bilinear.isCont [*OF bounded-bilinear-scaleR*]

lemmas *isCont-of-real* [*simp*] =
bounded-linear.isCont [*OF bounded-linear-of-real*]

lemma *isCont-power* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. *f x* ^ *n*) *a*
for *f* :: *'a::t2-space* ⇒ *'b::{power,real-normed-algebra}*
⟨*proof*⟩

lemma *isCont-sum* [*simp*]: ∀ *i*∈*A*. *isCont* (*f i*) *a* ⟹ *isCont* (λ*x*. ∑ *i*∈*A*. *f i x*) *a*
for *f* :: *'a* ⇒ *'b::t2-space* ⇒ *'c::topological-comm-monoid-add*
⟨*proof*⟩

101.12 Uniform Continuity

lemma *uniformly-continuous-on-def*:
fixes *f* :: *'a::metric-space* ⇒ *'b::metric-space*
shows *uniformly-continuous-on s f* ⟷
(∀ *e*>*0*. ∃ *d*>*0*. ∀ *x*∈*s*. ∀ *x'*∈*s*. *dist x' x* < *d* ⟶ *dist* (*f x'*) (*f x*) < *e*)
⟨*proof*⟩

abbreviation *isUCont* :: [*'a::metric-space* ⇒ *'b::metric-space*] ⇒ *bool*
where *isUCont f* ≡ *uniformly-continuous-on UNIV f*

lemma *isUCont-def*: *isUCont f* $\longleftrightarrow$ ($\forall$ *r*>*0*. $\exists$ *s*>*0*. $\forall$ *x y*. *dist x y* < *s* $\longrightarrow$ *dist* (*f x*) (*f y*) < *r*)
⟨*proof*⟩

lemma *isUCont-isCont*: *isUCont f* $\Longrightarrow$ *isCont f x*
⟨*proof*⟩

lemma *uniformly-continuous-on-Cauchy*:
fixes *f* :: *'a*::*metric-space* ⇒ *'b*::*metric-space*
assumes *uniformly-continuous-on S f Cauchy X* $\bigwedge$*n*. *X n* ∈ *S*
shows *Cauchy* (λ*n*. *f* (*X n*))
⟨*proof*⟩

lemma *isUCont-Cauchy*: *isUCont f* $\Longrightarrow$ *Cauchy X* $\Longrightarrow$ *Cauchy* (λ*n*. *f* (*X n*))
⟨*proof*⟩

lemma *uniformly-continuous-imp-Cauchy-continuous*:
fixes *f* :: *'a*::*metric-space* ⇒ *'b*::*metric-space*
shows ⟦*uniformly-continuous-on S f*; *Cauchy σ*; $\bigwedge$*n*. (*σ n*) ∈ *S*⟧ $\Longrightarrow$ *Cauchy*(*f o σ*)
⟨*proof*⟩

lemma (**in** *bounded-linear*) *isUCont*: *isUCont f*
⟨*proof*⟩

lemma (**in** *bounded-linear*) *Cauchy*: *Cauchy X* $\Longrightarrow$ *Cauchy* (λ*n*. *f* (*X n*))
⟨*proof*⟩

lemma *LIM-less-bound*:
fixes *f* :: *real* ⇒ *real*
assumes *ev*: *b* < *x* ∀ *x'* ∈ { *b* <..< *x*}. *0* ≤ *f x'* **and** *isCont f x*
shows *0* ≤ *f x*
⟨*proof*⟩

101.13 Nested Intervals and Bisection – Needed for Compactness

lemma *nested-sequence-unique*:
assumes ∀ *n*. *f n* ≤ *f* (*Suc n*) ∀ *n*. *g* (*Suc n*) ≤ *g n* ∀ *n*. *f n* ≤ *g n* (λ*n*. *f n* − *g n*) ⟶ *0*
shows ∃ *l*::*real*. ((∀ *n*. *f n* ≤ *l*) ∧ *f* ⟶ *l*) ∧ ((∀ *n*. *l* ≤ *g n*) ∧ *g* ⟶ *l*)
⟨*proof*⟩

lemma *Bolzano*[*consumes 1*, *case-names trans local*]:
fixes *P* :: *real* ⇒ *real* ⇒ *bool*
assumes [*arith*]: *a* ≤ *b*
and *trans*: $\bigwedge$*a b c*. *P a b* $\Longrightarrow$ *P b c* $\Longrightarrow$ *a* ≤ *b* $\Longrightarrow$ *b* ≤ *c* $\Longrightarrow$ *P a c*
and *local*: $\bigwedge$*x*. *a* ≤ *x* $\Longrightarrow$ *x* ≤ *b* $\Longrightarrow$ ∃ *d*>*0*. ∀ *a b*. *a* ≤ *x* ∧ *x* ≤ *b* ∧ *b* − *a* < *d* ⟶ *P a b*

shows $P\ a\ b$
⟨*proof*⟩

lemma *compact-Icc*[*simp*, *intro*]: *compact* $\{a\ ..\ b::real\}$
⟨*proof*⟩

lemma *continuous-image-closed-interval*:
fixes $a\ b$ **and** $f :: real \Rightarrow real$
defines $S \equiv \{a..b\}$
assumes $a \leq b$ **and** f: *continuous-on* $S\ f$
shows $\exists\, c\ d.\ f`S = \{c..d\} \wedge c \leq d$
⟨*proof*⟩

lemma *open-Collect-positive*:
fixes $f :: 'a::t2\text{-}space \Rightarrow real$
assumes f: *continuous-on* $s\ f$
shows $\exists\, A.\ open\ A \wedge A \cap s = \{x \in s.\ 0 < f\ x\}$
⟨*proof*⟩

lemma *open-Collect-less-Int*:
fixes $f\ g :: 'a::t2\text{-}space \Rightarrow real$
assumes f: *continuous-on* $s\ f$
and g: *continuous-on* $s\ g$
shows $\exists\, A.\ open\ A \wedge A \cap s = \{x \in s.\ f\ x < g\ x\}$
⟨*proof*⟩

101.14 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

lemma *isCont-eq-Ub*:
fixes $f :: real \Rightarrow 'a::linorder\text{-}topology$
shows $a \leq b \Longrightarrow \forall\, x::real.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \Longrightarrow$
$\exists\, M.\ (\forall\, x.\ a \leq x \wedge x \leq b \longrightarrow f\ x \leq M) \wedge (\exists\, x.\ a \leq x \wedge x \leq b \wedge f\ x = M)$
⟨*proof*⟩

lemma *isCont-eq-Lb*:
fixes $f :: real \Rightarrow 'a::linorder\text{-}topology$
shows $a \leq b \Longrightarrow \forall\, x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \Longrightarrow$
$\exists\, M.\ (\forall\, x.\ a \leq x \wedge x \leq b \longrightarrow M \leq f\ x) \wedge (\exists\, x.\ a \leq x \wedge x \leq b \wedge f\ x = M)$
⟨*proof*⟩

lemma *isCont-bounded*:
fixes $f :: real \Rightarrow 'a::linorder\text{-}topology$
shows $a \leq b \Longrightarrow \forall\, x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \Longrightarrow \exists\, M.\ \forall\, x.\ a \leq x \wedge x \leq b \longrightarrow f\ x \leq M$
⟨*proof*⟩

lemma *isCont-has-Ub*:

fixes f :: *real* ⇒ *'a::linorder-topology*
shows $a \leq b \Longrightarrow \forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \Longrightarrow$
$\exists M.\ (\forall x.\ a \leq x \wedge x \leq b \longrightarrow f\ x \leq M) \wedge (\forall N.\ N < M \longrightarrow (\exists x.\ a \leq x \wedge x \leq b \wedge N < f\ x))$
⟨*proof*⟩

lemma *IVT-objl*:
$(f\ a \leq y \wedge y \leq f\ b \wedge a \leq b \wedge (\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x)) \longrightarrow$
$(\exists x.\ a \leq x \wedge x \leq b \wedge f\ x = y)$
for $a\ y$ :: *real*
⟨*proof*⟩

lemma *IVT2-objl*:
$(f\ b \leq y \wedge y \leq f\ a \wedge a \leq b \wedge (\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x)) \longrightarrow$
$(\exists x.\ a \leq x \wedge x \leq b \wedge f\ x = y)$
for $b\ y$ :: *real*
⟨*proof*⟩

lemma *isCont-Lb-Ub*:
fixes f :: *real* ⇒ *real*
assumes $a \leq b$ $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x$
shows $\exists L\ M.\ (\forall x.\ a \leq x \wedge x \leq b \longrightarrow L \leq f\ x \wedge f\ x \leq M) \wedge$
$(\forall y.\ L \leq y \wedge y \leq M \longrightarrow (\exists x.\ a \leq x \wedge x \leq b \wedge (f\ x = y)))$
⟨*proof*⟩

Continuity of inverse function.

lemma *isCont-inverse-function*:
fixes $f\ g$ :: *real* ⇒ *real*
assumes *d*: $0 < d$
and *inj*: $\forall z.\ |z-x| \leq d \longrightarrow g\ (f\ z) = z$
and *cont*: $\forall z.\ |z-x| \leq d \longrightarrow isCont\ f\ z$
shows $isCont\ g\ (f\ x)$
⟨*proof*⟩

lemma *isCont-inverse-function2*:
fixes $f\ g$ :: *real* ⇒ *real*
shows
$a < x \Longrightarrow x < b \Longrightarrow$
$\forall z.\ a \leq z \wedge z \leq b \longrightarrow g\ (f\ z) = z \Longrightarrow$
$\forall z.\ a \leq z \wedge z \leq b \longrightarrow isCont\ f\ z \Longrightarrow isCont\ g\ (f\ x)$
⟨*proof*⟩

lemma *isCont-inv-fun*:
fixes $f\ g$ :: *real* ⇒ *real*
shows $0 < d \Longrightarrow (\forall z.\ |z - x| \leq d \longrightarrow g\ (f\ z) = z) \Longrightarrow$
$\forall z.\ |z - x| \leq d \longrightarrow isCont\ f\ z \Longrightarrow isCont\ g\ (f\ x)$
⟨*proof*⟩

Bartle/Sherbert: Introduction to Real Analysis, Theorem 4.2.9, p. 110.

lemma *LIM-fun-gt-zero*: $f\ -c\rightarrow\ l \Longrightarrow 0 < l \Longrightarrow \exists\, r.\ 0 < r \wedge (\forall\, x.\ x \neq c \wedge |c - x| < r \longrightarrow 0 < f\, x)$
for $f :: real \Rightarrow real$
⟨*proof*⟩

lemma *LIM-fun-less-zero*: $f\ -c\rightarrow\ l \Longrightarrow l < 0 \Longrightarrow \exists\, r.\ 0 < r \wedge (\forall\, x.\ x \neq c \wedge |c - x| < r \longrightarrow f\, x < 0)$
for $f :: real \Rightarrow real$
⟨*proof*⟩

lemma *LIM-fun-not-zero*: $f\ -c\rightarrow\ l \Longrightarrow l \neq 0 \Longrightarrow \exists\, r.\ 0 < r \wedge (\forall\, x.\ x \neq c \wedge |c - x| < r \longrightarrow f\, x \neq 0)$
for $f :: real \Rightarrow real$
⟨*proof*⟩

end

theory *Inequalities*
imports *Real-Vector-Spaces*
begin

lemma *Sum-Icc-int*: $(m::int) \leq n \Longrightarrow \sum\ \{m..n\} = (n*(n+1) - m*(m-1))\ div\ 2$
⟨*proof*⟩

lemma *Sum-Icc-nat*: **assumes** $(m::nat) \leq n$
shows $\sum\ \{m..n\} = (n*(n+1) - m*(m-1))\ div\ 2$
⟨*proof*⟩

lemma *Sum-Ico-nat*: **assumes** $(m::nat) \leq n$
shows $\sum\ \{m..<n\} = (n*(n-1) - m*(m-1))\ div\ 2$
⟨*proof*⟩

lemma *Chebyshev-sum-upper*:
fixes $a\ b::nat \Rightarrow 'a::linordered\text{-}idom$
assumes $\bigwedge i\ j.\ i \leq j \Longrightarrow j < n \Longrightarrow a\ i \leq a\ j$
assumes $\bigwedge i\ j.\ i \leq j \Longrightarrow j < n \Longrightarrow b\ i \geq b\ j$
shows $of\text{-}nat\ n * (\sum k=0..<n.\ a\ k * b\ k) \leq (\sum k=0..<n.\ a\ k) * (\sum k=0..<n.\ b\ k)$
⟨*proof*⟩

lemma *Chebyshev-sum-upper-nat*:
fixes $a\ b :: nat \Rightarrow nat$
shows $(\bigwedge i\ j.\ [\![\ i \leq j;\ j<n\ ]\!] \Longrightarrow a\ i \leq a\ j) \Longrightarrow$
$(\bigwedge i\ j.\ [\![\ i \leq j;\ j<n\ ]\!] \Longrightarrow b\ i \geq b\ j) \Longrightarrow$
$n * (\sum i=0..<n.\ a\ i * b\ i) \leq (\sum i=0..<n.\ a\ i) * (\sum i=0..<n.\ b\ i)$
⟨*proof*⟩

end

102 Infinite Series

theory *Series*
imports *Limits Inequalities*
begin

102.1 Definition of infinite summability

definition *sums* :: $(nat \Rightarrow 'a::\{topological\text{-}space,\ comm\text{-}monoid\text{-}add\}) \Rightarrow 'a \Rightarrow bool$
(**infixr** *sums 80*)
where $f\ sums\ s \longleftrightarrow (\lambda n.\ \sum i{<}n.\ f\ i) \longrightarrow s$

definition *summable* :: $(nat \Rightarrow 'a::\{topological\text{-}space,\ comm\text{-}monoid\text{-}add\}) \Rightarrow bool$
where $summable\ f \longleftrightarrow (\exists s.\ f\ sums\ s)$

definition *suminf* :: $(nat \Rightarrow 'a::\{topological\text{-}space,\ comm\text{-}monoid\text{-}add\}) \Rightarrow 'a$
(**binder** $\sum$ *10*)
where $suminf\ f = (THE\ s.\ f\ sums\ s)$

Variants of the definition

lemma *sums-def'*: $f\ sums\ s \longleftrightarrow (\lambda n.\ \sum i = 0..n.\ f\ i) \longrightarrow s$
⟨*proof*⟩

lemma *sums-def-le*: $f\ sums\ s \longleftrightarrow (\lambda n.\ \sum i{\le}n.\ f\ i) \longrightarrow s$
⟨*proof*⟩

102.2 Infinite summability on topological monoids

lemma *sums-subst*[*trans*]: $f = g \Longrightarrow g\ sums\ z \Longrightarrow f\ sums\ z$
⟨*proof*⟩

lemma *sums-cong*: $(\bigwedge n.\ f\ n = g\ n) \Longrightarrow f\ sums\ c \longleftrightarrow g\ sums\ c$
⟨*proof*⟩

lemma *sums-summable*: $f\ sums\ l \Longrightarrow summable\ f$
⟨*proof*⟩

lemma *summable-iff-convergent*: $summable\ f \longleftrightarrow convergent\ (\lambda n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *summable-iff-convergent'*: $summable\ f \longleftrightarrow convergent\ (\lambda n.\ sum\ f\ \{..n\})$
⟨*proof*⟩

lemma *suminf-eq-lim*: $suminf\ f = lim\ (\lambda n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *sums-zero*[*simp*, *intro*]: $(\lambda n.\ 0)$ *sums* *0*
⟨*proof*⟩

lemma *summable-zero*[*simp*, *intro*]: *summable* $(\lambda n.\ 0)$
⟨*proof*⟩

lemma *sums-group*: *f sums s* $\Longrightarrow 0 < k \Longrightarrow (\lambda n.\ sum\ f\ \{n * k\ ..< n * k + k\})$ *sums s*
⟨*proof*⟩

lemma *suminf-cong*: $(\bigwedge n.\ f\ n = g\ n) \Longrightarrow suminf\ f = suminf\ g$
⟨*proof*⟩

lemma *summable-cong*:
fixes *f g* :: *nat* ⇒ *'a*::*real-normed-vector*
assumes *eventually* $(\lambda x.\ f\ x = g\ x)$ *sequentially*
shows *summable f* = *summable g*
⟨*proof*⟩

lemma *sums-finite*:
assumes [*simp*]: *finite N*
and *f*: $\bigwedge n.\ n \notin N \Longrightarrow f\ n = 0$
shows *f sums* $(\sum n{\in}N.\ f\ n)$
⟨*proof*⟩

corollary *sums-0*: $(\bigwedge n.\ f\ n = 0) \Longrightarrow (f\ sums\ 0)$
⟨*proof*⟩

lemma *summable-finite*: *finite N* $\Longrightarrow (\bigwedge n.\ n \notin N \Longrightarrow f\ n = 0) \Longrightarrow$ *summable f*
⟨*proof*⟩

lemma *sums-If-finite-set*: *finite A* $\Longrightarrow (\lambda r.\ if\ r \in A\ then\ f\ r\ else\ 0)$ *sums* $(\sum r{\in}A.\ f\ r)$
⟨*proof*⟩

lemma *summable-If-finite-set*[*simp*, *intro*]: *finite A* $\Longrightarrow$ *summable* $(\lambda r.\ if\ r \in A\ then\ f\ r\ else\ 0)$
⟨*proof*⟩

lemma *sums-If-finite*: *finite* $\{r.\ P\ r\} \Longrightarrow (\lambda r.\ if\ P\ r\ then\ f\ r\ else\ 0)$ *sums* $(\sum r \mid P\ r.\ f\ r)$
⟨*proof*⟩

lemma *summable-If-finite*[*simp*, *intro*]: *finite* $\{r.\ P\ r\} \Longrightarrow$ *summable* $(\lambda r.\ if\ P\ r\ then\ f\ r\ else\ 0)$
⟨*proof*⟩

lemma *sums-single*: $(\lambda r.\ if\ r = i\ then\ f\ r\ else\ 0)$ *sums f i*

⟨*proof*⟩

lemma *summable-single*[*simp*, *intro*]: *summable* ($\lambda r.$ *if* $r = i$ *then* $f\ r$ *else* 0)
⟨*proof*⟩

context
fixes f :: *nat* ⇒ 'a::{*t2-space*,*comm-monoid-add*}
begin

lemma *summable-sums*[*intro*]: *summable* $f \Longrightarrow f$ *sums* (*suminf* f)
⟨*proof*⟩

lemma *summable-LIMSEQ*: *summable* $f \Longrightarrow (\lambda n. \sum i{<}n.\ f\ i) \longrightarrow$ *suminf* f
⟨*proof*⟩

lemma *sums-unique*: f *sums* $s \Longrightarrow s =$ *suminf* f
⟨*proof*⟩

lemma *sums-iff*: f *sums* $x \longleftrightarrow$ *summable* $f \wedge$ *suminf* $f = x$
⟨*proof*⟩

lemma *summable-sums-iff*: *summable* $f \longleftrightarrow f$ *sums suminf* f
⟨*proof*⟩

lemma *sums-unique2*: f *sums* $a \Longrightarrow f$ *sums* $b \Longrightarrow a = b$
for $a\ b$ :: 'a
⟨*proof*⟩

lemma *suminf-finite*:
assumes N: *finite* N
and f: $\bigwedge n.\ n \notin N \Longrightarrow f\ n = 0$
shows *suminf* $f = (\sum n{\in}N.\ f\ n)$
⟨*proof*⟩

end

lemma *suminf-zero*[*simp*]: *suminf* ($\lambda n.$ 0::'a::{*t2-space*, *comm-monoid-add*}) = 0
⟨*proof*⟩

102.3 Infinite summability on ordered, topological monoids

lemma *sums-le*: $\forall n.\ f\ n \le g\ n \Longrightarrow f$ *sums* $s \Longrightarrow g$ *sums* $t \Longrightarrow s \le t$
for $f\ g$ :: *nat* ⇒ 'a::{*ordered-comm-monoid-add*,*linorder-topology*}
⟨*proof*⟩

context
fixes f :: *nat* ⇒ 'a::{*ordered-comm-monoid-add*,*linorder-topology*}
begin

lemma *suminf-le*: $\forall n.\ f\ n \leq g\ n \Longrightarrow$ *summable f* $\Longrightarrow$ *summable g* $\Longrightarrow$ *suminf f* $\leq$ *suminf g*
⟨*proof*⟩

lemma *sum-le-suminf*: *summable f* $\Longrightarrow \forall m \geq n.\ 0 \leq f\ m \Longrightarrow$ *sum f* $\{..<n\} \leq$ *suminf f*
⟨*proof*⟩

lemma *suminf-nonneg*: *summable f* $\Longrightarrow \forall n.\ 0 \leq f\ n \Longrightarrow 0 \leq$ *suminf f*
⟨*proof*⟩

lemma *suminf-le-const*: *summable f* $\Longrightarrow (\bigwedge n.$ *sum f* $\{..<n\} \leq x) \Longrightarrow$ *suminf f* $\leq x$
⟨*proof*⟩

lemma *suminf-eq-zero-iff*: *summable f* $\Longrightarrow \forall n.\ 0 \leq f\ n \Longrightarrow$ *suminf f* $= 0 \longleftrightarrow (\forall n.\ f\ n = 0)$
⟨*proof*⟩

lemma *suminf-pos-iff*: *summable f* $\Longrightarrow \forall n.\ 0 \leq f\ n \Longrightarrow 0 <$ *suminf f* $\longleftrightarrow (\exists i.\ 0 < f\ i)$
⟨*proof*⟩

lemma *suminf-pos2*:
assumes *summable f* $\forall n.\ 0 \leq f\ n$ $0 < f\ i$
shows $0 <$ *suminf f*
⟨*proof*⟩

lemma *suminf-pos*: *summable f* $\Longrightarrow \forall n.\ 0 < f\ n \Longrightarrow 0 <$ *suminf f*
⟨*proof*⟩

end

context
fixes f :: *nat* $\Rightarrow$ *'a*::{*ordered-cancel-comm-monoid-add*,*linorder-topology*}
begin

lemma *sum-less-suminf2*:
summable f $\Longrightarrow \forall m \geq n.\ 0 \leq f\ m \Longrightarrow n \leq i \Longrightarrow 0 < f\ i \Longrightarrow$ *sum f* $\{..<n\} <$ *suminf f*
⟨*proof*⟩

lemma *sum-less-suminf*: *summable f* $\Longrightarrow \forall m \geq n.\ 0 < f\ m \Longrightarrow$ *sum f* $\{..<n\} <$ *suminf f*
⟨*proof*⟩

end

lemma *summableI-nonneg-bounded*:

fixes f :: $nat \Rightarrow$ $'a$::{*ordered-comm-monoid-add*,*linorder-topology*,*conditionally-complete-linorder*}
assumes $pos[simp]$: $\bigwedge n.\ 0 \le f\ n$
and le: $\bigwedge n.\ (\sum i{<}n.\ f\ i) \le x$
shows *summable f*
⟨*proof*⟩

lemma *summableI*[*intro*, *simp*]: *summable f*
for f :: $nat \Rightarrow$ $'a$::{*canonically-ordered-monoid-add*,*linorder-topology*,*complete-linorder*}
⟨*proof*⟩

102.4 Infinite summability on topological monoids

context
fixes $f\ g$ :: $nat \Rightarrow$ $'a$::{*t2-space*,*topological-comm-monoid-add*}
begin

lemma *sums-Suc*:
assumes $(\lambda n.\ f\ (Suc\ n))$ *sums* l
shows f *sums* $(l + f\ 0)$
⟨*proof*⟩

lemma *sums-add*: f *sums* $a \Longrightarrow g$ *sums* $b \Longrightarrow (\lambda n.\ f\ n + g\ n)$ *sums* $(a + b)$
⟨*proof*⟩

lemma *summable-add*: *summable* $f \Longrightarrow$ *summable* $g \Longrightarrow$ *summable* $(\lambda n.\ f\ n + g\ n)$
⟨*proof*⟩

lemma *suminf-add*: *summable* $f \Longrightarrow$ *summable* $g \Longrightarrow$ *suminf* f + *suminf* g = $(\sum n.\ f\ n + g\ n)$
⟨*proof*⟩

end

context
fixes f :: $'i \Rightarrow nat \Rightarrow$ $'a$::{*t2-space*,*topological-comm-monoid-add*}
and I :: $'i$ *set*
begin

lemma *sums-sum*: $(\bigwedge i.\ i \in I \Longrightarrow (f\ i)$ *sums* $(x\ i)) \Longrightarrow (\lambda n.\ \sum i{\in}I.\ f\ i\ n)$ *sums* $(\sum i{\in}I.\ x\ i)$
⟨*proof*⟩

lemma *suminf-sum*: $(\bigwedge i.\ i \in I \Longrightarrow$ *summable* $(f\ i)) \Longrightarrow (\sum n.\ \sum i{\in}I.\ f\ i\ n) = (\sum i{\in}I.\ \sum n.\ f\ i\ n)$
⟨*proof*⟩

lemma *summable-sum*: $(\bigwedge i.\ i \in I \Longrightarrow$ *summable* $(f\ i)) \Longrightarrow$ *summable* $(\lambda n.\ \sum i{\in}I.\ f\ i\ n)$

⟨*proof*⟩

end

102.5 Infinite summability on real normed vector spaces

context
fixes $f :: nat \Rightarrow 'a::real\text{-}normed\text{-}vector$
begin

lemma *sums-Suc-iff*: $(\lambda n.\ f\ (Suc\ n))\ sums\ s \longleftrightarrow f\ sums\ (s + f\ 0)$
⟨*proof*⟩

lemma *summable-Suc-iff*: $summable\ (\lambda n.\ f\ (Suc\ n)) = summable\ f$
⟨*proof*⟩

lemma *sums-Suc-imp*: $f\ 0 = 0 \Longrightarrow (\lambda n.\ f\ (Suc\ n))\ sums\ s \Longrightarrow (\lambda n.\ f\ n)\ sums\ s$
⟨*proof*⟩

end

context
fixes $f :: nat \Rightarrow 'a::real\text{-}normed\text{-}vector$
begin

lemma *sums-diff*: $f\ sums\ a \Longrightarrow g\ sums\ b \Longrightarrow (\lambda n.\ f\ n - g\ n)\ sums\ (a - b)$
⟨*proof*⟩

lemma *summable-diff*: $summable\ f \Longrightarrow summable\ g \Longrightarrow summable\ (\lambda n.\ f\ n - g\ n)$
⟨*proof*⟩

lemma *suminf-diff*: $summable\ f \Longrightarrow summable\ g \Longrightarrow suminf\ f - suminf\ g = (\sum n.\ f\ n - g\ n)$
⟨*proof*⟩

lemma *sums-minus*: $f\ sums\ a \Longrightarrow (\lambda n.\ -\ f\ n)\ sums\ (-\ a)$
⟨*proof*⟩

lemma *summable-minus*: $summable\ f \Longrightarrow summable\ (\lambda n.\ -\ f\ n)$
⟨*proof*⟩

lemma *suminf-minus*: $summable\ f \Longrightarrow (\sum n.\ -\ f\ n) = -\ (\sum n.\ f\ n)$
⟨*proof*⟩

lemma *sums-iff-shift*: $(\lambda i.\ f\ (i + n))\ sums\ s \longleftrightarrow f\ sums\ (s + (\sum i{<}n.\ f\ i))$
⟨*proof*⟩

corollary *sums-iff-shift'*: $(\lambda i.\ f\ (i + n))\ sums\ (s - (\sum i{<}n.\ f\ i)) \longleftrightarrow f\ sums\ s$

⟨proof⟩

lemma *sums-zero-iff-shift*:
assumes $\bigwedge i.\ i < n \Longrightarrow f\ i = 0$
shows $(\lambda i.\ f\ (i{+}n))$ *sums* $s \longleftrightarrow (\lambda i.\ f\ i)$ *sums* s
⟨proof⟩

lemma *summable-iff-shift*: *summable* $(\lambda n.\ f\ (n + k)) \longleftrightarrow$ *summable* f
⟨proof⟩

lemma *sums-split-initial-segment*: f *sums* $s \Longrightarrow (\lambda i.\ f\ (i + n))$ *sums* $(s - (\sum i{<}n.\ f\ i))$
⟨proof⟩

lemma *summable-ignore-initial-segment*: *summable* $f \Longrightarrow$ *summable* $(\lambda n.\ f(n + k))$
⟨proof⟩

lemma *suminf-minus-initial-segment*: *summable* $f \Longrightarrow (\sum n.\ f\ (n + k)) = (\sum n.\ f\ n) - (\sum i{<}k.\ f\ i)$
⟨proof⟩

lemma *suminf-split-initial-segment*: *summable* $f \Longrightarrow$ *suminf* $f = (\sum n.\ f(n + k)) + (\sum i{<}k.\ f\ i)$
⟨proof⟩

lemma *suminf-split-head*: *summable* $f \Longrightarrow (\sum n.\ f\ (Suc\ n)) =$ *suminf* $f - f\ 0$
⟨proof⟩

lemma *suminf-exist-split*:
fixes r :: *real*
assumes $0 < r$ **and** *summable* f
shows $\exists N.\ \forall n{\geq}N.\ norm\ (\sum i.\ f\ (i + n)) < r$
⟨proof⟩

lemma *summable-LIMSEQ-zero*: *summable* $f \Longrightarrow f \longrightarrow 0$
⟨proof⟩

lemma *summable-imp-convergent*: *summable* $f \Longrightarrow$ *convergent* f
⟨proof⟩

lemma *summable-imp-Bseq*: *summable* $f \Longrightarrow$ *Bseq* f
⟨proof⟩

end

lemma *summable-minus-iff*: *summable* $(\lambda n.\ - f\ n) \longleftrightarrow$ *summable* f
for f :: *nat* ⇒ *'a::real-normed-vector*
⟨proof⟩

lemma (**in** *bounded-linear*) *sums*: $(\lambda n.\ X\ n)$ *sums* $a \Longrightarrow (\lambda n.\ f\ (X\ n))$ *sums* $(f\ a)$
⟨*proof*⟩

lemma (**in** *bounded-linear*) *summable*: *summable* $(\lambda n.\ X\ n) \Longrightarrow$ *summable* $(\lambda n.\ f\ (X\ n))$
⟨*proof*⟩

lemma (**in** *bounded-linear*) *suminf*: *summable* $(\lambda n.\ X\ n) \Longrightarrow f\ (\sum n.\ X\ n) = (\sum n.\ f\ (X\ n))$
⟨*proof*⟩

lemmas *sums-of-real* = *bounded-linear.sums* [*OF bounded-linear-of-real*]
lemmas *summable-of-real* = *bounded-linear.summable* [*OF bounded-linear-of-real*]
lemmas *suminf-of-real* = *bounded-linear.suminf* [*OF bounded-linear-of-real*]

lemmas *sums-scaleR-left* = *bounded-linear.sums*[*OF bounded-linear-scaleR-left*]
lemmas *summable-scaleR-left* = *bounded-linear.summable*[*OF bounded-linear-scaleR-left*]
lemmas *suminf-scaleR-left* = *bounded-linear.suminf*[*OF bounded-linear-scaleR-left*]

lemmas *sums-scaleR-right* = *bounded-linear.sums*[*OF bounded-linear-scaleR-right*]
lemmas *summable-scaleR-right* = *bounded-linear.summable*[*OF bounded-linear-scaleR-right*]
lemmas *suminf-scaleR-right* = *bounded-linear.suminf*[*OF bounded-linear-scaleR-right*]

lemma *summable-const-iff*: *summable* $(\lambda\text{-}.\ c) \longleftrightarrow c = 0$
for c :: $'a$::*real-normed-vector*
⟨*proof*⟩

102.6 Infinite summability on real normed algebras

context
fixes f :: *nat* $\Rightarrow$ $'a$::*real-normed-algebra*
begin

lemma *sums-mult*: f *sums* $a \Longrightarrow (\lambda n.\ c * f\ n)$ *sums* $(c * a)$
⟨*proof*⟩

lemma *summable-mult*: *summable* $f \Longrightarrow$ *summable* $(\lambda n.\ c * f\ n)$
⟨*proof*⟩

lemma *suminf-mult*: *summable* $f \Longrightarrow$ *suminf* $(\lambda n.\ c * f\ n) = c *$ *suminf* f
⟨*proof*⟩

lemma *sums-mult2*: f *sums* $a \Longrightarrow (\lambda n.\ f\ n * c)$ *sums* $(a * c)$
⟨*proof*⟩

lemma *summable-mult2*: *summable* $f \Longrightarrow$ *summable* $(\lambda n.\ f\ n * c)$
⟨*proof*⟩

lemma *suminf-mult2*: $summable\ f \Longrightarrow suminf\ f * c = (\sum n.\ f\ n * c)$
⟨*proof*⟩

end

lemma *sums-mult-iff*:
fixes $f :: nat \Rightarrow 'a::\{real\text{-}normed\text{-}algebra,field\}$
assumes $c \neq 0$
shows $(\lambda n.\ c * f\ n)\ sums\ (c * d) \longleftrightarrow f\ sums\ d$
⟨*proof*⟩

lemma *sums-mult2-iff*:
fixes $f :: nat \Rightarrow 'a::\{real\text{-}normed\text{-}algebra,field\}$
assumes $c \neq 0$
shows $(\lambda n.\ f\ n * c)\ sums\ (d * c) \longleftrightarrow f\ sums\ d$
⟨*proof*⟩

lemma *sums-of-real-iff*:
$(\lambda n.\ of\text{-}real\ (f\ n) :: 'a::real\text{-}normed\text{-}div\text{-}algebra)\ sums\ of\text{-}real\ c \longleftrightarrow f\ sums\ c$
⟨*proof*⟩

102.7 Infinite summability on real normed fields

context
fixes $c :: 'a::real\text{-}normed\text{-}field$
begin

lemma *sums-divide*: $f\ sums\ a \Longrightarrow (\lambda n.\ f\ n\ /\ c)\ sums\ (a\ /\ c)$
⟨*proof*⟩

lemma *summable-divide*: $summable\ f \Longrightarrow summable\ (\lambda n.\ f\ n\ /\ c)$
⟨*proof*⟩

lemma *suminf-divide*: $summable\ f \Longrightarrow suminf\ (\lambda n.\ f\ n\ /\ c) = suminf\ f\ /\ c$
⟨*proof*⟩

lemma *sums-mult-D*: $(\lambda n.\ c * f\ n)\ sums\ a \Longrightarrow c \neq 0 \Longrightarrow f\ sums\ (a/c)$
⟨*proof*⟩

lemma *summable-mult-D*: $summable\ (\lambda n.\ c * f\ n) \Longrightarrow c \neq 0 \Longrightarrow summable\ f$
⟨*proof*⟩

Sum of a geometric progression.

lemma *geometric-sums*:
assumes *less-1*: $norm\ c < 1$
shows $(\lambda n.\ c\hat{\ }n)\ sums\ (1\ /\ (1 - c))$
⟨*proof*⟩

lemma *summable-geometric*: $norm\ c < 1 \Longrightarrow summable\ (\lambda n.\ c\hat{\ }n)$

⟨*proof*⟩

lemma *suminf-geometric*: $norm\ c < 1 \Longrightarrow suminf\ (\lambda n.\ c\hat{}n) = 1\ /\ (1 - c)$
⟨*proof*⟩

lemma *summable-geometric-iff*: $summable\ (\lambda n.\ c\ \hat{}\ n) \longleftrightarrow norm\ c < 1$
⟨*proof*⟩

end

lemma *power-half-series*: $(\lambda n.\ (1/2::real)\hat{}Suc\ n)\ sums\ 1$
⟨*proof*⟩

102.8 Telescoping

lemma *telescope-sums*:
fixes c :: $'a::real\text{-}normed\text{-}vector$
assumes $f \longrightarrow c$
shows $(\lambda n.\ f\ (Suc\ n) - f\ n)\ sums\ (c - f\ 0)$
⟨*proof*⟩

lemma *telescope-sums'*:
fixes c :: $'a::real\text{-}normed\text{-}vector$
assumes $f \longrightarrow c$
shows $(\lambda n.\ f\ n - f\ (Suc\ n))\ sums\ (f\ 0 - c)$
⟨*proof*⟩

lemma *telescope-summable*:
fixes c :: $'a::real\text{-}normed\text{-}vector$
assumes $f \longrightarrow c$
shows $summable\ (\lambda n.\ f\ (Suc\ n) - f\ n)$
⟨*proof*⟩

lemma *telescope-summable'*:
fixes c :: $'a::real\text{-}normed\text{-}vector$
assumes $f \longrightarrow c$
shows $summable\ (\lambda n.\ f\ n - f\ (Suc\ n))$
⟨*proof*⟩

102.9 Infinite summability on Banach spaces

Cauchy-type criterion for convergence of series (c.f. Harrison).

lemma *summable-Cauchy*: $summable\ f \longleftrightarrow (\forall\, e>0.\ \exists\, N.\ \forall\, m\geq N.\ \forall\, n.\ norm\ (sum\ f\ \{m..<n\}) < e)$
for f :: $nat \Rightarrow 'a::banach$
⟨*proof*⟩

context
fixes f :: $nat \Rightarrow 'a::banach$

begin

Absolute convergence imples normal convergence.

lemma *summable-norm-cancel*: *summable* $(\lambda n.\ norm\ (f\ n)) \Longrightarrow$ *summable f*
⟨*proof*⟩

lemma *summable-norm*: *summable* $(\lambda n.\ norm\ (f\ n)) \Longrightarrow norm\ (suminf\ f) \leq (\sum n.\ norm\ (f\ n))$
⟨*proof*⟩

Comparison tests.

lemma *summable-comparison-test*: $\exists N.\ \forall n \geq N.\ norm\ (f\ n) \leq g\ n \Longrightarrow$ *summable* $g \Longrightarrow$ *summable f*
⟨*proof*⟩

lemma *summable-comparison-test-ev*:
eventually $(\lambda n.\ norm\ (f\ n) \leq g\ n)$ *sequentially* $\Longrightarrow$ *summable g* $\Longrightarrow$ *summable f*
⟨*proof*⟩

A better argument order.

lemma *summable-comparison-test'*: *summable g* $\Longrightarrow (\bigwedge n.\ n \geq N \Longrightarrow norm\ (f\ n) \leq g\ n) \Longrightarrow$ *summable f*
⟨*proof*⟩

102.10 The Ratio Test

lemma *summable-ratio-test*:
assumes $c < 1$ $\bigwedge n.\ n \geq N \Longrightarrow norm\ (f\ (Suc\ n)) \leq c * norm\ (f\ n)$
shows *summable f*
⟨*proof*⟩

end

Relations among convergence and absolute convergence for power series.

lemma *Abel-lemma*:
fixes a :: *nat* ⇒ *'a::real-normed-vector*
assumes r: $0 \leq r$
and $r0$: $r < r0$
and M: $\bigwedge n.\ norm\ (a\ n) * r0\hat{}n \leq M$
shows *summable* $(\lambda n.\ norm\ (a\ n) * r\hat{}n)$
⟨*proof*⟩

Summability of geometric series for real algebras.

lemma *complete-algebra-summable-geometric*:
fixes x :: *'a::{real-normed-algebra-1,banach}*
assumes *norm* $x < 1$
shows *summable* $(\lambda n.\ x\ \hat{}\ n)$
⟨*proof*⟩

102.11 Cauchy Product Formula

Proof based on Analysis WebNotes: Chapter 07, Class 41 http://www.math.unl.edu/~webnotes/classes/class41/prp77.htm

lemma *Cauchy-product-sums*:
fixes $a\ b :: nat \Rightarrow 'a::\{real\text{-}normed\text{-}algebra,banach\}$
assumes a: $summable\ (\lambda k.\ norm\ (a\ k))$
and b: $summable\ (\lambda k.\ norm\ (b\ k))$
shows $(\lambda k.\ \sum i{\le}k.\ a\ i * b\ (k - i))\ sums\ ((\sum k.\ a\ k) * (\sum k.\ b\ k))$
⟨*proof*⟩

lemma *Cauchy-product*:
fixes $a\ b :: nat \Rightarrow 'a::\{real\text{-}normed\text{-}algebra,banach\}$
assumes $summable\ (\lambda k.\ norm\ (a\ k))$
and $summable\ (\lambda k.\ norm\ (b\ k))$
shows $(\sum k.\ a\ k) * (\sum k.\ b\ k) = (\sum k.\ \sum i{\le}k.\ a\ i * b\ (k - i))$
⟨*proof*⟩

lemma *summable-Cauchy-product*:
fixes $a\ b :: nat \Rightarrow 'a::\{real\text{-}normed\text{-}algebra,banach\}$
assumes $summable\ (\lambda k.\ norm\ (a\ k))$
and $summable\ (\lambda k.\ norm\ (b\ k))$
shows $summable\ (\lambda k.\ \sum i{\le}k.\ a\ i * b\ (k - i))$
⟨*proof*⟩

102.12 Series on *reals*

lemma *summable-norm-comparison-test*:
$\exists N.\ \forall n{\ge}N.\ norm\ (f\ n) \le g\ n \Longrightarrow summable\ g \Longrightarrow summable\ (\lambda n.\ norm\ (f\ n))$
⟨*proof*⟩

lemma *summable-rabs-comparison-test*: $\exists N.\ \forall n{\ge}N.\ |f\ n| \le g\ n \Longrightarrow summable\ g \Longrightarrow summable\ (\lambda n.\ |f\ n|)$
for $f :: nat \Rightarrow real$
⟨*proof*⟩

lemma *summable-rabs-cancel*: $summable\ (\lambda n.\ |f\ n|) \Longrightarrow summable\ f$
for $f :: nat \Rightarrow real$
⟨*proof*⟩

lemma *summable-rabs*: $summable\ (\lambda n.\ |f\ n|) \Longrightarrow |suminf\ f| \le (\sum n.\ |f\ n|)$
for $f :: nat \Rightarrow real$
⟨*proof*⟩

lemma *summable-zero-power* [*simp*]: $summable\ (\lambda n.\ 0\ \hat{}\ n :: 'a::\{comm\text{-}ring\text{-}1,topological\text{-}space\})$
⟨*proof*⟩

lemma *summable-zero-power'* [*simp*]: $summable\ (\lambda n.\ f\ n * 0\ \hat{}\ n :: 'a::\{ring\text{-}1,topological\text{-}space\})$
⟨*proof*⟩

lemma *summable-power-series*:
  fixes z :: *real*
  assumes *le-1*: $\bigwedge i.\ f\ i \le 1$
    and *nonneg*: $\bigwedge i.\ 0 \le f\ i$
    and z: $0 \le z\ z < 1$
  shows *summable* $(\lambda i.\ f\ i * z\hat{}i)$
⟨*proof*⟩

lemma *summable-0-powser*: *summable* $(\lambda n.\ f\ n * 0\ \hat{}\ n :: 'a::real\text{-}normed\text{-}div\text{-}algebra)$
⟨*proof*⟩

lemma *summable-powser-split-head*:
  summable $(\lambda n.\ f\ (Suc\ n) * z\ \hat{}\ n :: 'a::real\text{-}normed\text{-}div\text{-}algebra) =$ *summable* $(\lambda n.$
$f\ n * z\ \hat{}\ n)$
⟨*proof*⟩

lemma *summable-powser-ignore-initial-segment*:
  fixes f :: $nat \Rightarrow 'a$ :: *real-normed-div-algebra*
  shows *summable* $(\lambda n.\ f\ (n + m) * z\ \hat{}\ n) \longleftrightarrow$ *summable* $(\lambda n.\ f\ n * z\ \hat{}\ n)$
⟨*proof*⟩

lemma *powser-split-head*:
  fixes f :: $nat \Rightarrow 'a::\{real\text{-}normed\text{-}div\text{-}algebra,banach\}$
  assumes *summable* $(\lambda n.\ f\ n * z\ \hat{}\ n)$
  shows *suminf* $(\lambda n.\ f\ n * z\ \hat{}\ n) = f\ 0 +$ *suminf* $(\lambda n.\ f\ (Suc\ n) * z\ \hat{}\ n) * z$
    and *suminf* $(\lambda n.\ f\ (Suc\ n) * z\ \hat{}\ n) * z =$ *suminf* $(\lambda n.\ f\ n * z\ \hat{}\ n) - f\ 0$
    and *summable* $(\lambda n.\ f\ (Suc\ n) * z\ \hat{}\ n)$
⟨*proof*⟩

lemma *summable-partial-sum-bound*:
  fixes f :: $nat \Rightarrow 'a$ :: *banach*
    and e :: *real*
  assumes *summable*: *summable* f
    and e: $e > 0$
  obtains N **where** $\bigwedge m\ n.\ m \ge N \Longrightarrow norm\ (\sum k{=}m..n.\ f\ k) < e$
⟨*proof*⟩

lemma *powser-sums-if*:
  $(\lambda n.\ (if\ n = m\ then\ (1 :: 'a::\{ring\text{-}1,topological\text{-}space\})\ else\ 0) * z\hat{}n)\ sums\ z\hat{}m$
⟨*proof*⟩

lemma
  fixes f :: $nat \Rightarrow real$
  assumes *summable* f
    and *inj* g
    and *pos*: $\bigwedge x.\ 0 \le f\ x$
  shows *summable-reindex*: *summable* $(f \circ g)$
    and *suminf-reindex-mono*: *suminf* $(f \circ g) \le$ *suminf* f

and *suminf-reindex*: $(\bigwedge x.\ x \notin range\ g \Longrightarrow f\ x = 0) \Longrightarrow suminf\ (f \circ g) = suminf\ f$
⟨*proof*⟩

lemma *sums-mono-reindex*:
assumes *subseq*: *strict-mono g*
and *zero*: $\bigwedge n.\ n \notin range\ g \Longrightarrow f\ n = 0$
shows $(\lambda n.\ f\ (g\ n))\ sums\ c \longleftrightarrow f\ sums\ c$
⟨*proof*⟩

lemma *summable-mono-reindex*:
assumes *subseq*: *strict-mono g*
and *zero*: $\bigwedge n.\ n \notin range\ g \Longrightarrow f\ n = 0$
shows $summable\ (\lambda n.\ f\ (g\ n)) \longleftrightarrow summable\ f$
⟨*proof*⟩

lemma *suminf-mono-reindex*:
fixes $f :: nat \Rightarrow 'a::\{t2\text{-}space,comm\text{-}monoid\text{-}add\}$
assumes *strict-mono g* $\bigwedge n.\ n \notin range\ g \Longrightarrow f\ n = 0$
shows $suminf\ (\lambda n.\ f\ (g\ n)) = suminf\ f$
⟨*proof*⟩

end

103 Differentiation

theory *Deriv*
imports *Limits*
begin

103.1 Frechet derivative

definition *has-derivative* :: $('a::real\text{-}normed\text{-}vector \Rightarrow 'b::real\text{-}normed\text{-}vector) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ filter \Rightarrow bool$ (**infix** (*has′-derivative*) *50*)
where $(f\ has\text{-}derivative\ f')\ F \longleftrightarrow$
$bounded\text{-}linear\ f' \wedge$
$((\lambda y.\ ((f\ y - f\ (Lim\ F\ (\lambda x.\ x))) - f'\ (y - Lim\ F\ (\lambda x.\ x)))\ /_R\ norm\ (y - Lim\ F\ (\lambda x.\ x))) \longrightarrow 0)\ F$

Usually the filter F is *at x within s*. $(f\ has\text{-}derivative\ D)\ (at\ x\ within\ s)$ means: D is the derivative of function f at point x within the set s. Where s is used to express left or right sided derivatives. In most cases s is either a variable or *UNIV*.

lemma *has-derivative-eq-rhs*: $(f\ has\text{-}derivative\ f')\ F \Longrightarrow f' = g' \Longrightarrow (f\ has\text{-}derivative\ g')\ F$
⟨*proof*⟩

definition *has-field-derivative* :: $('a::real\text{-}normed\text{-}field \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ filter \Rightarrow bool$

(**infix** (*has′-field′-derivative*) *50*)
where (f *has-field-derivative* D) $F \longleftrightarrow$ (f *has-derivative op* $*$ D) F

lemma *DERIV-cong*: (f *has-field-derivative* X) $F \Longrightarrow X = Y \Longrightarrow$ (f *has-field-derivative* Y) F
⟨*proof*⟩

definition *has-vector-derivative* :: (*real* ⇒ ′*b*::*real-normed-vector*) ⇒ ′*b* ⇒ *real filter* ⇒ *bool*
(**infix** *has′-vector′-derivative 50*)
where (f *has-vector-derivative* f') *net* $\longleftrightarrow$ (f *has-derivative* ($\lambda x.\ x *_R f'$)) *net*

lemma *has-vector-derivative-eq-rhs*:
(f *has-vector-derivative* X) $F \Longrightarrow X = Y \Longrightarrow$ (f *has-vector-derivative* Y) F
⟨*proof*⟩

named-theorems *derivative-intros structural introduction rules for derivatives*
⟨*ML*⟩

The following syntax is only used as a legacy syntax.

abbreviation (*input*)
FDERIV :: (′*a*::*real-normed-vector* ⇒ ′*b*::*real-normed-vector*) ⇒ ′*a* ⇒ (′*a* ⇒ ′*b*) ⇒ *bool*
((*FDERIV* (-)/ (-)/ :> (-)) [*1000*, *1000*, *60*] *60*)
where *FDERIV* f x :> $f' \equiv$ (f *has-derivative* f') (*at* x)

lemma *has-derivative-bounded-linear*: (f *has-derivative* f') $F \Longrightarrow$ *bounded-linear* f'
⟨*proof*⟩

lemma *has-derivative-linear*: (f *has-derivative* f') $F \Longrightarrow$ *linear* f'
⟨*proof*⟩

lemma *has-derivative-ident*[*derivative-intros*, *simp*]: (($\lambda x.\ x$) *has-derivative* ($\lambda x.\ x$)) F
⟨*proof*⟩

lemma *has-derivative-id* [*derivative-intros*, *simp*]: (*id has-derivative id*) (*at a*)
⟨*proof*⟩

lemma *has-derivative-const*[*derivative-intros*, *simp*]: (($\lambda x.\ c$) *has-derivative* ($\lambda x.\ 0$)) F
⟨*proof*⟩

lemma (**in** *bounded-linear*) *bounded-linear*: *bounded-linear* f ⟨*proof*⟩

lemma (**in** *bounded-linear*) *has-derivative*:
(g *has-derivative* g') $F \Longrightarrow$ (($\lambda x.\ f\ (g\ x)$) *has-derivative* ($\lambda x.\ f\ (g'\ x)$)) F
⟨*proof*⟩

lemmas *has-derivative-scaleR-right* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-scaleR-right*]

lemmas *has-derivative-scaleR-left* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-scaleR-left*]

lemmas *has-derivative-mult-right* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-mult-right*]

lemmas *has-derivative-mult-left* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-mult-left*]

lemma *has-derivative-add*[*simp*, *derivative-intros*]:
assumes f: (f *has-derivative* f') F
and g: (g *has-derivative* g') F
shows $((\lambda x.\ f\ x + g\ x)$ *has-derivative* $(\lambda x.\ f'\ x + g'\ x))\ F$
⟨*proof*⟩

lemma *has-derivative-sum*[*simp*, *derivative-intros*]:
$(\bigwedge i.\ i \in I \Longrightarrow (f\ i$ *has-derivative* $f'\ i)\ F) \Longrightarrow$
$((\lambda x.\ \sum i{\in}I.\ f\ i\ x)$ *has-derivative* $(\lambda x.\ \sum i{\in}I.\ f'\ i\ x))\ F$
⟨*proof*⟩

lemma *has-derivative-minus*[*simp*, *derivative-intros*]:
$(f$ *has-derivative* $f')\ F \Longrightarrow ((\lambda x.\ -\ f\ x)$ *has-derivative* $(\lambda x.\ -\ f'\ x))\ F$
⟨*proof*⟩

lemma *has-derivative-diff*[*simp*, *derivative-intros*]:
$(f$ *has-derivative* $f')\ F \Longrightarrow (g$ *has-derivative* $g')\ F \Longrightarrow$
$((\lambda x.\ f\ x - g\ x)$ *has-derivative* $(\lambda x.\ f'\ x - g'\ x))\ F$
⟨*proof*⟩

lemma *has-derivative-at-within*:
$(f$ *has-derivative* $f')$ (*at* x *within* s) $\longleftrightarrow$
(*bounded-linear* $f' \wedge ((\lambda y.\ ((f\ y - f\ x) - f'\ (y - x))\ /_R$ *norm* $(y - x)) \longrightarrow$
0) (*at* x *within* s))
⟨*proof*⟩

lemma *has-derivative-iff-norm*:
$(f$ *has-derivative* $f')$ (*at* x *within* s) $\longleftrightarrow$
bounded-linear $f' \wedge ((\lambda y.$ *norm* $((f\ y - f\ x) - f'\ (y - x))$ / *norm* $(y - x))$
$\longrightarrow$ *0*) (*at* x *within* s)
⟨*proof*⟩

lemma *has-derivative-at*:
$(f$ *has-derivative* $D)$ (*at* x) $\longleftrightarrow$
(*bounded-linear* $D \wedge (\lambda h.$ *norm* $(f\ (x + h) - f\ x - D\ h)$ / *norm* $h)$ −*0*→ *0*)
⟨*proof*⟩

lemma *field-has-derivative-at*:
  fixes x :: $'a$::*real-normed-field*
  shows $(f$ *has-derivative op* $*$ $D)$ $(at\ x) \longleftrightarrow (\lambda h.\ (f\ (x + h) - f\ x) / h) -0\rightarrow D$
  ⟨*proof*⟩

lemma *has-derivativeI*:
  bounded-linear $f' \Longrightarrow$
    $((\lambda y.\ ((f\ y - f\ x) - f'\ (y - x)) /_R\ norm\ (y - x)) \longrightarrow 0)\ (at\ x\ within\ s) \Longrightarrow$
    $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$
  ⟨*proof*⟩

lemma *has-derivativeI-sandwich*:
  assumes e: $0 < e$
    and *bounded*: *bounded-linear* f'
    and *sandwich*: $(\bigwedge y.\ y \in s \Longrightarrow y \neq x \Longrightarrow dist\ y\ x < e \Longrightarrow$
      $norm\ ((f\ y - f\ x) - f'\ (y - x)) / norm\ (y - x) \leq H\ y)$
    and $(H \longrightarrow 0)$ $(at\ x\ within\ s)$
  shows $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$
  ⟨*proof*⟩

lemma *has-derivative-subset*:
  $(f$ *has-derivative* $f')$ $(at\ x\ within\ s) \Longrightarrow t \subseteq s \Longrightarrow (f$ *has-derivative* $f')$ $(at\ x\ within\ t)$
  ⟨*proof*⟩

lemmas *has-derivative-within-subset* = *has-derivative-subset*

103.2 Continuity

lemma *has-derivative-continuous*:
  assumes f: $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$
  shows *continuous* $(at\ x\ within\ s)\ f$
⟨*proof*⟩

103.3 Composition

lemma *tendsto-at-iff-tendsto-nhds-within*:
  $f\ x = y \Longrightarrow (f \longrightarrow y)\ (at\ x\ within\ s) \longleftrightarrow (f \longrightarrow y)\ (inf\ (nhds\ x)\ (principal\ s))$
  ⟨*proof*⟩

lemma *has-derivative-in-compose*:
  assumes f: $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$
    and g: $(g$ *has-derivative* $g')$ $(at\ (f\ x)\ within\ (f`s))$
  shows $((\lambda x.\ g\ (f\ x))$ *has-derivative* $(\lambda x.\ g'\ (f'\ x)))$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *has-derivative-compose*:
  $(f$ *has-derivative* $f')$ $(at\ x\ within\ s) \Longrightarrow (g$ *has-derivative* $g')$ $(at\ (f\ x)) \Longrightarrow$

$((\lambda x.\ g\ (f\ x))$ *has-derivative* $(\lambda x.\ g'\ (f'\ x)))$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *FDERIV*:
assumes f: $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$ **and** g: $(g$ *has-derivative* $g')$ $(at\ x\ within\ s)$
shows $((\lambda x.\ f\ x\ {**}\ g\ x)$ *has-derivative* $(\lambda h.\ f\ x\ {**}\ g'\ h + f'\ h\ {**}\ g\ x))$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemmas *has-derivative-mult*[*simp*, *derivative-intros*] = *bounded-bilinear.FDERIV*[*OF bounded-bilinear-mult*]
lemmas *has-derivative-scaleR*[*simp*, *derivative-intros*] = *bounded-bilinear.FDERIV*[*OF bounded-bilinear-scaleR*]

lemma *has-derivative-prod*[*simp*, *derivative-intros*]:
fixes f :: $'i \Rightarrow 'a$::*real-normed-vector* $\Rightarrow 'b$::*real-normed-field*
shows $(\bigwedge i.\ i \in I \Longrightarrow (f\ i$ *has-derivative* $f'\ i)\ (at\ x\ within\ s)) \Longrightarrow$
$((\lambda x.\ \prod i{\in}I.\ f\ i\ x)$ *has-derivative* $(\lambda y.\ \sum i{\in}I.\ f'\ i\ y * (\prod j{\in}I - \{i\}.\ f\ j\ x)))$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *has-derivative-power*[*simp*, *derivative-intros*]:
fixes f :: $'a$ :: *real-normed-vector* $\Rightarrow 'b$ :: *real-normed-field*
assumes f: $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$
shows $((\lambda x.\ f\ x\hat{}n)$ *has-derivative* $(\lambda y.$ *of-nat* $n * f'\ y * f\ x\hat{}(n - 1)))$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *has-derivative-inverse'*:
fixes x :: $'a$::*real-normed-div-algebra*
assumes x: $x \neq 0$
shows (*inverse has-derivative* $(\lambda h.\ -$ (*inverse* $x * h *$ *inverse* $x)))$ $(at\ x\ within\ s)$
(**is** (*?inv has-derivative ?f*) -)
⟨*proof*⟩

lemma *has-derivative-inverse*[*simp*, *derivative-intros*]:
fixes f :: - $\Rightarrow 'a$::*real-normed-div-algebra*
assumes x: $f\ x \neq 0$
and f: $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$
shows $((\lambda x.$ *inverse* $(f\ x))$ *has-derivative* $(\lambda h.\ -$ (*inverse* $(f\ x) * f'\ h *$ *inverse* $(f\ x))))$
$(at\ x\ within\ s)$
⟨*proof*⟩

lemma *has-derivative-divide*[*simp*, *derivative-intros*]:
fixes f :: - $\Rightarrow 'a$::*real-normed-div-algebra*
assumes f: $(f$ *has-derivative* $f')$ $(at\ x\ within\ s)$

and *g*: (*g has-derivative g'*) (*at x within s*)
assumes *x*: $g\ x \neq 0$
shows (($\lambda x.\ f\ x\ /\ g\ x$) *has-derivative*
($\lambda h.\ -\ f\ x * (inverse\ (g\ x) * g'\ h * inverse\ (g\ x)) + f'\ h\ /\ g\ x$)) (*at x within s*)
⟨*proof*⟩

Conventional form requires mult-AC laws. Types real and complex only.

lemma *has-derivative-divide'*[*derivative-intros*]:
fixes *f* :: - ⇒ *'a::real-normed-field*
assumes *f*: (*f has-derivative f'*) (*at x within s*)
and *g*: (*g has-derivative g'*) (*at x within s*)
and *x*: $g\ x \neq 0$
shows (($\lambda x.\ f\ x\ /\ g\ x$) *has-derivative* ($\lambda h.\ (f'\ h * g\ x - f\ x * g'\ h)\ /\ (g\ x * g\ x)$)) (*at x within s*)
⟨*proof*⟩

103.4 Uniqueness

This can not generally shown for *op has-derivative*, as we need to approach the point from all directions. There is a proof in *Analysis* for *euclidean-space*.

lemma *has-derivative-zero-unique*:
assumes (($\lambda x.\ 0$) *has-derivative F*) (*at x*)
shows $F = (\lambda h.\ 0)$
⟨*proof*⟩

lemma *has-derivative-unique*:
assumes (*f has-derivative F*) (*at x*)
and (*f has-derivative F'*) (*at x*)
shows $F = F'$
⟨*proof*⟩

103.5 Differentiability predicate

definition *differentiable* :: (*'a::real-normed-vector* ⇒ *'b::real-normed-vector*) ⇒ *'a filter* ⇒ *bool*
(**infix** *differentiable 50*)
where *f differentiable F* ⟷ (∃ *D*. (*f has-derivative D*) *F*)

lemma *differentiable-subset*:
f differentiable (*at x within s*) ⟹ $t \subseteq s$ ⟹ *f differentiable* (*at x within t*)
⟨*proof*⟩

lemmas *differentiable-within-subset* = *differentiable-subset*

lemma *differentiable-ident* [*simp*, *derivative-intros*]: ($\lambda x.\ x$) *differentiable F*
⟨*proof*⟩

lemma *differentiable-const* [*simp*, *derivative-intros*]: ($\lambda z.\ a$) *differentiable F*

⟨*proof*⟩

lemma *differentiable-in-compose*:
f *differentiable* (*at* (g x) *within* (g's)) $\Longrightarrow$ g *differentiable* (*at* x *within* s) $\Longrightarrow$
($\lambda x.$ f (g x)) *differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *differentiable-compose*:
f *differentiable* (*at* (g x)) $\Longrightarrow$ g *differentiable* (*at* x *within* s) $\Longrightarrow$
($\lambda x.$ f (g x)) *differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *differentiable-sum* [*simp*, *derivative-intros*]:
f *differentiable* F $\Longrightarrow$ g *differentiable* F $\Longrightarrow$ ($\lambda x.$ f x + g x) *differentiable* F
⟨*proof*⟩

lemma *differentiable-minus* [*simp*, *derivative-intros*]:
f *differentiable* F $\Longrightarrow$ ($\lambda x.$ $-$ f x) *differentiable* F
⟨*proof*⟩

lemma *differentiable-diff* [*simp*, *derivative-intros*]:
f *differentiable* F $\Longrightarrow$ g *differentiable* F $\Longrightarrow$ ($\lambda x.$ f x $-$ g x) *differentiable* F
⟨*proof*⟩

lemma *differentiable-mult* [*simp*, *derivative-intros*]:
fixes f g :: '*a*::*real-normed-vector* $\Rightarrow$ '*b*::*real-normed-algebra*
shows f *differentiable* (*at* x *within* s) $\Longrightarrow$ g *differentiable* (*at* x *within* s) $\Longrightarrow$
($\lambda x.$ f x $*$ g x) *differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *differentiable-inverse* [*simp*, *derivative-intros*]:
fixes f :: '*a*::*real-normed-vector* $\Rightarrow$ '*b*::*real-normed-field*
shows f *differentiable* (*at* x *within* s) $\Longrightarrow$ f x $\neq$ *0* $\Longrightarrow$
($\lambda x.$ *inverse* (f x)) *differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *differentiable-divide* [*simp*, *derivative-intros*]:
fixes f g :: '*a*::*real-normed-vector* $\Rightarrow$ '*b*::*real-normed-field*
shows f *differentiable* (*at* x *within* s) $\Longrightarrow$ g *differentiable* (*at* x *within* s) $\Longrightarrow$
g x $\neq$ *0* $\Longrightarrow$ ($\lambda x.$ f x / g x) *differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *differentiable-power* [*simp*, *derivative-intros*]:
fixes f g :: '*a*::*real-normed-vector* $\Rightarrow$ '*b*::*real-normed-field*
shows f *differentiable* (*at* x *within* s) $\Longrightarrow$ ($\lambda x.$ f x ^ n) *differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *differentiable-scaleR* [*simp*, *derivative-intros*]:

f differentiable (*at* x *within* s) $\Longrightarrow$ *g differentiable* (*at* x *within* s) $\Longrightarrow$
($\lambda x.$ $f\ x *_R g\ x$) *differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *has-derivative-imp-has-field-derivative*:
(f *has-derivative* D) $F \Longrightarrow (\bigwedge x.\ x * D' = D\ x) \Longrightarrow$ (f *has-field-derivative* D') F
⟨*proof*⟩

lemma *has-field-derivative-imp-has-derivative*:
(f *has-field-derivative* D) $F \Longrightarrow$ (f *has-derivative op* $*$ D) F
⟨*proof*⟩

lemma *DERIV-subset*:
(f *has-field-derivative* f') (*at* x *within* s) $\Longrightarrow t \subseteq s \Longrightarrow$
(f *has-field-derivative* f') (*at* x *within* t)
⟨*proof*⟩

lemma *has-field-derivative-at-within*:
(f *has-field-derivative* f') (*at* x) $\Longrightarrow$ (f *has-field-derivative* f') (*at* x *within* s)
⟨*proof*⟩

abbreviation (*input*)
DERIV :: ('*a*::*real-normed-field* ⇒ '*a*) ⇒ '*a* ⇒ '*a* ⇒ *bool*
((*DERIV* (-)/ (-)/ :> (-)) [*1000*, *1000*, *60*] *60*)
where *DERIV f x* :> $D \equiv$ (f *has-field-derivative* D) (*at* x)

abbreviation *has-real-derivative* :: (*real* ⇒ *real*) ⇒ *real* ⇒ *real filter* ⇒ *bool*
(**infix** (*has*'-*real*'-*derivative*) *50*)
where (f *has-real-derivative* D) $F \equiv$ (f *has-field-derivative* D) F

lemma *real-differentiable-def*:
f differentiable at x within s $\longleftrightarrow$ ($\exists D.$ (f *has-real-derivative* D) (*at* x *within* s))
⟨*proof*⟩

lemma *real-differentiableE* [*elim?*]:
assumes f: *f differentiable* (*at* x *within* s)
obtains *df* **where** (f *has-real-derivative df*) (*at* x *within* s)
⟨*proof*⟩

lemma *differentiableD*:
f differentiable (*at* x *within* s) $\Longrightarrow \exists D.$ (f *has-real-derivative* D) (*at* x *within* s)
⟨*proof*⟩

lemma *differentiableI*:
(f *has-real-derivative* D) (*at* x *within* s) $\Longrightarrow$ *f differentiable* (*at* x *within* s)
⟨*proof*⟩

lemma *has-field-derivative-iff*:
(f *has-field-derivative* D) (*at* x *within* S) $\longleftrightarrow$

$((\lambda y.\ (f\ y - f\ x)\ /\ (y - x)) \longrightarrow D)\ (at\ x\ within\ S)$
⟨proof⟩

lemma *DERIV-def*: $DERIV\ f\ x :> D \longleftrightarrow (\lambda h.\ (f\ (x + h) - f\ x)\ /\ h) -0\rightarrow D$
⟨proof⟩

lemma *mult-commute-abs*: $(\lambda x.\ x * c) = op * c$
for c :: $'a$::*ab-semigroup-mult*
⟨proof⟩

103.6 Vector derivative

lemma *has-field-derivative-iff-has-vector-derivative*:
$(f\ has\text{-}field\text{-}derivative\ y)\ F \longleftrightarrow (f\ has\text{-}vector\text{-}derivative\ y)\ F$
⟨proof⟩

lemma *has-field-derivative-subset*:
$(f\ has\text{-}field\text{-}derivative\ y)\ (at\ x\ within\ s) \Longrightarrow t \subseteq s \Longrightarrow$
$(f\ has\text{-}field\text{-}derivative\ y)\ (at\ x\ within\ t)$
⟨proof⟩

lemma *has-vector-derivative-const*[*simp*, *derivative-intros*]: $((\lambda x.\ c)\ has\text{-}vector\text{-}derivative\ 0)\ net$
⟨proof⟩

lemma *has-vector-derivative-id*[*simp*, *derivative-intros*]: $((\lambda x.\ x)\ has\text{-}vector\text{-}derivative\ 1)\ net$
⟨proof⟩

lemma *has-vector-derivative-minus*[*derivative-intros*]:
$(f\ has\text{-}vector\text{-}derivative\ f')\ net \Longrightarrow ((\lambda x.\ -\ f\ x)\ has\text{-}vector\text{-}derivative\ (-\ f'))\ net$
⟨proof⟩

lemma *has-vector-derivative-add*[*derivative-intros*]:
$(f\ has\text{-}vector\text{-}derivative\ f')\ net \Longrightarrow (g\ has\text{-}vector\text{-}derivative\ g')\ net \Longrightarrow$
$((\lambda x.\ f\ x + g\ x)\ has\text{-}vector\text{-}derivative\ (f' + g'))\ net$
⟨proof⟩

lemma *has-vector-derivative-sum*[*derivative-intros*]:
$(\bigwedge i.\ i \in I \Longrightarrow (f\ i\ has\text{-}vector\text{-}derivative\ f'\ i)\ net) \Longrightarrow$
$((\lambda x.\ \sum i{\in}I.\ f\ i\ x)\ has\text{-}vector\text{-}derivative\ (\sum i{\in}I.\ f'\ i))\ net$
⟨proof⟩

lemma *has-vector-derivative-diff*[*derivative-intros*]:
$(f\ has\text{-}vector\text{-}derivative\ f')\ net \Longrightarrow (g\ has\text{-}vector\text{-}derivative\ g')\ net \Longrightarrow$
$((\lambda x.\ f\ x - g\ x)\ has\text{-}vector\text{-}derivative\ (f' - g'))\ net$
⟨proof⟩

lemma *has-vector-derivative-add-const*:

$((\lambda t.\ g\ t + z)$ *has-vector-derivative* $f')$ *net* $= ((\lambda t.\ g\ t)$ *has-vector-derivative* $f')$ *net*
⟨*proof*⟩

lemma *has-vector-derivative-diff-const*:
$((\lambda t.\ g\ t - z)$ *has-vector-derivative* $f')$ *net* $= ((\lambda t.\ g\ t)$ *has-vector-derivative* $f')$ *net*
⟨*proof*⟩

lemma (**in** *bounded-linear*) *has-vector-derivative*:
assumes (g *has-vector-derivative* g') F
shows $((\lambda x.\ f\ (g\ x))$ *has-vector-derivative* $f\ g')$ F
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *has-vector-derivative*:
assumes (f *has-vector-derivative* f') (*at* x *within* s)
and (g *has-vector-derivative* g') (*at* x *within* s)
shows $((\lambda x.\ f\,x ** g\ x)$ *has-vector-derivative* $(f\,x ** g' + f' ** g\ x))$ (*at* x *within* s)
⟨*proof*⟩

lemma *has-vector-derivative-scaleR*[*derivative-intros*]:
(f *has-field-derivative* f') (*at* x *within* s) $\Longrightarrow$ (g *has-vector-derivative* g') (*at* x *within* s) $\Longrightarrow$
$((\lambda x.\ f\,x *_R g\ x)$ *has-vector-derivative* $(f\,x *_R g' + f' *_R g\ x))$ (*at* x *within* s)
⟨*proof*⟩

lemma *has-vector-derivative-mult*[*derivative-intros*]:
(f *has-vector-derivative* f') (*at* x *within* s) $\Longrightarrow$ (g *has-vector-derivative* g') (*at* x *within* s) $\Longrightarrow$
$((\lambda x.\ f\,x * g\ x)$ *has-vector-derivative* $(f\,x * g' + f' * g\ x))$ (*at* x *within* s)
for $f\ g$ :: *real* $\Rightarrow$ *'a::real-normed-algebra*
⟨*proof*⟩

lemma *has-vector-derivative-of-real*[*derivative-intros*]:
(f *has-field-derivative* D) $F \Longrightarrow ((\lambda x.$ *of-real* $(f\,x))$ *has-vector-derivative* (*of-real* D)) F
⟨*proof*⟩

lemma *has-vector-derivative-continuous*:
(f *has-vector-derivative* D) (*at* x *within* s) $\Longrightarrow$ *continuous* (*at* x *within* s) f
⟨*proof*⟩

lemma *has-vector-derivative-mult-right*[*derivative-intros*]:
fixes a :: *'a::real-normed-algebra*
shows (f *has-vector-derivative* x) $F \Longrightarrow ((\lambda x.\ a * f\ x)$ *has-vector-derivative* $(a * x))$ F
⟨*proof*⟩

lemma *has-vector-derivative-mult-left*[*derivative-intros*]:
fixes a :: $'a$::*real-normed-algebra*
shows (f *has-vector-derivative* x) $F \Longrightarrow ((\lambda x.\ f\ x * a)$ *has-vector-derivative* $(x * a))\ F$
⟨*proof*⟩

103.7 Derivatives

lemma *DERIV-D*: *DERIV* $f\ x$:> $D \Longrightarrow (\lambda h.\ (f\ (x + h) - f\ x)\ /\ h)\ {-0}{\rightarrow}\ D$
⟨*proof*⟩

lemma *has-field-derivativeD*:
(f *has-field-derivative* D) (*at* x *within* S) $\Longrightarrow$
$((\lambda y.\ (f\ y - f\ x)\ /\ (y - x)) \longrightarrow D)$ (*at* x *within* S)
⟨*proof*⟩

lemma *DERIV-const* [*simp*, *derivative-intros*]: $((\lambda x.\ k)$ *has-field-derivative* $0)\ F$
⟨*proof*⟩

lemma *DERIV-ident* [*simp*, *derivative-intros*]: $((\lambda x.\ x)$ *has-field-derivative* $1)\ F$
⟨*proof*⟩

lemma *field-differentiable-add*[*derivative-intros*]:
(f *has-field-derivative* f') $F \Longrightarrow$ (g *has-field-derivative* g') $F \Longrightarrow$
$((\lambda z.\ f\ z + g\ z)$ *has-field-derivative* $f' + g')\ F$
⟨*proof*⟩

corollary *DERIV-add*:
(f *has-field-derivative* D) (*at* x *within* s) $\Longrightarrow$ (g *has-field-derivative* E) (*at* x *within* s) $\Longrightarrow$
$((\lambda x.\ f\ x + g\ x)$ *has-field-derivative* $D + E$) (*at* x *within* s)
⟨*proof*⟩

lemma *field-differentiable-minus*[*derivative-intros*]:
(f *has-field-derivative* f') $F \Longrightarrow ((\lambda z.\ - (f\ z))$ *has-field-derivative* $-f')\ F$
⟨*proof*⟩

corollary *DERIV-minus*:
(f *has-field-derivative* D) (*at* x *within* s) $\Longrightarrow$
$((\lambda x.\ - f\ x)$ *has-field-derivative* $-D$) (*at* x *within* s)
⟨*proof*⟩

lemma *field-differentiable-diff*[*derivative-intros*]:
(f *has-field-derivative* f') $F \Longrightarrow$
(g *has-field-derivative* g') $F \Longrightarrow ((\lambda z.\ f\ z - g\ z)$ *has-field-derivative* $f' - g')\ F$
⟨*proof*⟩

corollary *DERIV-diff*:
(f *has-field-derivative* D) (*at* x *within* s) $\Longrightarrow$

$(g$ *has-field-derivative* $E)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x - g\ x)$ *has-field-derivative* $D - E)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-continuous*: $(f$ *has-field-derivative* $D)$ $(at\ x\ within\ s) \Longrightarrow$ *continuous* $(at\ x\ within\ s)\ f$
⟨*proof*⟩

corollary *DERIV-isCont*: $DERIV\ f\ x :> D \Longrightarrow isCont\ f\ x$
⟨*proof*⟩

lemma *DERIV-continuous-on*:
$(\bigwedge x.\ x \in s \Longrightarrow (f$ *has-field-derivative* $(D\ x))$ $(at\ x\ within\ s)) \Longrightarrow$ *continuous-on* $s\ f$
⟨*proof*⟩

lemma *DERIV-mult′*:
$(f$ *has-field-derivative* $D)$ $(at\ x\ within\ s) \Longrightarrow (g$ *has-field-derivative* $E)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x * g\ x)$ *has-field-derivative* $f\ x * E + D * g\ x)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-mult*[*derivative-intros*]:
$(f$ *has-field-derivative* $Da)$ $(at\ x\ within\ s) \Longrightarrow (g$ *has-field-derivative* $Db)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x * g\ x)$ *has-field-derivative* $Da * g\ x + Db * f\ x)$ $(at\ x\ within\ s)$
⟨*proof*⟩

Derivative of linear multiplication

lemma *DERIV-cmult*:
$(f$ *has-field-derivative* $D)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ c * f\ x)$ *has-field-derivative* $c * D)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-cmult-right*:
$(f$ *has-field-derivative* $D)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x * c)$ *has-field-derivative* $D * c)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-cmult-Id* [*simp*]: $(op * c$ *has-field-derivative* $c)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-cdivide*:
$(f$ *has-field-derivative* $D)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x\ /\ c)$ *has-field-derivative* $D\ /\ c)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-unique*: $DERIV\ f\ x :> D \Longrightarrow DERIV\ f\ x :> E \Longrightarrow D = E$
⟨*proof*⟩

lemma *DERIV-sum*[*derivative-intros*]:
$(\bigwedge n.\ n \in S \Longrightarrow ((\lambda x.\ f\ x\ n)\ \textit{has-field-derivative}\ (f'\ x\ n))\ F) \Longrightarrow$
$((\lambda x.\ \textit{sum}\ (f\ x)\ S)\ \textit{has-field-derivative}\ \textit{sum}\ (f'\ x)\ S)\ F$
⟨*proof*⟩

lemma *DERIV-inverse'*[*derivative-intros*]:
assumes $(f\ \textit{has-field-derivative}\ D)\ (\textit{at}\ x\ \textit{within}\ s)$
and $f\ x \neq 0$
shows $((\lambda x.\ \textit{inverse}\ (f\ x))\ \textit{has-field-derivative}\ -\ (\textit{inverse}\ (f\ x) * D * \textit{inverse}\ (f\ x)))$
$(\textit{at}\ x\ \textit{within}\ s)$
⟨*proof*⟩

Power of -1

lemma *DERIV-inverse*:
$x \neq 0 \Longrightarrow ((\lambda x.\ \textit{inverse}(x))\ \textit{has-field-derivative}\ -\ (\textit{inverse}\ x\ \hat{}\ \textit{Suc}\ (\textit{Suc}\ 0)))\ (\textit{at}\ x\ \textit{within}\ s)$
⟨*proof*⟩

Derivative of inverse

lemma *DERIV-inverse-fun*:
$(f\ \textit{has-field-derivative}\ d)\ (\textit{at}\ x\ \textit{within}\ s) \Longrightarrow f\ x \neq 0 \Longrightarrow$
$((\lambda x.\ \textit{inverse}\ (f\ x))\ \textit{has-field-derivative}\ (-\ (d * \textit{inverse}(f\ x\ \hat{}\ \textit{Suc}\ (\textit{Suc}\ 0)))))\ (\textit{at}\ x\ \textit{within}\ s)$
⟨*proof*⟩

Derivative of quotient

lemma *DERIV-divide*[*derivative-intros*]:
$(f\ \textit{has-field-derivative}\ D)\ (\textit{at}\ x\ \textit{within}\ s) \Longrightarrow$
$(g\ \textit{has-field-derivative}\ E)\ (\textit{at}\ x\ \textit{within}\ s) \Longrightarrow g\ x \neq 0 \Longrightarrow$
$((\lambda x.\ f\ x\ /\ g\ x)\ \textit{has-field-derivative}\ (D * g\ x - f\ x * E)\ /\ (g\ x * g\ x))\ (\textit{at}\ x\ \textit{within}\ s)$
⟨*proof*⟩

lemma *DERIV-quotient*:
$(f\ \textit{has-field-derivative}\ d)\ (\textit{at}\ x\ \textit{within}\ s) \Longrightarrow$
$(g\ \textit{has-field-derivative}\ e)\ (\textit{at}\ x\ \textit{within}\ s) \Longrightarrow g\ x \neq 0 \Longrightarrow$
$((\lambda y.\ f\ y\ /\ g\ y)\ \textit{has-field-derivative}\ (d * g\ x - (e * f\ x))\ /\ (g\ x\ \hat{}\ \textit{Suc}\ (\textit{Suc}\ 0)))\ (\textit{at}\ x\ \textit{within}\ s)$
⟨*proof*⟩

lemma *DERIV-power-Suc*:
$(f\ \textit{has-field-derivative}\ D)\ (\textit{at}\ x\ \textit{within}\ s) \Longrightarrow$
$((\lambda x.\ f\ x\ \hat{}\ \textit{Suc}\ n)\ \textit{has-field-derivative}\ (1 + \textit{of-nat}\ n) * (D * f\ x\ \hat{}\ n))\ (\textit{at}\ x\ \textit{within}\ s)$
⟨*proof*⟩

lemma *DERIV-power*[*derivative-intros*]:

$(f$ *has-field-derivative* $D)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x\ \hat{}\ n)$ *has-field-derivative* *of-nat* $n * (D * f\ x\ \hat{}\ (n - Suc\ 0)))$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-pow*: $((\lambda x.\ x\ \hat{}\ n)$ *has-field-derivative* *real* $n * (x\ \hat{}\ (n - Suc\ 0)))$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-chain'*: $(f$ *has-field-derivative* $D)$ $(at\ x\ within\ s) \Longrightarrow DERIV\ g\ (f\ x) :> E \Longrightarrow$
$((\lambda x.\ g\ (f\ x))$ *has-field-derivative* $E * D)$ $(at\ x\ within\ s)$
⟨*proof*⟩

corollary *DERIV-chain2*: $DERIV\ f\ (g\ x) :> Da \Longrightarrow (g$ *has-field-derivative* $Db)$ $(at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ (g\ x))$ *has-field-derivative* $Da * Db)$ $(at\ x\ within\ s)$
⟨*proof*⟩

Standard version

lemma *DERIV-chain*:
$DERIV\ f\ (g\ x) :> Da \Longrightarrow (g$ *has-field-derivative* $Db)$ $(at\ x\ within\ s) \Longrightarrow$
$(f \circ g$ *has-field-derivative* $Da * Db)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-image-chain*:
$(f$ *has-field-derivative* $Da)$ $(at\ (g\ x)\ within\ (g\ `\ s)) \Longrightarrow$
$(g$ *has-field-derivative* $Db)$ $(at\ x\ within\ s) \Longrightarrow$
$(f \circ g$ *has-field-derivative* $Da * Db)$ $(at\ x\ within\ s)$
⟨*proof*⟩

lemma *DERIV-chain-s*:
assumes $(\bigwedge x.\ x \in s \Longrightarrow DERIV\ g\ x :> g'(x))$
and $DERIV\ f\ x :> f'$
and $f\ x \in s$
shows $DERIV\ (\lambda x.\ g(f\ x))\ x :> f' * g'(f\ x)$
⟨*proof*⟩

lemma *DERIV-chain3*:
assumes $(\bigwedge x.\ DERIV\ g\ x :> g'(x))$
and $DERIV\ f\ x :> f'$
shows $DERIV\ (\lambda x.\ g(f\ x))\ x :> f' * g'(f\ x)$
⟨*proof*⟩

Alternative definition for differentiability

lemma *DERIV-LIM-iff*:
fixes $f :: 'a::\{real\text{-}normed\text{-}vector,inverse\} \Rightarrow 'a$
shows $((\lambda h.\ (f\ (a + h) - f\ a)\ /\ h) -0\to D) = ((\lambda x.\ (f\ x - f\ a)\ /\ (x - a))$

$-a\rightarrow D)$
⟨*proof*⟩

lemmas *DERIV-iff2* = *has-field-derivative-iff*

lemma *has-field-derivative-cong-ev*:
assumes $x = y$
and $*$: *eventually* $(\lambda x.\ x \in s \longrightarrow f\ x = g\ x)$ $(nhds\ x)$
and $u = v\ s = t\ x \in s$
shows $(f$ *has-field-derivative* $u)$ $(at\ x\ within\ s) = (g$ *has-field-derivative* $v)$ $(at\ y$ *within* $t)$
⟨*proof*⟩

lemma *DERIV-cong-ev*:
$x = y \Longrightarrow$ *eventually* $(\lambda x.\ f\ x = g\ x)$ $(nhds\ x) \Longrightarrow u = v \Longrightarrow$
DERIV $f\ x :> u \longleftrightarrow$ *DERIV* $g\ y :> v$
⟨*proof*⟩

lemma *DERIV-shift*:
$(f$ *has-field-derivative* $y)$ $(at\ (x + z)) = ((\lambda x.\ f\ (x + z))$ *has-field-derivative* $y)$ $(at\ x)$
⟨*proof*⟩

lemma *DERIV-mirror*: (*DERIV* $f\ (-\ x) :> y) \longleftrightarrow$ (*DERIV* $(\lambda x.\ f\ (-\ x))\ x :> -\ y)$
for $f :: real \Rightarrow real$ **and** $x\ y :: real$
⟨*proof*⟩

lemma *floor-has-real-derivative*:
fixes $f :: real \Rightarrow 'a$::{*floor-ceiling*,*order-topology*}
assumes *isCont* $f\ x$
and $f\ x \notin \mathbb{Z}$
shows $((\lambda x.\ floor\ (f\ x))$ *has-real-derivative* $0)$ $(at\ x)$
⟨*proof*⟩

Caratheodory formulation of derivative at a point

lemma *CARAT-DERIV*:
(*DERIV* $f\ x :> l) \longleftrightarrow (\exists g.\ (\forall z.\ f\ z - f\ x = g\ z * (z - x)) \land isCont\ g\ x \land g\ x = l)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

103.8 Local extrema

If $(0::'a) < f'\ x$ then x is Locally Strictly Increasing At The Right.

lemma *has-real-derivative-pos-inc-right*:
fixes $f :: real \Rightarrow real$
assumes *der*: $(f$ *has-real-derivative* $l)$ $(at\ x\ within\ S)$
and l: $0 < l$

shows $\exists\, d > 0.\ \forall\, h > 0.\ x + h \in S \longrightarrow h < d \longrightarrow f\, x < f\ (x + h)$
⟨*proof*⟩

lemma *DERIV-pos-inc-right*:
fixes f :: *real* ⇒ *real*
assumes *der*: *DERIV f x* :> l
and l: $0 < l$
shows $\exists\, d > 0.\ \forall\, h > 0.\ h < d \longrightarrow f\, x < f\ (x + h)$
⟨*proof*⟩

lemma *has-real-derivative-neg-dec-left*:
fixes f :: *real* ⇒ *real*
assumes *der*: (f *has-real-derivative* l) (*at* x *within* S)
and $l < 0$
shows $\exists\, d > 0.\ \forall\, h > 0.\ x - h \in S \longrightarrow h < d \longrightarrow f\, x < f\ (x - h)$
⟨*proof*⟩

lemma *DERIV-neg-dec-left*:
fixes f :: *real* ⇒ *real*
assumes *der*: *DERIV f x* :> l
and l: $l < 0$
shows $\exists\, d > 0.\ \forall\, h > 0.\ h < d \longrightarrow f\, x < f\ (x - h)$
⟨*proof*⟩

lemma *has-real-derivative-pos-inc-left*:
fixes f :: *real* ⇒ *real*
shows (f *has-real-derivative* l) (*at* x *within* S) $\Longrightarrow 0 < l \Longrightarrow$
$\exists\, d{>}0.\ \forall\, h{>}0.\ x - h \in S \longrightarrow h < d \longrightarrow f\ (x - h) < f\, x$
⟨*proof*⟩

lemma *DERIV-pos-inc-left*:
fixes f :: *real* ⇒ *real*
shows *DERIV f x* :> $l \Longrightarrow 0 < l \Longrightarrow \exists\, d > 0.\ \forall\, h > 0.\ h < d \longrightarrow f\ (x - h)$
$< f\, x$
⟨*proof*⟩

lemma *has-real-derivative-neg-dec-right*:
fixes f :: *real* ⇒ *real*
shows (f *has-real-derivative* l) (*at* x *within* S) $\Longrightarrow l < 0 \Longrightarrow$
$\exists\, d > 0.\ \forall\, h > 0.\ x + h \in S \longrightarrow h < d \longrightarrow f\, x > f\ (x + h)$
⟨*proof*⟩

lemma *DERIV-neg-dec-right*:
fixes f :: *real* ⇒ *real*
shows *DERIV f x* :> $l \Longrightarrow l < 0 \Longrightarrow \exists\, d > 0.\ \forall\, h > 0.\ h < d \longrightarrow f\, x > f\ (x$
$+ h)$
⟨*proof*⟩

lemma *DERIV-local-max*:

fixes $f :: real \Rightarrow real$
assumes *der*: $DERIV\ f\ x :> l$
and *d*: $0 < d$
and *le*: $\forall y.\ |x - y| < d \longrightarrow f\ y \leq f\ x$
shows $l = 0$
⟨*proof*⟩

Similar theorem for a local minimum

lemma *DERIV-local-min*:
fixes $f :: real \Rightarrow real$
shows $DERIV\ f\ x :> l \Longrightarrow 0 < d \Longrightarrow \forall y.\ |x - y| < d \longrightarrow f\ x \leq f\ y \Longrightarrow l = 0$
⟨*proof*⟩

In particular, if a function is locally flat

lemma *DERIV-local-const*:
fixes $f :: real \Rightarrow real$
shows $DERIV\ f\ x :> l \Longrightarrow 0 < d \Longrightarrow \forall y.\ |x - y| < d \longrightarrow f\ x = f\ y \Longrightarrow l = 0$
⟨*proof*⟩

103.9 Rolle's Theorem

Lemma about introducing open ball in open interval

lemma *lemma-interval-lt*: $a < x \Longrightarrow x < b \Longrightarrow \exists d.\ 0 < d \wedge (\forall y.\ |x - y| < d \longrightarrow a < y \wedge y < b)$
for $a\ b\ x :: real$
⟨*proof*⟩

lemma *lemma-interval*: $a < x \Longrightarrow x < b \Longrightarrow \exists d.\ 0 < d \wedge (\forall y.\ |x - y| < d \longrightarrow a \leq y \wedge y \leq b)$
for $a\ b\ x :: real$
⟨*proof*⟩

Rolle's Theorem. If f is defined and continuous on the closed interval $[a,b]$ and differentiable on the open interval (a,b), and $f\ a = f\ b$, then there exists $x0 \in (a,b)$ such that $f'\ x0 = (0::'a)$

theorem *Rolle*:
fixes $a\ b :: real$
assumes *lt*: $a < b$
and *eq*: $f\ a = f\ b$
and *con*: $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x$
and *dif* [*rule-format*]: $\forall x.\ a < x \wedge x < b \longrightarrow f\ differentiable\ (at\ x)$
shows $\exists z.\ a < z \wedge z < b \wedge DERIV\ f\ z :> 0$
⟨*proof*⟩

103.10 Mean Value Theorem

lemma *lemma-MVT*: $f\ a - (f\ b - f\ a) / (b - a) * a = f\ b - (f\ b - f\ a) / (b - a) * b$

for $a\ b$ :: *real*
⟨*proof*⟩

theorem *MVT*:
fixes $a\ b$ :: *real*
assumes *lt*: $a < b$
and *con*: $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x$
and *dif* [*rule-format*]: $\forall x.\ a < x \wedge x < b \longrightarrow f\ differentiable\ (at\ x)$
shows $\exists l\ z.\ a < z \wedge z < b \wedge DERIV\ f\ z :> l \wedge f\ b - f\ a = (b - a) * l$
⟨*proof*⟩

lemma *MVT2*:
$a < b \Longrightarrow \forall x.\ a \leq x \wedge x \leq b \longrightarrow DERIV\ f\ x :> f'\ x \Longrightarrow$
$\exists z{::}real.\ a < z \wedge z < b \wedge (f\ b - f\ a = (b - a) * f'\ z)$
⟨*proof*⟩

A function is constant if its derivative is 0 over an interval.

lemma *DERIV-isconst-end*:
fixes f :: *real* ⇒ *real*
shows $a < b \Longrightarrow$
$\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \Longrightarrow$
$\forall x.\ a < x \wedge x < b \longrightarrow DERIV\ f\ x :> 0 \Longrightarrow f\ b = f\ a$
⟨*proof*⟩

lemma *DERIV-isconst1*:
fixes f :: *real* ⇒ *real*
shows $a < b \Longrightarrow$
$\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \Longrightarrow$
$\forall x.\ a < x \wedge x < b \longrightarrow DERIV\ f\ x :> 0 \Longrightarrow$
$\forall x.\ a \leq x \wedge x \leq b \longrightarrow f\ x = f\ a$
⟨*proof*⟩

lemma *DERIV-isconst2*:
fixes f :: *real* ⇒ *real*
shows $a < b \Longrightarrow$
$\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x \Longrightarrow$
$\forall x.\ a < x \wedge x < b \longrightarrow DERIV\ f\ x :> 0 \Longrightarrow$
$a \leq x \Longrightarrow x \leq b \Longrightarrow f\ x = f\ a$
⟨*proof*⟩

lemma *DERIV-isconst3*:
fixes $a\ b\ x\ y$ :: *real*
assumes $a < b$
and $x \in \{a <..< b\}$
and $y \in \{a <..< b\}$
and *derivable*: $\bigwedge x.\ x \in \{a <..< b\} \Longrightarrow DERIV\ f\ x :> 0$
shows $f\ x = f\ y$
⟨*proof*⟩

lemma *DERIV-isconst-all*:
fixes $f :: real \Rightarrow real$
shows $\forall x.\ DERIV\ f\ x :> 0 \Longrightarrow f\ x = f\ y$
⟨*proof*⟩

lemma *DERIV-const-ratio-const*:
fixes $f :: real \Rightarrow real$
shows $a \neq b \Longrightarrow \forall x.\ DERIV\ f\ x :> k \Longrightarrow f\ b - f\ a = (b - a) * k$
⟨*proof*⟩

lemma *DERIV-const-ratio-const2*:
fixes $f :: real \Rightarrow real$
shows $a \neq b \Longrightarrow \forall x.\ DERIV\ f\ x :> k \Longrightarrow (f\ b - f\ a) / (b - a) = k$
⟨*proof*⟩

lemma *real-average-minus-first* [*simp*]: $(a + b) / 2 - a = (b - a) / 2$
for $a\ b :: real$
⟨*proof*⟩

lemma *real-average-minus-second* [*simp*]: $(b + a) / 2 - a = (b - a) / 2$
for $a\ b :: real$
⟨*proof*⟩

Gallileo's "trick": average velocity = av. of end velocities.

lemma *DERIV-const-average*:
fixes $v :: real \Rightarrow real$
and $a\ b :: real$
assumes *neq*: $a \neq b$
and *der*: $\forall x.\ DERIV\ v\ x :> k$
shows $v\ ((a + b) / 2) = (v\ a + v\ b) / 2$
⟨*proof*⟩

A function with positive derivative is increasing. A simple proof using the MVT, by Jeremy Avigad. And variants.

lemma *DERIV-pos-imp-increasing-open*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a < b$
and $\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y > 0)$
and *con*: $\bigwedge x.\ a \leq x \Longrightarrow x \leq b \Longrightarrow isCont\ f\ x$
shows $f\ a < f\ b$
⟨*proof*⟩

lemma *DERIV-pos-imp-increasing*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a < b$
and $\forall x.\ a \leq x \wedge x \leq b \longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y > 0)$
shows $f\ a < f\ b$

⟨proof⟩

lemma *DERIV-nonneg-imp-nondecreasing*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a \leq b$
and $\forall x.\ a \leq x \wedge x \leq b \longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y \geq 0)$
shows $f\ a \leq f\ b$
⟨proof⟩

lemma *DERIV-neg-imp-decreasing-open*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a < b$
and $\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y < 0)$
and *con*: $\bigwedge x.\ a \leq x \Longrightarrow x \leq b \Longrightarrow isCont\ f\ x$
shows $f\ a > f\ b$
⟨proof⟩

lemma *DERIV-neg-imp-decreasing*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a < b$
and $\forall x.\ a \leq x \wedge x \leq b \longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y < 0)$
shows $f\ a > f\ b$
⟨proof⟩

lemma *DERIV-nonpos-imp-nonincreasing*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a \leq b$
and $\forall x.\ a \leq x \wedge x \leq b \longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y \leq 0)$
shows $f\ a \geq f\ b$
⟨proof⟩

lemma *DERIV-pos-imp-increasing-at-bot*:
fixes $f :: real \Rightarrow real$
assumes $\bigwedge x.\ x \leq b \Longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y > 0)$
and *lim*: $(f \longrightarrow flim)\ at\text{-}bot$
shows $flim < f\ b$
⟨proof⟩

lemma *DERIV-neg-imp-decreasing-at-top*:
fixes $f :: real \Rightarrow real$
assumes *der*: $\bigwedge x.\ x \geq b \Longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y < 0)$
and *lim*: $(f \longrightarrow flim)\ at\text{-}top$
shows $flim < f\ b$
⟨proof⟩

Derivative of inverse function

lemma *DERIV-inverse-function*:
fixes $f\ g :: real \Rightarrow real$
assumes *der*: $DERIV\ f\ (g\ x) :> D$
and *neq*: $D \neq 0$
and *x*: $a < x\ x < b$
and *inj*: $\forall y.\ a < y \wedge y < b \longrightarrow f\ (g\ y) = y$
and *cont*: $isCont\ g\ x$
shows $DERIV\ g\ x :> inverse\ D$
⟨*proof*⟩

103.11 Generalized Mean Value Theorem

theorem *GMVT*:
fixes $a\ b :: real$
assumes *alb*: $a < b$
and *fc*: $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x$
and *fd*: $\forall x.\ a < x \wedge x < b \longrightarrow f\ differentiable\ (at\ x)$
and *gc*: $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ g\ x$
and *gd*: $\forall x.\ a < x \wedge x < b \longrightarrow g\ differentiable\ (at\ x)$
shows $\exists g'c\ f'c\ c.$
$DERIV\ g\ c :> g'c \wedge DERIV\ f\ c :> f'c \wedge a < c \wedge c < b \wedge (f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$
⟨*proof*⟩

lemma *GMVT'*:
fixes $f\ g :: real \Rightarrow real$
assumes $a < b$
and *isCont-f*: $\bigwedge z.\ a \leq z \Longrightarrow z \leq b \Longrightarrow isCont\ f\ z$
and *isCont-g*: $\bigwedge z.\ a \leq z \Longrightarrow z \leq b \Longrightarrow isCont\ g\ z$
and *DERIV-g*: $\bigwedge z.\ a < z \Longrightarrow z < b \Longrightarrow DERIV\ g\ z :> (g'\ z)$
and *DERIV-f*: $\bigwedge z.\ a < z \Longrightarrow z < b \Longrightarrow DERIV\ f\ z :> (f'\ z)$
shows $\exists c.\ a < c \wedge c < b \wedge (f\ b - f\ a) * g'\ c = (g\ b - g\ a) * f'\ c$
⟨*proof*⟩

103.12 L'Hopitals rule

lemma *isCont-If-ge*:
fixes $a :: 'a :: linorder\text{-}topology$
shows $continuous\ (at\text{-}left\ a)\ g \Longrightarrow (f \longrightarrow g\ a)\ (at\text{-}right\ a) \Longrightarrow$
$isCont\ (\lambda x.\ if\ x \leq a\ then\ g\ x\ else\ f\ x)\ a$
⟨*proof*⟩

lemma *lhopital-right-0*:
fixes $f0\ g0 :: real \Rightarrow real$
assumes *f-0*: $(f0 \longrightarrow 0)\ (at\text{-}right\ 0)$
and *g-0*: $(g0 \longrightarrow 0)\ (at\text{-}right\ 0)$
and *ev*:
$eventually\ (\lambda x.\ g0\ x \neq 0)\ (at\text{-}right\ 0)$
$eventually\ (\lambda x.\ g'\ x \neq 0)\ (at\text{-}right\ 0)$
$eventually\ (\lambda x.\ DERIV\ f0\ x :> f'\ x)\ (at\text{-}right\ 0)$

    eventually (λ*x*. *DERIV g0 x* :> *g′ x*) (*at-right 0*)
  and *lim*: *filterlim* (λ *x*. (*f′ x* / *g′ x*)) *F* (*at-right 0*)
 shows *filterlim* (λ *x*. *f0 x* / *g0 x*) *F* (*at-right 0*)
⟨*proof*⟩

lemma *lhopital-right*:
 (*f* ⟶ *0*) (*at-right x*) ⟹ (*g* ⟶ *0*) (*at-right x*) ⟹
  eventually (λ*x*. *g x* ≠ *0*) (*at-right x*) ⟹
  eventually (λ*x*. *g′ x* ≠ *0*) (*at-right x*) ⟹
  eventually (λ*x*. *DERIV f x* :> *f′ x*) (*at-right x*) ⟹
  eventually (λ*x*. *DERIV g x* :> *g′ x*) (*at-right x*) ⟹
  filterlim (λ *x*. (*f′ x* / *g′ x*)) *F* (*at-right x*) ⟹
 filterlim (λ *x*. *f x* / *g x*) *F* (*at-right x*)
 for *x* :: *real*
 ⟨*proof*⟩

lemma *lhopital-left*:
 (*f* ⟶ *0*) (*at-left x*) ⟹ (*g* ⟶ *0*) (*at-left x*) ⟹
  eventually (λ*x*. *g x* ≠ *0*) (*at-left x*) ⟹
  eventually (λ*x*. *g′ x* ≠ *0*) (*at-left x*) ⟹
  eventually (λ*x*. *DERIV f x* :> *f′ x*) (*at-left x*) ⟹
  eventually (λ*x*. *DERIV g x* :> *g′ x*) (*at-left x*) ⟹
  filterlim (λ *x*. (*f′ x* / *g′ x*)) *F* (*at-left x*) ⟹
 filterlim (λ *x*. *f x* / *g x*) *F* (*at-left x*)
 for *x* :: *real*
 ⟨*proof*⟩

lemma *lhopital*:
 (*f* ⟶ *0*) (*at x*) ⟹ (*g* ⟶ *0*) (*at x*) ⟹
  eventually (λ*x*. *g x* ≠ *0*) (*at x*) ⟹
  eventually (λ*x*. *g′ x* ≠ *0*) (*at x*) ⟹
  eventually (λ*x*. *DERIV f x* :> *f′ x*) (*at x*) ⟹
  eventually (λ*x*. *DERIV g x* :> *g′ x*) (*at x*) ⟹
  filterlim (λ *x*. (*f′ x* / *g′ x*)) *F* (*at x*) ⟹
 filterlim (λ *x*. *f x* / *g x*) *F* (*at x*)
 for *x* :: *real*
 ⟨*proof*⟩

lemma *lhopital-right-0-at-top*:
 fixes *f g* :: *real* ⇒ *real*
 assumes *g-0*: *LIM x at-right 0*. *g x* :> *at-top*
  and *ev*:
   eventually (λ*x*. *g′ x* ≠ *0*) (*at-right 0*)
   eventually (λ*x*. *DERIV f x* :> *f′ x*) (*at-right 0*)
   eventually (λ*x*. *DERIV g x* :> *g′ x*) (*at-right 0*)
  and *lim*: ((λ *x*. (*f′ x* / *g′ x*)) ⟶ *x*) (*at-right 0*)
 shows ((λ *x*. *f x* / *g x*) ⟶ *x*) (*at-right 0*)
 ⟨*proof*⟩

lemma *lhopital-right-at-top*:
LIM x at-right x. (g::*real* ⇒ *real*) x :> *at-top* ⟹
eventually (λx. g' $x \neq 0$) (*at-right* x) ⟹
eventually (λx. *DERIV* f x :> f' x) (*at-right* x) ⟹
eventually (λx. *DERIV* g x :> g' x) (*at-right* x) ⟹
((λ x. (f' x / g' x)) ⟶ y) (*at-right* x) ⟹
((λ x. f x / g x) ⟶ y) (*at-right* x)
⟨*proof*⟩

lemma *lhopital-left-at-top*:
LIM x at-left x. g x :> *at-top* ⟹
eventually (λx. g' $x \neq 0$) (*at-left* x) ⟹
eventually (λx. *DERIV* f x :> f' x) (*at-left* x) ⟹
eventually (λx. *DERIV* g x :> g' x) (*at-left* x) ⟹
((λ x. (f' x / g' x)) ⟶ y) (*at-left* x) ⟹
((λ x. f x / g x) ⟶ y) (*at-left* x)
for x :: *real*
⟨*proof*⟩

lemma *lhopital-at-top*:
LIM x at x. (g::*real* ⇒ *real*) x :> *at-top* ⟹
eventually (λx. g' $x \neq 0$) (*at* x) ⟹
eventually (λx. *DERIV* f x :> f' x) (*at* x) ⟹
eventually (λx. *DERIV* g x :> g' x) (*at* x) ⟹
((λ x. (f' x / g' x)) ⟶ y) (*at* x) ⟹
((λ x. f x / g x) ⟶ y) (*at* x)
⟨*proof*⟩

lemma *lhospital-at-top-at-top*:
fixes f g :: *real* ⇒ *real*
assumes *g-0*: *LIM x at-top*. g x :> *at-top*
and g': *eventually* (λx. g' $x \neq 0$) *at-top*
and *Df*: *eventually* (λx. *DERIV* f x :> f' x) *at-top*
and *Dg*: *eventually* (λx. *DERIV* g x :> g' x) *at-top*
and *lim*: ((λ x. (f' x / g' x)) ⟶ x) *at-top*
shows ((λ x. f x / g x) ⟶ x) *at-top*
⟨*proof*⟩

lemma *lhopital-right-at-top-at-top*:
fixes f g :: *real* ⇒ *real*
assumes *f-0*: *LIM x at-right a*. f x :> *at-top*
assumes *g-0*: *LIM x at-right a*. g x :> *at-top*
and *ev*:
eventually (λx. *DERIV* f x :> f' x) (*at-right* a)
eventually (λx. *DERIV* g x :> g' x) (*at-right* a)
and *lim*: *filterlim* (λ x. (f' x / g' x)) *at-top* (*at-right* a)
shows *filterlim* (λ x. f x / g x) *at-top* (*at-right* a)
⟨*proof*⟩

lemma *lhopital-right-at-top-at-bot*:
  fixes *f g* :: *real* ⇒ *real*
  assumes *f-0*: *LIM x at-right a. f x :> at-top*
  assumes *g-0*: *LIM x at-right a. g x :> at-bot*
    and *ev*:
      eventually (λ*x*. *DERIV f x :> f′ x*) (*at-right a*)
      eventually (λ*x*. *DERIV g x :> g′ x*) (*at-right a*)
    and *lim*: *filterlim* (λ *x*. (*f′ x* / *g′ x*)) *at-bot* (*at-right a*)
  shows *filterlim* (λ *x*. *f x* / *g x*) *at-bot* (*at-right a*)
⟨*proof*⟩

lemma *lhopital-left-at-top-at-top*:
  fixes *f g* :: *real* ⇒ *real*
  assumes *f-0*: *LIM x at-left a. f x :> at-top*
  assumes *g-0*: *LIM x at-left a. g x :> at-top*
    and *ev*:
      eventually (λ*x*. *DERIV f x :> f′ x*) (*at-left a*)
      eventually (λ*x*. *DERIV g x :> g′ x*) (*at-left a*)
    and *lim*: *filterlim* (λ *x*. (*f′ x* / *g′ x*)) *at-top* (*at-left a*)
  shows *filterlim* (λ *x*. *f x* / *g x*) *at-top* (*at-left a*)
  ⟨*proof*⟩

lemma *lhopital-left-at-top-at-bot*:
  fixes *f g* :: *real* ⇒ *real*
  assumes *f-0*: *LIM x at-left a. f x :> at-top*
  assumes *g-0*: *LIM x at-left a. g x :> at-bot*
    and *ev*:
      eventually (λ*x*. *DERIV f x :> f′ x*) (*at-left a*)
      eventually (λ*x*. *DERIV g x :> g′ x*) (*at-left a*)
    and *lim*: *filterlim* (λ *x*. (*f′ x* / *g′ x*)) *at-bot* (*at-left a*)
  shows *filterlim* (λ *x*. *f x* / *g x*) *at-bot* (*at-left a*)
  ⟨*proof*⟩

lemma *lhopital-at-top-at-top*:
  fixes *f g* :: *real* ⇒ *real*
  assumes *f-0*: *LIM x at a. f x :> at-top*
  assumes *g-0*: *LIM x at a. g x :> at-top*
    and *ev*:
      eventually (λ*x*. *DERIV f x :> f′ x*) (*at a*)
      eventually (λ*x*. *DERIV g x :> g′ x*) (*at a*)
    and *lim*: *filterlim* (λ *x*. (*f′ x* / *g′ x*)) *at-top* (*at a*)
  shows *filterlim* (λ *x*. *f x* / *g x*) *at-top* (*at a*)
  ⟨*proof*⟩

lemma *lhopital-at-top-at-bot*:
  fixes *f g* :: *real* ⇒ *real*
  assumes *f-0*: *LIM x at a. f x :> at-top*
  assumes *g-0*: *LIM x at a. g x :> at-bot*

  and *ev*:
    eventually $(\lambda x.\ DERIV\ f\ x :> f'\ x)$ $(at\ a)$
    eventually $(\lambda x.\ DERIV\ g\ x :> g'\ x)$ $(at\ a)$
  and *lim*: *filterlim* $(\lambda\ x.\ (f'\ x\ /\ g'\ x))$ *at-bot* $(at\ a)$
 shows *filterlim* $(\lambda\ x.\ f\ x\ /\ g\ x)$ *at-bot* $(at\ a)$
 ⟨*proof*⟩

end

104 Nth Roots of Real Numbers

theory *NthRoot*
 imports *Deriv*
begin

104.1 Existence of Nth Root

Existence follows from the Intermediate Value Theorem

lemma *realpow-pos-nth*:
 fixes *a* :: *real*
 assumes *n*: $0 < n$
  and *a*: $0 < a$
 shows $\exists\, r>0.\ r\ \hat{}\ n = a$
⟨*proof*⟩

lemma *realpow-pos-nth2*: $(0::real) < a \Longrightarrow \exists\, r>0.\ r\ \hat{}\ Suc\ n = a$
 ⟨*proof*⟩

Uniqueness of nth positive root.

lemma *realpow-pos-nth-unique*: $0 < n \Longrightarrow 0 < a \Longrightarrow \exists!r.\ 0 < r \wedge r\ \hat{}\ n = a$
for *a* :: *real*
 ⟨*proof*⟩

104.2 Nth Root

We define roots of negative reals such that *root* n $(-\ x) = -$ *root* n x. This allows us to omit side conditions from many theorems.

lemma *inj-sgn-power*:
 assumes $0 < n$
 shows *inj* $(\lambda y.\ sgn\ y * |y|\hat{}n :: real)$
  (**is** *inj ?f*)
⟨*proof*⟩

lemma *sgn-power-injE*:
 $sgn\ a * |a|\ \hat{}\ n = x \Longrightarrow x = sgn\ b * |b|\ \hat{}\ n \Longrightarrow 0 < n \Longrightarrow a = b$
 for *a b* :: *real*

⟨*proof*⟩

definition *root* :: *nat* ⇒ *real* ⇒ *real*
where *root n x* = (*if n* = *0 then 0 else the-inv* (λ*y*. *sgn y* ∗ |*y*| ˆ*n*) *x*)

lemma *root-0* [*simp*]: *root 0 x* = *0*
⟨*proof*⟩

lemma *root-sgn-power*: *0* < *n* ⟹ *root n* (*sgn y* ∗ |*y*| ˆ*n*) = *y*
⟨*proof*⟩

lemma *sgn-power-root*:
assumes *0* < *n*
shows *sgn* (*root n x*) ∗ |(*root n x*)| ˆ*n* = *x*
(**is** *?f* (*root n x*) = *x*)
⟨*proof*⟩

lemma *split-root*: *P* (*root n x*) ⟷ (*n* = *0* ⟶ *P 0*) ∧ (*0* < *n* ⟶ (∀ *y*. *sgn y* ∗ |*y*| ˆ*n* = *x* ⟶ *P y*))
⟨*proof*⟩

lemma *real-root-zero* [*simp*]: *root n 0* = *0*
⟨*proof*⟩

lemma *real-root-minus*: *root n* (− *x*) = − *root n x*
⟨*proof*⟩

lemma *real-root-less-mono*: *0* < *n* ⟹ *x* < *y* ⟹ *root n x* < *root n y*
⟨*proof*⟩

lemma *real-root-gt-zero*: *0* < *n* ⟹ *0* < *x* ⟹ *0* < *root n x*
⟨*proof*⟩

lemma *real-root-ge-zero*: *0* ≤ *x* ⟹ *0* ≤ *root n x*
⟨*proof*⟩

lemma *real-root-pow-pos*: *0* < *n* ⟹ *0* < *x* ⟹ *root n x* ˆ *n* = *x*
⟨*proof*⟩

lemma *real-root-pow-pos2* [*simp*]: *0* < *n* ⟹ *0* ≤ *x* ⟹ *root n x* ˆ *n* = *x*
⟨*proof*⟩

lemma *sgn-root*: *0* < *n* ⟹ *sgn* (*root n x*) = *sgn x*
⟨*proof*⟩

lemma *odd-real-root-pow*: *odd n* ⟹ *root n x* ˆ *n* = *x*
⟨*proof*⟩

lemma *real-root-power-cancel*: *0* < *n* ⟹ *0* ≤ *x* ⟹ *root n* (*x* ˆ *n*) = *x*

⟨*proof*⟩

lemma *odd-real-root-power-cancel*: $odd\ n \Longrightarrow root\ n\ (x\ \hat{}\ n) = x$
⟨*proof*⟩

lemma *real-root-pos-unique*: $0 < n \Longrightarrow 0 \leq y \Longrightarrow y\ \hat{}\ n = x \Longrightarrow root\ n\ x = y$
⟨*proof*⟩

lemma *odd-real-root-unique*: $odd\ n \Longrightarrow y\ \hat{}\ n = x \Longrightarrow root\ n\ x = y$
⟨*proof*⟩

lemma *real-root-one* [*simp*]: $0 < n \Longrightarrow root\ n\ 1 = 1$
⟨*proof*⟩

Root function is strictly monotonic, hence injective.

lemma *real-root-le-mono*: $0 < n \Longrightarrow x \leq y \Longrightarrow root\ n\ x \leq root\ n\ y$
⟨*proof*⟩

lemma *real-root-less-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x < root\ n\ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *real-root-le-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x \leq root\ n\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *real-root-eq-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x = root\ n\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemmas *real-root-gt-0-iff* [*simp*] = *real-root-less-iff* [**where** $x=0$, *simplified*]
lemmas *real-root-lt-0-iff* [*simp*] = *real-root-less-iff* [**where** $y=0$, *simplified*]
lemmas *real-root-ge-0-iff* [*simp*] = *real-root-le-iff* [**where** $x=0$, *simplified*]
lemmas *real-root-le-0-iff* [*simp*] = *real-root-le-iff* [**where** $y=0$, *simplified*]
lemmas *real-root-eq-0-iff* [*simp*] = *real-root-eq-iff* [**where** $y=0$, *simplified*]

lemma *real-root-gt-1-iff* [*simp*]: $0 < n \Longrightarrow 1 < root\ n\ y \longleftrightarrow 1 < y$
⟨*proof*⟩

lemma *real-root-lt-1-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x < 1 \longleftrightarrow x < 1$
⟨*proof*⟩

lemma *real-root-ge-1-iff* [*simp*]: $0 < n \Longrightarrow 1 \leq root\ n\ y \longleftrightarrow 1 \leq y$
⟨*proof*⟩

lemma *real-root-le-1-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x \leq 1 \longleftrightarrow x \leq 1$
⟨*proof*⟩

lemma *real-root-eq-1-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x = 1 \longleftrightarrow x = 1$
⟨*proof*⟩

Roots of multiplication and division.

lemma *real-root-mult*: $root\ n\ (x * y) = root\ n\ x * root\ n\ y$
⟨*proof*⟩

lemma *real-root-inverse*: $root\ n\ (inverse\ x) = inverse\ (root\ n\ x)$
⟨*proof*⟩

lemma *real-root-divide*: $root\ n\ (x\ /\ y) = root\ n\ x\ /\ root\ n\ y$
⟨*proof*⟩

lemma *real-root-abs*: $0 < n \Longrightarrow root\ n\ |x| = |root\ n\ x|$
⟨*proof*⟩

lemma *real-root-power*: $0 < n \Longrightarrow root\ n\ (x\ \hat{}\ k) = root\ n\ x\ \hat{}\ k$
⟨*proof*⟩

Roots of roots.

lemma *real-root-Suc-0* [*simp*]: $root\ (Suc\ 0)\ x = x$
⟨*proof*⟩

lemma *real-root-mult-exp*: $root\ (m * n)\ x = root\ m\ (root\ n\ x)$
⟨*proof*⟩

lemma *real-root-commute*: $root\ m\ (root\ n\ x) = root\ n\ (root\ m\ x)$
⟨*proof*⟩

Monotonicity in first argument.

lemma *real-root-strict-decreasing*:
assumes $0 < n\ n < N\ 1 < x$
shows $root\ N\ x < root\ n\ x$
⟨*proof*⟩

lemma *real-root-strict-increasing*:
assumes $0 < n\ n < N\ 0 < x\ x < 1$
shows $root\ n\ x < root\ N\ x$
⟨*proof*⟩

lemma *real-root-decreasing*: $0 < n \Longrightarrow n < N \Longrightarrow 1 \leq x \Longrightarrow root\ N\ x \leq root\ n\ x$
⟨*proof*⟩

lemma *real-root-increasing*: $0 < n \Longrightarrow n < N \Longrightarrow 0 \leq x \Longrightarrow x \leq 1 \Longrightarrow root\ n\ x \leq root\ N\ x$
⟨*proof*⟩

Continuity and derivatives.

lemma *isCont-real-root*: $isCont\ (root\ n)\ x$
⟨*proof*⟩

lemma *tendsto-real-root* [*tendsto-intros*]:

$(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ root\ n\ (f\ x)) \longrightarrow root\ n\ x)\ F$
⟨proof⟩

lemma *continuous-real-root* [*continuous-intros*]:
$continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ root\ n\ (f\ x))$
⟨proof⟩

lemma *continuous-on-real-root* [*continuous-intros*]:
$continuous\text{-}on\ s\ f \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ root\ n\ (f\ x))$
⟨proof⟩

lemma *DERIV-real-root*:
assumes n: $0 < n$
and x: $0 < x$
shows $DERIV\ (root\ n)\ x :> inverse\ (real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0))$
⟨proof⟩

lemma *DERIV-odd-real-root*:
assumes n: $odd\ n$
and x: $x \neq 0$
shows $DERIV\ (root\ n)\ x :> inverse\ (real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0))$
⟨proof⟩

lemma *DERIV-even-real-root*:
assumes n: $0 < n$
and $even\ n$
and x: $x < 0$
shows $DERIV\ (root\ n)\ x :> inverse\ (-\ real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0))$
⟨proof⟩

lemma *DERIV-real-root-generic*:
assumes $0 < n$
and $x \neq 0$
and $even\ n \Longrightarrow 0 < x \Longrightarrow D = inverse\ (real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0))$
and $even\ n \Longrightarrow x < 0 \Longrightarrow D = -\ inverse\ (real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0))$
and $odd\ n \Longrightarrow D = inverse\ (real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0))$
shows $DERIV\ (root\ n)\ x :> D$
⟨proof⟩

104.3 Square Root

definition $sqrt :: real \Rightarrow real$
where $sqrt = root\ 2$

lemma *pos2*: $0 < (2::nat)$
⟨proof⟩

lemma *real-sqrt-unique*: $y^2 = x \Longrightarrow 0 \leq y \Longrightarrow sqrt\ x = y$
⟨proof⟩

lemma *real-sqrt-abs* [*simp*]: $sqrt\ (x^2) = |x|$
⟨*proof*⟩

lemma *real-sqrt-pow2* [*simp*]: $0 \leq x \Longrightarrow (sqrt\ x)^2 = x$
⟨*proof*⟩

lemma *real-sqrt-pow2-iff* [*simp*]: $(sqrt\ x)^2 = x \longleftrightarrow 0 \leq x$
⟨*proof*⟩

lemma *real-sqrt-zero* [*simp*]: $sqrt\ 0 = 0$
⟨*proof*⟩

lemma *real-sqrt-one* [*simp*]: $sqrt\ 1 = 1$
⟨*proof*⟩

lemma *real-sqrt-four* [*simp*]: $sqrt\ 4 = 2$
⟨*proof*⟩

lemma *real-sqrt-minus*: $sqrt\ (-\ x) = -\ sqrt\ x$
⟨*proof*⟩

lemma *real-sqrt-mult*: $sqrt\ (x * y) = sqrt\ x * sqrt\ y$
⟨*proof*⟩

lemma *real-sqrt-mult-self* [*simp*]: $sqrt\ a * sqrt\ a = |a|$
⟨*proof*⟩

lemma *real-sqrt-inverse*: $sqrt\ (inverse\ x) = inverse\ (sqrt\ x)$
⟨*proof*⟩

lemma *real-sqrt-divide*: $sqrt\ (x\ /\ y) = sqrt\ x\ /\ sqrt\ y$
⟨*proof*⟩

lemma *real-sqrt-power*: $sqrt\ (x\ \hat{}\ k) = sqrt\ x\ \hat{}\ k$
⟨*proof*⟩

lemma *real-sqrt-gt-zero*: $0 < x \Longrightarrow 0 < sqrt\ x$
⟨*proof*⟩

lemma *real-sqrt-ge-zero*: $0 \leq x \Longrightarrow 0 \leq sqrt\ x$
⟨*proof*⟩

lemma *real-sqrt-less-mono*: $x < y \Longrightarrow sqrt\ x < sqrt\ y$
⟨*proof*⟩

lemma *real-sqrt-le-mono*: $x \leq y \Longrightarrow sqrt\ x \leq sqrt\ y$
⟨*proof*⟩

lemma *real-sqrt-less-iff* [*simp*]: $sqrt\ x < sqrt\ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *real-sqrt-le-iff* [*simp*]: $sqrt\ x \le sqrt\ y \longleftrightarrow x \le y$
⟨*proof*⟩

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

lemma *real-less-lsqrt*: $0 \le x \Longrightarrow 0 \le y \Longrightarrow x < y^2 \Longrightarrow sqrt\ x < y$
⟨*proof*⟩

lemma *real-le-lsqrt*: $0 \le x \Longrightarrow 0 \le y \Longrightarrow x \le y^2 \Longrightarrow sqrt\ x \le y$
⟨*proof*⟩

lemma *real-le-rsqrt*: $x^2 \le y \Longrightarrow x \le sqrt\ y$
⟨*proof*⟩

lemma *real-less-rsqrt*: $x^2 < y \Longrightarrow x < sqrt\ y$
⟨*proof*⟩

lemma *real-sqrt-power-even*:
assumes $even\ n\ x \ge 0$
shows $sqrt\ x\ \hat{}\ n = x\ \hat{}\ (n\ div\ 2)$
⟨*proof*⟩

lemma *sqrt-le-D*: $sqrt\ x \le y \Longrightarrow x \le y^2$
⟨*proof*⟩

lemma *sqrt-even-pow2*:
assumes n: $even\ n$
shows $sqrt\ (2\ \hat{}\ n) = 2\ \hat{}\ (n\ div\ 2)$
⟨*proof*⟩

lemmas *real-sqrt-gt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** $x=0$, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-lt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** $y=0$, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-ge-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** $x=0$, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-le-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** $y=0$, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-eq-0-iff* [*simp*] = *real-sqrt-eq-iff* [**where** $y=0$, *unfolded real-sqrt-zero*]

lemmas *real-sqrt-gt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** $x=1$, *unfolded real-sqrt-one*]
lemmas *real-sqrt-lt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** $y=1$, *unfolded real-sqrt-one*]
lemmas *real-sqrt-ge-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** $x=1$, *unfolded real-sqrt-one*]
lemmas *real-sqrt-le-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** $y=1$, *unfolded real-sqrt-one*]
lemmas *real-sqrt-eq-1-iff* [*simp*] = *real-sqrt-eq-iff* [**where** $y=1$, *unfolded real-sqrt-one*]

lemma *sqrt-add-le-add-sqrt*:
assumes $0 \le x\ 0 \le y$
shows $sqrt\ (x + y) \le sqrt\ x + sqrt\ y$

⟨proof⟩

lemma *isCont-real-sqrt*: *isCont sqrt x*
⟨proof⟩

lemma *tendsto-real-sqrt* [*tendsto-intros*]:
$(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ sqrt\ (f\ x)) \longrightarrow sqrt\ x)\ F$
⟨proof⟩

lemma *continuous-real-sqrt* [*continuous-intros*]:
continuous F f $\Longrightarrow$ *continuous F* $(\lambda x.\ sqrt\ (f\ x))$
⟨proof⟩

lemma *continuous-on-real-sqrt* [*continuous-intros*]:
continuous-on s f $\Longrightarrow$ *continuous-on s* $(\lambda x.\ sqrt\ (f\ x))$
⟨proof⟩

lemma *DERIV-real-sqrt-generic*:
assumes $x \neq 0$
and $x > 0 \Longrightarrow D = inverse\ (sqrt\ x)\ /\ 2$
and $x < 0 \Longrightarrow D = -\ inverse\ (sqrt\ x)\ /\ 2$
shows *DERIV sqrt x :> D*
⟨proof⟩

lemma *DERIV-real-sqrt*: $0 < x \Longrightarrow$ *DERIV sqrt x :> inverse (sqrt x) / 2*
⟨proof⟩

declare
DERIV-real-sqrt-generic[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-real-root-generic[*THEN DERIV-chain2*, *derivative-intros*]

lemma *not-real-square-gt-zero* [*simp*]: $\neg\ 0 < x * x \longleftrightarrow x = 0$
for *x* :: *real*
⟨proof⟩

lemma *real-sqrt-abs2* [*simp*]: $sqrt\ (x * x) = |x|$
⟨proof⟩

lemma *real-inv-sqrt-pow2*: $0 < x \Longrightarrow (inverse\ (sqrt\ x))^2 = inverse\ x$
⟨proof⟩

lemma *real-sqrt-eq-zero-cancel*: $0 \le x \Longrightarrow sqrt\ x = 0 \Longrightarrow x = 0$
⟨proof⟩

lemma *real-sqrt-ge-one*: $1 \le x \Longrightarrow 1 \le sqrt\ x$
⟨proof⟩

lemma *sqrt-divide-self-eq*:
assumes *nneg*: $0 \le x$

shows *sqrt x / x = inverse (sqrt x)*
⟨*proof*⟩

lemma *real-div-sqrt*: $0 \leq x \Longrightarrow x\ /\ sqrt\ x = sqrt\ x$
⟨*proof*⟩

lemma *real-divide-square-eq* [*simp*]: $(r * a)\ /\ (r * r) = a\ /\ r$
for *a r* :: *real*
⟨*proof*⟩

lemma *lemma-real-divide-sqrt-less*: $0 < u \Longrightarrow u\ /\ sqrt\ 2 < u$
⟨*proof*⟩

lemma *four-x-squared*: $4 * x^2 = (2 * x)^2$
for *x* :: *real*
⟨*proof*⟩

lemma *sqrt-at-top*: *LIM x at-top. sqrt x :: real :> at-top*
⟨*proof*⟩

104.4 Square Root of Sum of Squares

lemma *sum-squares-bound*: $2 * x * y \leq x^2 + y^2$
for *x y* :: *'a::linordered-field*
⟨*proof*⟩

lemma *arith-geo-mean*:
fixes *u* :: *'a::linordered-field*
assumes $u^2 = x * y$ $x \geq 0$ $y \geq 0$
shows $u \leq (x + y)/2$
⟨*proof*⟩

lemma *arith-geo-mean-sqrt*:
fixes *x* :: *real*
assumes $x \geq 0$ $y \geq 0$
shows $sqrt\ (x * y) \leq (x + y)/2$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-mult-ge-zero* [*simp*]: $0 \leq sqrt\ ((x^2 + y^2) * (xa^2 + ya^2))$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-mult-squared-eq* [*simp*]:
$(sqrt\ ((x^2 + y^2) * (xa^2 + ya^2)))^2 = (x^2 + y^2) * (xa^2 + ya^2)$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-eq-cancel*: $sqrt\ (x^2 + y^2) = x \Longrightarrow y = 0$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-eq-cancel2*: $sqrt\ (x^2 + y^2) = y \Longrightarrow x = 0$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-ge1* [*simp*]: $x \le sqrt\ (x^2 + y^2)$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-ge2* [*simp*]: $y \le sqrt\ (x^2 + y^2)$
⟨*proof*⟩

lemma *real-sqrt-ge-abs1* [*simp*]: $|x| \le sqrt\ (x^2 + y^2)$
⟨*proof*⟩

lemma *real-sqrt-ge-abs2* [*simp*]: $|y| \le sqrt\ (x^2 + y^2)$
⟨*proof*⟩

lemma *le-real-sqrt-sumsq* [*simp*]: $x \le sqrt\ (x * x + y * y)$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-triangle-ineq*:
$sqrt\ ((a + c)^2 + (b + d)^2) \le sqrt\ (a^2 + b^2) + sqrt\ (c^2 + d^2)$
⟨*proof*⟩

lemma *real-sqrt-sum-squares-less*: $|x| < u\ /\ sqrt\ 2 \Longrightarrow |y| < u\ /\ sqrt\ 2 \Longrightarrow sqrt\ (x^2 + y^2) < u$
⟨*proof*⟩

lemma *sqrt2-less-2*: $sqrt\ 2 < (2::real)$
⟨*proof*⟩

lemma *sqrt-sum-squares-half-less*:
$x < u/2 \Longrightarrow y < u/2 \Longrightarrow 0 \le x \Longrightarrow 0 \le y \Longrightarrow sqrt\ (x^2 + y^2) < u$
⟨*proof*⟩

lemma *LIMSEQ-root*: $(\lambda n.\ root\ n\ n) \longrightarrow 1$
⟨*proof*⟩

lemma *LIMSEQ-root-const*:
assumes $0 < c$
shows $(\lambda n.\ root\ n\ c) \longrightarrow 1$
⟨*proof*⟩

Legacy theorem names:

lemmas *real-root-pos2* = *real-root-power-cancel*
lemmas *real-root-pos-pos* = *real-root-gt-zero* [*THEN order-less-imp-le*]
lemmas *real-root-pos-pos-le* = *real-root-ge-zero*
lemmas *real-sqrt-mult-distrib* = *real-sqrt-mult*
lemmas *real-sqrt-mult-distrib2* = *real-sqrt-mult*
lemmas *real-sqrt-eq-zero-cancel-iff* = *real-sqrt-eq-0-iff*

end

105 Power Series, Transcendental Functions etc.

theory *Transcendental*
imports *Series Deriv NthRoot*
begin

A fact theorem on reals.

lemma *square-fact-le-2-fact*: $fact\ n * fact\ n \le (fact\ (2 * n) :: real)$
$\langle proof \rangle$

lemma *fact-in-Reals*: $fact\ n \in \mathbb{R}$
$\langle proof \rangle$

lemma *of-real-fact* [*simp*]: $of\text{-}real\ (fact\ n) = fact\ n$
$\langle proof \rangle$

lemma *pochhammer-of-real*: $pochhammer\ (of\text{-}real\ x)\ n = of\text{-}real\ (pochhammer\ x\ n)$
$\langle proof \rangle$

lemma *norm-fact* [*simp*]: $norm\ (fact\ n :: 'a::real\text{-}normed\text{-}algebra\text{-}1) = fact\ n$
$\langle proof \rangle$

lemma *root-test-convergence*:
fixes $f :: nat \Rightarrow 'a::banach$
assumes f: $(\lambda n.\ root\ n\ (norm\ (f\ n))) \longrightarrow x$ — could be weakened to lim sup

and $x < 1$
shows *summable f*
$\langle proof \rangle$

105.1 More facts about binomial coefficients

These facts could have been proven before, but having real numbers makes the proofs a lot easier.

lemma *central-binomial-odd*:
$odd\ n \Longrightarrow n\ choose\ (Suc\ (n\ div\ 2)) = n\ choose\ (n\ div\ 2)$
$\langle proof \rangle$

lemma *binomial-less-binomial-Suc*:
assumes k: $k < n\ div\ 2$
shows $n\ choose\ k < n\ choose\ (Suc\ k)$
$\langle proof \rangle$

lemma *binomial-strict-mono*:
assumes $k < k'$ $2{*}k' \le n$

shows n *choose* $k < n$ *choose* k'
⟨*proof*⟩

lemma *binomial-mono*:
assumes $k \leq k'$ $2*k' \leq n$
shows n *choose* $k \leq n$ *choose* k'
⟨*proof*⟩

lemma *binomial-strict-antimono*:
assumes $k < k'$ $2 * k \geq n$ $k' \leq n$
shows n *choose* $k > n$ *choose* k'
⟨*proof*⟩

lemma *binomial-antimono*:
assumes $k \leq k'$ $k \geq n$ *div* 2 $k' \leq n$
shows n *choose* $k \geq n$ *choose* k'
⟨*proof*⟩

lemma *binomial-maximum*: n *choose* $k \leq n$ *choose* $(n$ *div* $2)$
⟨*proof*⟩

lemma *binomial-maximum'*: $(2*n)$ *choose* $k \leq (2*n)$ *choose* n
⟨*proof*⟩

lemma *central-binomial-lower-bound*:
assumes $n > 0$
shows $4\hat{}n$ / $(2*$*real* $n) \leq$ *real* $((2*n)$ *choose* $n)$
⟨*proof*⟩

105.2 Properties of Power Series

lemma *powser-zero* [*simp*]: $(\sum n.\ f\ n * 0\ \hat{}\ n) = f\ 0$
for f :: *nat* ⇒ *'a::real-normed-algebra-1*
⟨*proof*⟩

lemma *powser-sums-zero*: $(\lambda n.\ a\ n * 0\hat{}n)$ *sums* $a\ 0$
for a :: *nat* ⇒ *'a::real-normed-div-algebra*
⟨*proof*⟩

lemma *powser-sums-zero-iff* [*simp*]: $(\lambda n.\ a\ n * 0\hat{}n)$ *sums* $x \longleftrightarrow a\ 0 = x$
for a :: *nat* ⇒ *'a::real-normed-div-algebra*
⟨*proof*⟩

Power series has a circle or radius of convergence: if it sums for x, then it sums absolutely for z with $|z| < |x|$.

lemma *powser-insidea*:
fixes x z :: *'a::real-normed-div-algebra*
assumes *1*: *summable* $(\lambda n.\ f\ n * x\hat{}n)$
and *2*: *norm* $z <$ *norm* x

shows *summable* ($\lambda n.$ *norm* (f n $*$ z ^ n))
⟨*proof*⟩

lemma *powser-inside*:
fixes f :: *nat* ⇒ $'a$::{*real-normed-div-algebra*,*banach*}
shows
summable ($\lambda n.$ f n $*$ (x^n)) ⟹ *norm* z < *norm* x ⟹
summable ($\lambda n.$ f n $*$ (z ^ n))
⟨*proof*⟩

lemma *powser-times-n-limit-0*:
fixes x :: $'a$::{*real-normed-div-algebra*,*banach*}
assumes *norm* x < *1*
shows ($\lambda n.$ *of-nat* n $*$ x ^ n) ⟶ *0*
⟨*proof*⟩

corollary *lim-n-over-pown*:
fixes x :: $'a$::{*real-normed-field*,*banach*}
shows *1* < *norm* x ⟹ (($\lambda n.$ *of-nat* n / x^n) ⟶ *0*) *sequentially*
⟨*proof*⟩

lemma *sum-split-even-odd*:
fixes f :: *nat* ⇒ *real*
shows ($\sum$ i<*2* $*$ n. *if even* i *then* f i *else* g i) = ($\sum$ i<n. f (*2* $*$ i)) + ($\sum$ i<n. g (*2* $*$ i + *1*))
⟨*proof*⟩

lemma *sums-if'*:
fixes g :: *nat* ⇒ *real*
assumes g *sums* x
shows (λ n. *if even* n *then* *0* *else* g ((n − *1*) *div* *2*)) *sums* x
⟨*proof*⟩

lemma *sums-if*:
fixes g :: *nat* ⇒ *real*
assumes g *sums* x **and** f *sums* y
shows (λ n. *if even* n *then* f (n *div* *2*) *else* g ((n − *1*) *div* *2*)) *sums* (x + y)
⟨*proof*⟩

105.3 Alternating series test / Leibniz formula

lemma *sums-alternating-upper-lower*:
fixes a :: *nat* ⇒ *real*
assumes *mono*: $\bigwedge n.$ a (*Suc* n) ≤ a n
and *a-pos*: $\bigwedge n.$ *0* ≤ a n
and a ⟶ *0*
shows ∃ l. ((∀ n. ($\sum$ i<*2*$*$$n$. (− *1*)^$i$$*$$a$ i) ≤ l) ∧ (λ n. $\sum$ i<*2*$*$$n$. (− *1*)^$i$$*$$a$ i) ⟶ l) ∧
((∀ n. l ≤ ($\sum$ i<*2*$*$$n$ + *1*. (− *1*)^$i$$*$$a$ i)) ∧ (λ n. $\sum$ i<*2*$*$$n$ + *1*. (−

$1)\hat{\ }i*a\ i) \longrightarrow l)$
(**is** $\exists\, l.\ ((\forall\, n.\ ?f\ n \le l) \wedge \text{-}) \wedge ((\forall\, n.\ l \le ?g\ n) \wedge \text{-}))$
$\langle proof \rangle$

lemma *summable-Leibniz'*:
fixes $a :: nat \Rightarrow real$
assumes *a-zero*: $a \longrightarrow 0$
and *a-pos*: $\bigwedge n.\ 0 \le a\ n$
and *a-monotone*: $\bigwedge n.\ a\ (Suc\ n) \le a\ n$
shows *summable*: $summable\ (\lambda\ n.\ (-1)\hat{\ }n * a\ n)$
and $\bigwedge n.\ (\sum i{<}2{*}n.\ (-1)\hat{\ }i{*}a\ i) \le (\sum i.\ (-1)\hat{\ }i{*}a\ i)$
and $(\lambda n.\ \sum i{<}2{*}n.\ (-1)\hat{\ }i{*}a\ i) \longrightarrow (\sum i.\ (-1)\hat{\ }i{*}a\ i)$
and $\bigwedge n.\ (\sum i.\ (-1)\hat{\ }i{*}a\ i) \le (\sum i{<}2{*}n{+}1.\ (-1)\hat{\ }i{*}a\ i)$
and $(\lambda n.\ \sum i{<}2{*}n{+}1.\ (-1)\hat{\ }i{*}a\ i) \longrightarrow (\sum i.\ (-1)\hat{\ }i{*}a\ i)$
$\langle proof \rangle$

theorem *summable-Leibniz*:
fixes $a :: nat \Rightarrow real$
assumes *a-zero*: $a \longrightarrow 0$
and *monoseq a*
shows $summable\ (\lambda\ n.\ (-1)\hat{\ }n * a\ n)$ (**is** *?summable*)
and $0 < a\ 0 \longrightarrow$
$(\forall\, n.\ (\sum i.\ (-\ 1)\hat{\ }i{*}a\ i) \in \{\ \sum i{<}2{*}n.\ (-\ 1)\hat{\ }i * a\ i\ ..\ \sum i{<}2{*}n{+}1.\ (-\ 1)\hat{\ }i * a\ i\})$ (**is** *?pos*)
and $a\ 0 < 0 \longrightarrow$
$(\forall\, n.\ (\sum i.\ (-\ 1)\hat{\ }i{*}a\ i) \in \{\ \sum i{<}2{*}n{+}1.\ (-\ 1)\hat{\ }i * a\ i\ ..\ \sum i{<}2{*}n.\ (-\ 1)\hat{\ }i * a\ i\})$ (**is** *?neg*)
and $(\lambda n.\ \sum i{<}2{*}n.\ (-\ 1)\hat{\ }i{*}a\ i) \longrightarrow (\sum i.\ (-\ 1)\hat{\ }i{*}a\ i)$ (**is** *?f*)
and $(\lambda n.\ \sum i{<}2{*}n{+}1.\ (-\ 1)\hat{\ }i{*}a\ i) \longrightarrow (\sum i.\ (-\ 1)\hat{\ }i{*}a\ i)$ (**is** *?g*)
$\langle proof \rangle$

105.4 Term-by-Term Differentiability of Power Series

definition *diffs* $:: (nat \Rightarrow 'a{::}ring\text{-}1) \Rightarrow nat \Rightarrow 'a$
where $diffs\ c = (\lambda n.\ of\text{-}nat\ (Suc\ n) * c\ (Suc\ n))$

Lemma about distributing negation over it.

lemma *diffs-minus*: $diffs\ (\lambda n.\ -\ c\ n) = (\lambda n.\ -\ diffs\ c\ n)$
$\langle proof \rangle$

lemma *diffs-equiv*:
fixes $x :: 'a{::}\{real\text{-}normed\text{-}vector, ring\text{-}1\}$
shows $summable\ (\lambda n.\ diffs\ c\ n * x\hat{\ }n) \Longrightarrow$
$(\lambda n.\ of\text{-}nat\ n * c\ n * x\hat{\ }(n - Suc\ 0))\ sums\ (\sum n.\ diffs\ c\ n * x\hat{\ }n)$
$\langle proof \rangle$

lemma *lemma-termdiff1*:
fixes $z :: 'a :: \{monoid\text{-}mult, comm\text{-}ring\}$
shows $(\sum p{<}m.\ (((z + h)\ \hat{\ }\ (m - p)) * (z\ \hat{\ }\ p)) - (z\ \hat{\ }\ m)) =$

$(\sum p<m.\ (z\ \hat{}\ p) * (((z + h)\ \hat{}\ (m - p)) - (z\ \hat{}\ (m - p))))$
⟨proof⟩

lemma *sumr-diff-mult-const2*: $sum\ f\ \{..<n\} - of\text{-}nat\ n * r = (\sum i<n.\ f\ i - r)$
for r :: $'a$::*ring-1*
⟨proof⟩

lemma *lemma-realpow-rev-sumr*:
$(\sum p<Suc\ n.\ (x\ \hat{}\ p) * (y\ \hat{}\ (n - p))) = (\sum p<Suc\ n.\ (x\ \hat{}\ (n - p)) * (y\ \hat{}\ p))$
⟨proof⟩

lemma *lemma-termdiff2*:
fixes h :: $'a$::*field*
assumes h: $h \neq 0$
shows $((z + h)\ \hat{}\ n - z\ \hat{}\ n)\ /\ h - of\text{-}nat\ n * z\ \hat{}\ (n - Suc\ 0) =$
$h * (\sum p< n - Suc\ 0.\ \sum q< n - Suc\ 0 - p.\ (z + h)\ \hat{}\ q * z\ \hat{}\ (n - 2 - q))$
(**is** *?lhs* = *?rhs*)
⟨proof⟩

lemma *real-sum-nat-ivl-bounded2*:
fixes K :: $'a$::*linordered-semidom*
assumes f: $\bigwedge p$::*nat*. $p < n \Longrightarrow f\ p \leq K$
and K: $0 \leq K$
shows $sum\ f\ \{..<n-k\} \leq of\text{-}nat\ n * K$
⟨proof⟩

lemma *lemma-termdiff3*:
fixes $h\ z$ :: $'a$::*real-normed-field*
assumes *1*: $h \neq 0$
and *2*: $norm\ z \leq K$
and *3*: $norm\ (z + h) \leq K$
shows $norm\ (((z + h)\ \hat{}\ n - z\ \hat{}\ n)\ /\ h - of\text{-}nat\ n * z\ \hat{}\ (n - Suc\ 0)) \leq$
$of\text{-}nat\ n * of\text{-}nat\ (n - Suc\ 0) * K\ \hat{}\ (n - 2) * norm\ h$
⟨proof⟩

lemma *lemma-termdiff4*:
fixes f :: $'a$::*real-normed-vector* ⇒ $'b$::*real-normed-vector*
and k :: *real*
assumes k: $0 < k$
and *le*: $\bigwedge h.\ h \neq 0 \Longrightarrow norm\ h < k \Longrightarrow norm\ (f\ h) \leq K * norm\ h$
shows $f\ -0\rightarrow\ 0$
⟨proof⟩

lemma *lemma-termdiff5*:
fixes g :: $'a$::*real-normed-vector* ⇒ *nat* ⇒ $'b$::*banach*
and k :: *real*
assumes k: $0 < k$
and f: *summable* f
and *le*: $\bigwedge h\ n.\ h \neq 0 \Longrightarrow norm\ h < k \Longrightarrow norm\ (g\ h\ n) \leq f\ n * norm\ h$

shows $(\lambda h.\ suminf\ (g\ h))\ -0\rightarrow\ 0$
$\langle proof \rangle$

lemma *termdiffs-aux*:
fixes x :: $'a$::{*real-normed-field*,*banach*}
assumes *1*: $summable\ (\lambda n.\ diffs\ (diffs\ c)\ n * K\ \hat{}\ n)$
and *2*: $norm\ x < norm\ K$
shows $(\lambda h.\ \sum n.\ c\ n * (((x + h)\ \hat{}\ n - x\hat{}n) / h - of\text{-}nat\ n * x\ \hat{}\ (n - Suc\ 0)))\ -0\rightarrow\ 0$
$\langle proof \rangle$

lemma *termdiffs*:
fixes $K\ x$ :: $'a$::{*real-normed-field*,*banach*}
assumes *1*: $summable\ (\lambda n.\ c\ n * K\ \hat{}\ n)$
and *2*: $summable\ (\lambda n.\ (diffs\ c)\ n * K\ \hat{}\ n)$
and *3*: $summable\ (\lambda n.\ (diffs\ (diffs\ c))\ n * K\ \hat{}\ n)$
and *4*: $norm\ x < norm\ K$
shows $DERIV\ (\lambda x.\ \sum n.\ c\ n * x\hat{}n)\ x :> (\sum n.\ (diffs\ c)\ n * x\hat{}n)$
$\langle proof \rangle$

105.5 The Derivative of a Power Series Has the Same Radius of Convergence

lemma *termdiff-converges*:
fixes x :: $'a$::{*real-normed-field*,*banach*}
assumes *K*: $norm\ x < K$
and *sm*: $\bigwedge x.\ norm\ x < K \Longrightarrow summable(\lambda n.\ c\ n * x\ \hat{}\ n)$
shows $summable\ (\lambda n.\ diffs\ c\ n * x\ \hat{}\ n)$
$\langle proof \rangle$

lemma *termdiff-converges-all*:
fixes x :: $'a$::{*real-normed-field*,*banach*}
assumes $\bigwedge x.\ summable\ (\lambda n.\ c\ n * x\hat{}n)$
shows $summable\ (\lambda n.\ diffs\ c\ n * x\hat{}n)$
$\langle proof \rangle$

lemma *termdiffs-strong*:
fixes $K\ x$ :: $'a$::{*real-normed-field*,*banach*}
assumes *sm*: $summable\ (\lambda n.\ c\ n * K\ \hat{}\ n)$
and *K*: $norm\ x < norm\ K$
shows $DERIV\ (\lambda x.\ \sum n.\ c\ n * x\hat{}n)\ x :> (\sum n.\ diffs\ c\ n * x\hat{}n)$
$\langle proof \rangle$

lemma *termdiffs-strong-converges-everywhere*:
fixes $K\ x$ :: $'a$::{*real-normed-field*,*banach*}
assumes $\bigwedge y.\ summable\ (\lambda n.\ c\ n * y\ \hat{}\ n)$

shows $((\lambda x.\ \sum n.\ c\ n * x\hat{\ }n)$ *has-field-derivative* $(\sum n.\ diffs\ c\ n * x\hat{\ }n))\ (at\ x)$
⟨*proof*⟩

lemma *termdiffs-strong'*:
fixes z :: $'a$ :: {*real-normed-field*,*banach*}
assumes $\bigwedge z.\ norm\ z < K \Longrightarrow summable\ (\lambda n.\ c\ n * z\ \hat{\ }\ n)$
assumes $norm\ z < K$
shows $((\lambda z.\ \sum n.\ c\ n * z\hat{\ }n)$ *has-field-derivative* $(\sum n.\ diffs\ c\ n * z\hat{\ }n))\ (at\ z)$
⟨*proof*⟩

lemma *termdiffs-sums-strong*:
fixes z :: $'a$ :: {*banach*,*real-normed-field*}
assumes *sums*: $\bigwedge z.\ norm\ z < K \Longrightarrow (\lambda n.\ c\ n * z\ \hat{\ }\ n)\ sums\ f\ z$
assumes *deriv*: $(f$ *has-field-derivative* $f')\ (at\ z)$
assumes *norm*: $norm\ z < K$
shows $(\lambda n.\ diffs\ c\ n * z\ \hat{\ }\ n)\ sums\ f'$
⟨*proof*⟩

lemma *isCont-powser*:
fixes $K\ x$ :: $'a$::{*real-normed-field*,*banach*}
assumes $summable\ (\lambda n.\ c\ n * K\ \hat{\ }\ n)$
assumes $norm\ x < norm\ K$
shows $isCont\ (\lambda x.\ \sum n.\ c\ n * x\hat{\ }n)\ x$
⟨*proof*⟩

lemmas *isCont-powser'* = *isCont-o2*[*OF* - *isCont-powser*]

lemma *isCont-powser-converges-everywhere*:
fixes $K\ x$ :: $'a$::{*real-normed-field*,*banach*}
assumes $\bigwedge y.\ summable\ (\lambda n.\ c\ n * y\ \hat{\ }\ n)$
shows $isCont\ (\lambda x.\ \sum n.\ c\ n * x\hat{\ }n)\ x$
⟨*proof*⟩

lemma *powser-limit-0*:
fixes a :: $nat \Rightarrow {'a}$::{*real-normed-field*,*banach*}
assumes *s*: $0 < s$
and *sm*: $\bigwedge x.\ norm\ x < s \Longrightarrow (\lambda n.\ a\ n * x\ \hat{\ }\ n)\ sums\ (f\ x)$
shows $(f \longrightarrow a\ 0)\ (at\ 0)$
⟨*proof*⟩

lemma *powser-limit-0-strong*:
fixes a :: $nat \Rightarrow {'a}$::{*real-normed-field*,*banach*}
assumes *s*: $0 < s$
and *sm*: $\bigwedge x.\ x \neq 0 \Longrightarrow norm\ x < s \Longrightarrow (\lambda n.\ a\ n * x\ \hat{\ }\ n)\ sums\ (f\ x)$
shows $(f \longrightarrow a\ 0)\ (at\ 0)$
⟨*proof*⟩

105.6 Derivability of power series

lemma *DERIV-series'*:
  fixes $f :: real \Rightarrow nat \Rightarrow real$
  assumes *DERIV-f*: $\bigwedge n.\ DERIV\ (\lambda\ x.\ f\ x\ n)\ x0 :> (f'\ x0\ n)$
    and *allf-summable*: $\bigwedge x.\ x \in \{a <..< b\} \Longrightarrow summable\ (f\ x)$
    and *x0-in-I*: $x0 \in \{a <..< b\}$
    and $summable\ (f'\ x0)$
    and $summable\ L$
    and *L-def*: $\bigwedge n\ x\ y.\ x \in \{a <..< b\} \Longrightarrow y \in \{a <..< b\} \Longrightarrow |f\ x\ n - f\ y\ n| \leq L\ n * |x - y|$
  shows $DERIV\ (\lambda\ x.\ suminf\ (f\ x))\ x0 :> (suminf\ (f'\ x0))$
  ⟨*proof*⟩

lemma *DERIV-power-series'*:
  fixes $f :: nat \Rightarrow real$
  assumes *converges*: $\bigwedge x.\ x \in \{-R <..< R\} \Longrightarrow summable\ (\lambda n.\ f\ n * real\ (Suc\ n) * x\hat{\ }n)$
    and *x0-in-I*: $x0 \in \{-R <..< R\}$
    and $0 < R$
  shows $DERIV\ (\lambda x.\ (\sum n.\ f\ n * x\hat{\ }(Suc\ n)))\ x0 :> (\sum n.\ f\ n * real\ (Suc\ n) * x0\hat{\ }n)$
    (**is** $DERIV\ (\lambda x.\ suminf\ (?f\ x))\ x0 :> suminf\ (?f'\ x0)$)
⟨*proof*⟩

lemma *geometric-deriv-sums*:
  fixes $z :: 'a :: \{real\text{-}normed\text{-}field,banach\}$
  assumes $norm\ z < 1$
  shows $(\lambda n.\ of\text{-}nat\ (Suc\ n) * z\ \hat{\ }\ n)\ sums\ (1\ /\ (1 - z)\hat{\ }2)$
⟨*proof*⟩

lemma *isCont-pochhammer* [*continuous-intros*]: $isCont\ (\lambda z.\ pochhammer\ z\ n)\ z$
  for $z :: 'a::real\text{-}normed\text{-}field$
  ⟨*proof*⟩

lemma *continuous-on-pochhammer* [*continuous-intros*]: $continuous\text{-}on\ A\ (\lambda z.\ pochhammer\ z\ n)$
  for $A :: 'a::real\text{-}normed\text{-}field\ set$
  ⟨*proof*⟩

lemmas *continuous-on-pochhammer'* [*continuous-intros*] =
  continuous-on-compose2[*OF continuous-on-pochhammer - subset-UNIV*]

105.7 Exponential Function

definition $exp :: 'a \Rightarrow 'a::\{real\text{-}normed\text{-}algebra\text{-}1,banach\}$
  where $exp = (\lambda x.\ \sum n.\ x\hat{\ }n\ /_R\ fact\ n)$

lemma *summable-exp-generic*:
  fixes $x :: 'a::\{real\text{-}normed\text{-}algebra\text{-}1,banach\}$

defines *S-def*: $S \equiv \lambda n.\ x\hat{\ }n\ /_R\ \mathit{fact}\ n$
shows *summable S*
⟨*proof*⟩

lemma *summable-norm-exp*: *summable* ($\lambda n.$ *norm* ($x\hat{\ }n\ /_R$ *fact n*))
for x :: $'a$::{*real-normed-algebra-1*,*banach*}
⟨*proof*⟩

lemma *summable-exp*: *summable* ($\lambda n.$ *inverse* (*fact n*) $*$ $x\hat{\ }n$)
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *exp-converges*: ($\lambda n.\ x\hat{\ }n\ /_R$ *fact n*) *sums exp x*
⟨*proof*⟩

lemma *exp-fdiffs*:
diffs ($\lambda n.$ *inverse* (*fact n*)) = ($\lambda n.$ *inverse* (*fact n* :: $'a$::{*real-normed-field*,*banach*}))
⟨*proof*⟩

lemma *diffs-of-real*: *diffs* ($\lambda n.$ *of-real* (*f n*)) = ($\lambda n.$ *of-real* (*diffs f n*))
⟨*proof*⟩

lemma *DERIV-exp* [*simp*]: *DERIV exp x* :> *exp x*
⟨*proof*⟩

declare *DERIV-exp*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-exp*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemma *norm-exp*: *norm* (*exp x*) $\le$ *exp* (*norm x*)
⟨*proof*⟩

lemma *isCont-exp*: *isCont exp x*
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-exp'* [*simp*]: *isCont f a* $\Longrightarrow$ *isCont* ($\lambda x.$ *exp* (*f x*)) *a*
for f :: - $\Rightarrow$$'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *tendsto-exp* [*tendsto-intros*]: ($f \longrightarrow a$) F $\Longrightarrow$ (($\lambda x.$ *exp* (*f x*)) $\longrightarrow$ *exp* *a*) F
for f:: - $\Rightarrow$$'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-exp* [*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* ($\lambda x.$ *exp* (*f x*))
for f :: - $\Rightarrow$$'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-on-exp* [*continuous-intros*]: *continuous-on s f* $\Longrightarrow$ *continuous-on s* ($\lambda x.$ *exp* (*f x*))
for *f* :: - $\Rightarrow$'*a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

105.7.1 Properties of the Exponential Function

lemma *exp-zero* [*simp*]: *exp 0* = *1*
⟨*proof*⟩

lemma *exp-series-add-commuting*:
fixes *x y* :: '*a*::{*real-normed-algebra-1*,*banach*}
defines *S-def*: $S \equiv \lambda x\ n.\ x\hat{\ }n\ /_R\ fact\ n$
assumes *comm*: $x * y = y * x$
shows $S\ (x + y)\ n = (\sum i{\le}n.\ S\ x\ i * S\ y\ (n - i))$
⟨*proof*⟩

lemma *exp-add-commuting*: $x * y = y * x \Longrightarrow exp\ (x + y) = exp\ x * exp\ y$
⟨*proof*⟩

lemma *exp-times-arg-commute*: $exp\ A * A = A * exp\ A$
⟨*proof*⟩

lemma *exp-add*: $exp\ (x + y) = exp\ x * exp\ y$
for *x y* :: '*a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *exp-double*: $exp(2 * z) = exp\ z\ \hat{\ }\ 2$
⟨*proof*⟩

lemmas *mult-exp-exp* = *exp-add* [*symmetric*]

lemma *exp-of-real*: *exp* (*of-real x*) = *of-real* (*exp x*)
⟨*proof*⟩

lemmas *of-real-exp* = *exp-of-real*[*symmetric*]

corollary *exp-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow exp\ z \in \mathbb{R}$
⟨*proof*⟩

lemma *exp-not-eq-zero* [*simp*]: $exp\ x \neq 0$
⟨*proof*⟩

lemma *exp-minus-inverse*: $exp\ x * exp\ (-\ x) = 1$
⟨*proof*⟩

lemma *exp-minus*: $exp\ (-\ x) = inverse\ (exp\ x)$
for *x* :: '*a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *exp-diff*: $exp\ (x - y) = exp\ x\ /\ exp\ y$
for $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
⟨*proof*⟩

lemma *exp-of-nat-mult*: $exp\ (of\text{-}nat\ n * x) = exp\ x\ \hat{}\ n$
for $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
⟨*proof*⟩

corollary *exp-of-nat2-mult*: $exp\ (x * of\text{-}nat\ n) = exp\ x\ \hat{}\ n$
for $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
⟨*proof*⟩

lemma *exp-sum*: $finite\ I \Longrightarrow exp\ (sum\ f\ I) = prod\ (\lambda x.\ exp\ (f\ x))\ I$
⟨*proof*⟩

lemma *exp-divide-power-eq*:
fixes $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
assumes $n > 0$
shows $exp\ (x\ /\ of\text{-}nat\ n)\ \hat{}\ n = exp\ x$
⟨*proof*⟩

105.7.2 Properties of the Exponential Function on Reals

Comparisons of *exp x* with zero.

Proof: because every exponential can be seen as a square.

lemma *exp-ge-zero* [*simp*]: $0 \le exp\ x$
for $x :: real$
⟨*proof*⟩

lemma *exp-gt-zero* [*simp*]: $0 < exp\ x$
for $x :: real$
⟨*proof*⟩

lemma *not-exp-less-zero* [*simp*]: $\neg\ exp\ x < 0$
for $x :: real$
⟨*proof*⟩

lemma *not-exp-le-zero* [*simp*]: $\neg\ exp\ x \le 0$
for $x :: real$
⟨*proof*⟩

lemma *abs-exp-cancel* [*simp*]: $|exp\ x| = exp\ x$
for $x :: real$
⟨*proof*⟩

Strict monotonicity of exponential.

lemma *exp-ge-add-one-self-aux*:

fixes x :: *real*
assumes $0 \leq x$
shows $1 + x \leq exp\ x$
⟨*proof*⟩

lemma *exp-gt-one*: $0 < x \Longrightarrow 1 < exp\ x$
for x :: *real*
⟨*proof*⟩

lemma *exp-less-mono*:
fixes $x\ y$ :: *real*
assumes $x < y$
shows $exp\ x < exp\ y$
⟨*proof*⟩

lemma *exp-less-cancel*: $exp\ x < exp\ y \Longrightarrow x < y$
for $x\ y$ :: *real*
⟨*proof*⟩

lemma *exp-less-cancel-iff* [*iff*]: $exp\ x < exp\ y \longleftrightarrow x < y$
for $x\ y$ :: *real*
⟨*proof*⟩

lemma *exp-le-cancel-iff* [*iff*]: $exp\ x \leq exp\ y \longleftrightarrow x \leq y$
for $x\ y$ :: *real*
⟨*proof*⟩

lemma *exp-inj-iff* [*iff*]: $exp\ x = exp\ y \longleftrightarrow x = y$
for $x\ y$ :: *real*
⟨*proof*⟩

Comparisons of *exp x* with one.

lemma *one-less-exp-iff* [*simp*]: $1 < exp\ x \longleftrightarrow 0 < x$
for x :: *real*
⟨*proof*⟩

lemma *exp-less-one-iff* [*simp*]: $exp\ x < 1 \longleftrightarrow x < 0$
for x :: *real*
⟨*proof*⟩

lemma *one-le-exp-iff* [*simp*]: $1 \leq exp\ x \longleftrightarrow 0 \leq x$
for x :: *real*
⟨*proof*⟩

lemma *exp-le-one-iff* [*simp*]: $exp\ x \leq 1 \longleftrightarrow x \leq 0$
for x :: *real*
⟨*proof*⟩

lemma *exp-eq-one-iff* [*simp*]: $exp\ x = 1 \longleftrightarrow x = 0$

for x :: *real*
⟨*proof*⟩

lemma *lemma-exp-total*: $1 \le y \Longrightarrow \exists x.\ 0 \le x \wedge x \le y - 1 \wedge exp\ x = y$
for y :: *real*
⟨*proof*⟩

lemma *exp-total*: $0 < y \Longrightarrow \exists x.\ exp\ x = y$
for y :: *real*
⟨*proof*⟩

105.8 Natural Logarithm

class *ln* = *real-normed-algebra-1* + *banach* +
fixes *ln* :: $'a \Rightarrow 'a$
assumes *ln-one* [*simp*]: $ln\ 1 = 0$

definition *powr* :: $'a \Rightarrow 'a \Rightarrow 'a::ln$ (**infixr** *powr 80*)
— exponentation via ln and exp
where [*code del*]: $x\ powr\ a \equiv if\ x = 0\ then\ 0\ else\ exp\ (a * ln\ x)$

lemma *powr-0* [*simp*]: $0\ powr\ z = 0$
⟨*proof*⟩

instantiation *real* :: *ln*
begin

definition *ln-real* :: $real \Rightarrow real$
where $ln\text{-}real\ x = (THE\ u.\ exp\ u = x)$

instance
⟨*proof*⟩

end

lemma *powr-eq-0-iff* [*simp*]: $w\ powr\ z = 0 \longleftrightarrow w = 0$
⟨*proof*⟩

lemma *ln-exp* [*simp*]: $ln\ (exp\ x) = x$
for x :: *real*
⟨*proof*⟩

lemma *exp-ln* [*simp*]: $0 < x \Longrightarrow exp\ (ln\ x) = x$
for x :: *real*
⟨*proof*⟩

lemma *exp-ln-iff* [*simp*]: $exp\ (ln\ x) = x \longleftrightarrow 0 < x$
for x :: *real*

⟨*proof*⟩

lemma *ln-unique*: $exp\ y = x \Longrightarrow ln\ x = y$
for $x :: real$
⟨*proof*⟩

lemma *ln-mult*: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ (x * y) = ln\ x + ln\ y$
for $x :: real$
⟨*proof*⟩

lemma *ln-prod*: $finite\ I \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i > 0) \Longrightarrow ln\ (prod\ f\ I) = sum\ (\lambda x.\ ln(f\ x))\ I$
for $f :: 'a \Rightarrow real$
⟨*proof*⟩

lemma *ln-inverse*: $0 < x \Longrightarrow ln\ (inverse\ x) = -\ ln\ x$
for $x :: real$
⟨*proof*⟩

lemma *ln-div*: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ (x\ /\ y) = ln\ x - ln\ y$
for $x :: real$
⟨*proof*⟩

lemma *ln-realpow*: $0 < x \Longrightarrow ln\ (x\hat{}n) = real\ n * ln\ x$
⟨*proof*⟩

lemma *ln-less-cancel-iff* [*simp*]: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ x < ln\ y \longleftrightarrow x < y$
for $x :: real$
⟨*proof*⟩

lemma *ln-le-cancel-iff* [*simp*]: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ x \le ln\ y \longleftrightarrow x \le y$
for $x :: real$
⟨*proof*⟩

lemma *ln-inj-iff* [*simp*]: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ x = ln\ y \longleftrightarrow x = y$
for $x :: real$
⟨*proof*⟩

lemma *ln-add-one-self-le-self*: $0 \le x \Longrightarrow ln\ (1 + x) \le x$
for $x :: real$
⟨*proof*⟩

lemma *ln-less-self* [*simp*]: $0 < x \Longrightarrow ln\ x < x$
for $x :: real$
⟨*proof*⟩

lemma *ln-ge-iff*: $\bigwedge x{::}real.\ 0 < x \Longrightarrow y \le ln\ x \longleftrightarrow exp\ y \le x$
⟨*proof*⟩

lemma *ln-ge-zero* [*simp*]: $1 \leq x \Longrightarrow 0 \leq ln\ x$
for x :: *real*
⟨*proof*⟩

lemma *ln-ge-zero-imp-ge-one*: $0 \leq ln\ x \Longrightarrow 0 < x \Longrightarrow 1 \leq x$
for x :: *real*
⟨*proof*⟩

lemma *ln-ge-zero-iff* [*simp*]: $0 < x \Longrightarrow 0 \leq ln\ x \longleftrightarrow 1 \leq x$
for x :: *real*
⟨*proof*⟩

lemma *ln-less-zero-iff* [*simp*]: $0 < x \Longrightarrow ln\ x < 0 \longleftrightarrow x < 1$
for x :: *real*
⟨*proof*⟩

lemma *ln-le-zero-iff* [*simp*]: $0 < x \Longrightarrow ln\ x \leq 0 \longleftrightarrow x \leq 1$
for x :: *real*
⟨*proof*⟩

lemma *ln-gt-zero*: $1 < x \Longrightarrow 0 < ln\ x$
for x :: *real*
⟨*proof*⟩

lemma *ln-gt-zero-imp-gt-one*: $0 < ln\ x \Longrightarrow 0 < x \Longrightarrow 1 < x$
for x :: *real*
⟨*proof*⟩

lemma *ln-gt-zero-iff* [*simp*]: $0 < x \Longrightarrow 0 < ln\ x \longleftrightarrow 1 < x$
for x :: *real*
⟨*proof*⟩

lemma *ln-eq-zero-iff* [*simp*]: $0 < x \Longrightarrow ln\ x = 0 \longleftrightarrow x = 1$
for x :: *real*
⟨*proof*⟩

lemma *ln-less-zero*: $0 < x \Longrightarrow x < 1 \Longrightarrow ln\ x < 0$
for x :: *real*
⟨*proof*⟩

lemma *ln-neg-is-const*: $x \leq 0 \Longrightarrow ln\ x = (THE\ x.\ False)$
for x :: *real*
⟨*proof*⟩

lemma *isCont-ln*:
fixes x :: *real*
assumes $x \neq 0$
shows *isCont ln x*
⟨*proof*⟩

lemma *tendsto-ln* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow a \neq 0 \Longrightarrow ((\lambda x.\ ln\ (f\ x)) \longrightarrow ln\ a)\ F$
for $a :: real$
⟨*proof*⟩

lemma *continuous-ln*:
$continuous\ F\ f \Longrightarrow f\ (Lim\ F\ (\lambda x.\ x)) \neq 0 \Longrightarrow continuous\ F\ (\lambda x.\ ln\ (f\ x :: real))$
⟨*proof*⟩

lemma *isCont-ln'* [*continuous-intros*]:
$continuous\ (at\ x)\ f \Longrightarrow f\ x \neq 0 \Longrightarrow continuous\ (at\ x)\ (\lambda x.\ ln\ (f\ x :: real))$
⟨*proof*⟩

lemma *continuous-within-ln* [*continuous-intros*]:
$continuous\ (at\ x\ within\ s)\ f \Longrightarrow f\ x \neq 0 \Longrightarrow continuous\ (at\ x\ within\ s)\ (\lambda x.\ ln\ (f\ x :: real))$
⟨*proof*⟩

lemma *continuous-on-ln* [*continuous-intros*]:
$continuous\text{-}on\ s\ f \Longrightarrow (\forall x \in s.\ f\ x \neq 0) \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ ln\ (f\ x :: real))$
⟨*proof*⟩

lemma *DERIV-ln*: $0 < x \Longrightarrow DERIV\ ln\ x :> inverse\ x$
for $x :: real$
⟨*proof*⟩

lemma *DERIV-ln-divide*: $0 < x \Longrightarrow DERIV\ ln\ x :> 1\ /\ x$
for $x :: real$
⟨*proof*⟩

declare *DERIV-ln-divide*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-ln-divide*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemma *ln-series*:
assumes $0 < x$ **and** $x < 2$
shows $ln\ x = (\sum\ n.\ (-1)\hat{\ }n * (1\ /\ real\ (n + 1)) * (x - 1)\hat{\ }(Suc\ n))$
(**is** $ln\ x = suminf\ (?f\ (x - 1))$)
⟨*proof*⟩

lemma *exp-first-terms*:
fixes $x :: 'a::\{real\text{-}normed\text{-}algebra\text{-}1, banach\}$
shows $exp\ x = (\sum n<k.\ inverse(fact\ n) *_R (x\ \hat{}\ n)) + (\sum n.\ inverse(fact\ (n + k)) *_R (x\ \hat{}\ (n + k)))$
⟨*proof*⟩

lemma *exp-first-term*: $exp\ x = 1 + (\sum n.\ inverse\ (fact\ (Suc\ n)) *_R (x\ \hat{}\ Suc\ n))$
for $x :: 'a::\{real\text{-}normed\text{-}algebra\text{-}1, banach\}$

⟨*proof*⟩

lemma *exp-first-two-terms*: $exp\ x = 1 + x + (\sum n.\ inverse\ (fact\ (n + 2)) *_R (x \hat{}\ (n + 2)))$
for x :: $'a$::{*real-normed-algebra-1*,*banach*}
⟨*proof*⟩

lemma *exp-bound*:
fixes x :: *real*
assumes a: $0 \leq x$
and b: $x \leq 1$
shows $exp\ x \leq 1 + x + x^2$
⟨*proof*⟩

corollary *exp-half-le2*: $exp(1/2) \leq (2::real)$
⟨*proof*⟩

corollary *exp-le*: $exp\ 1 \leq (3::real)$
⟨*proof*⟩

lemma *exp-bound-half*: $norm\ z \leq 1/2 \Longrightarrow norm\ (exp\ z) \leq 2$
⟨*proof*⟩

lemma *exp-bound-lemma*:
assumes $norm\ z \leq 1/2$
shows $norm\ (exp\ z) \leq 1 + 2 * norm\ z$
⟨*proof*⟩

lemma *real-exp-bound-lemma*: $0 \leq x \Longrightarrow x \leq 1/2 \Longrightarrow exp\ x \leq 1 + 2 * x$
for x :: *real*
⟨*proof*⟩

lemma *ln-one-minus-pos-upper-bound*:
fixes x :: *real*
assumes a: $0 \leq x$ **and** b: $x < 1$
shows $ln\ (1 - x) \leq - x$
⟨*proof*⟩

lemma *exp-ge-add-one-self* [*simp*]: $1 + x \leq exp\ x$
for x :: *real*
⟨*proof*⟩

lemma *ln-one-plus-pos-lower-bound*:
fixes x :: *real*
assumes a: $0 \leq x$ **and** b: $x \leq 1$
shows $x - x^2 \leq ln\ (1 + x)$
⟨*proof*⟩

lemma *ln-one-minus-pos-lower-bound*:

fixes x :: *real*
assumes a: $0 \leq x$ **and** b: $x \leq 1 / 2$
shows $- x - 2 * x^2 \leq ln (1 - x)$
⟨*proof*⟩

lemma *ln-add-one-self-le-self2*:
fixes x :: *real*
shows $-1 < x \Longrightarrow ln (1 + x) \leq x$
⟨*proof*⟩

lemma *abs-ln-one-plus-x-minus-x-bound-nonneg*:
fixes x :: *real*
assumes x: $0 \leq x$ **and** $x1$: $x \leq 1$
shows $|ln (1 + x) - x| \leq x^2$
⟨*proof*⟩

lemma *abs-ln-one-plus-x-minus-x-bound-nonpos*:
fixes x :: *real*
assumes a: $-(1 / 2) \leq x$ **and** b: $x \leq 0$
shows $|ln (1 + x) - x| \leq 2 * x^2$
⟨*proof*⟩

lemma *abs-ln-one-plus-x-minus-x-bound*:
fixes x :: *real*
shows $|x| \leq 1 / 2 \Longrightarrow |ln (1 + x) - x| \leq 2 * x^2$
⟨*proof*⟩

lemma *ln-x-over-x-mono*:
fixes x :: *real*
assumes x: $exp\ 1 \leq x$ $x \leq y$
shows $ln\ y / y \leq ln\ x / x$
⟨*proof*⟩

lemma *ln-le-minus-one*: $0 < x \Longrightarrow ln\ x \leq x - 1$
for x :: *real*
⟨*proof*⟩

corollary *ln-diff-le*: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ x - ln\ y \leq (x - y) / y$
for x :: *real*
⟨*proof*⟩

lemma *ln-eq-minus-one*:
fixes x :: *real*
assumes $0 < x$ $ln\ x = x - 1$
shows $x = 1$
⟨*proof*⟩

lemma *ln-x-over-x-tendsto-0*: $((\lambda x::real.\ ln\ x / x) \longrightarrow 0)$ *at-top*
⟨*proof*⟩

lemma *exp-ge-one-plus-x-over-n-power-n*:
  assumes $x \geq - \mathit{real}\ n\ n > 0$
  shows $(1 + x\ /\ \mathit{of\text{-}nat}\ n)\ \hat{}\ n \leq \exp x$
⟨*proof*⟩

lemma *exp-ge-one-minus-x-over-n-power-n*:
  assumes $x \leq \mathit{real}\ n\ n > 0$
  shows $(1 - x\ /\ \mathit{of\text{-}nat}\ n)\ \hat{}\ n \leq \exp (-x)$
  ⟨*proof*⟩

lemma *exp-at-bot*: $(\exp \longrightarrow (0{::}\mathit{real}))\ \mathit{at\text{-}bot}$
  ⟨*proof*⟩

lemma *exp-at-top*: $\mathit{LIM}\ x\ \mathit{at\text{-}top}.\ \exp x :: \mathit{real} :> \mathit{at\text{-}top}$
  ⟨*proof*⟩

lemma *lim-exp-minus-1*: $((\lambda z{::}'a.\ (\exp(z) - 1)\ /\ z) \longrightarrow 1)\ (\mathit{at}\ 0)$
  for $x :: 'a{::}\{\mathit{real\text{-}normed\text{-}field},\mathit{banach}\}$
⟨*proof*⟩

lemma *ln-at-0*: $\mathit{LIM}\ x\ \mathit{at\text{-}right}\ 0.\ \ln (x{::}\mathit{real}) :> \mathit{at\text{-}bot}$
  ⟨*proof*⟩

lemma *ln-at-top*: $\mathit{LIM}\ x\ \mathit{at\text{-}top}.\ \ln (x{::}\mathit{real}) :> \mathit{at\text{-}top}$
  ⟨*proof*⟩

lemma *filtermap-ln-at-top*: $\mathit{filtermap}\ (\ln{::}\mathit{real} \Rightarrow \mathit{real})\ \mathit{at\text{-}top} = \mathit{at\text{-}top}$
  ⟨*proof*⟩

lemma *filtermap-exp-at-top*: $\mathit{filtermap}\ (\exp{::}\mathit{real} \Rightarrow \mathit{real})\ \mathit{at\text{-}top} = \mathit{at\text{-}top}$
  ⟨*proof*⟩

lemma *filtermap-ln-at-right*: $\mathit{filtermap}\ \ln\ (\mathit{at\text{-}right}\ (0{::}\mathit{real})) = \mathit{at\text{-}bot}$
  ⟨*proof*⟩

lemma *tendsto-power-div-exp-0*: $((\lambda x.\ x\ \hat{}\ k\ /\ \exp x) \longrightarrow (0{::}\mathit{real}))\ \mathit{at\text{-}top}$
⟨*proof*⟩

105.8.1 A couple of simple bounds

lemma *exp-plus-inverse-exp*:
  fixes $x{::}\mathit{real}$
  shows $2 \leq \exp x + \mathit{inverse}\ (\exp x)$
⟨*proof*⟩

lemma *real-le-x-sinh*:
  fixes $x{::}\mathit{real}$
  assumes $0 \leq x$

shows $x \le (exp\ x - inverse(exp\ x)) / 2$
⟨*proof*⟩

lemma *real-le-abs-sinh*:
fixes $x::real$
shows $abs\ x \le abs((exp\ x - inverse(exp\ x)) / 2)$
⟨*proof*⟩

105.9 The general logarithm

definition $log :: real \Rightarrow real \Rightarrow real$
— logarithm of x to base a
where $log\ a\ x = ln\ x / ln\ a$

lemma *tendsto-log* [*tendsto-intros*]:
$(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow 0 < a \Longrightarrow a \ne 1 \Longrightarrow 0 < b \Longrightarrow$
$((\lambda x.\ log\ (f\ x)\ (g\ x)) \longrightarrow log\ a\ b)\ F$
⟨*proof*⟩

lemma *continuous-log*:
assumes $continuous\ F\ f$
and $continuous\ F\ g$
and $0 < f\ (Lim\ F\ (\lambda x.\ x))$
and $f\ (Lim\ F\ (\lambda x.\ x)) \ne 1$
and $0 < g\ (Lim\ F\ (\lambda x.\ x))$
shows $continuous\ F\ (\lambda x.\ log\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *continuous-at-within-log*[*continuous-intros*]:
assumes $continuous\ (at\ a\ within\ s)\ f$
and $continuous\ (at\ a\ within\ s)\ g$
and $0 < f\ a$
and $f\ a \ne 1$
and $0 < g\ a$
shows $continuous\ (at\ a\ within\ s)\ (\lambda x.\ log\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *isCont-log*[*continuous-intros*, *simp*]:
assumes $isCont\ f\ a\ isCont\ g\ a\ 0 < f\ a\ f\ a \ne 1\ 0 < g\ a$
shows $isCont\ (\lambda x.\ log\ (f\ x)\ (g\ x))\ a$
⟨*proof*⟩

lemma *continuous-on-log*[*continuous-intros*]:
assumes $continuous\text{-}on\ s\ f\ continuous\text{-}on\ s\ g$
and $\forall x \in s.\ 0 < f\ x\ \forall x \in s.\ f\ x \ne 1\ \forall x \in s.\ 0 < g\ x$
shows $continuous\text{-}on\ s\ (\lambda x.\ log\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *powr-one-eq-one* [*simp*]: $1\ powr\ a = 1$

⟨*proof*⟩

lemma *powr-zero-eq-one* [*simp*]: x *powr* $0 = ($*if* $x = 0$ *then* 0 *else* $1)$
⟨*proof*⟩

lemma *powr-one-gt-zero-iff* [*simp*]: x *powr* $1 = x \longleftrightarrow 0 \leq x$
for x :: *real*
⟨*proof*⟩
declare *powr-one-gt-zero-iff* [*THEN iffD2*, *simp*]

lemma *powr-diff*:
fixes w:: $'a$::{*ln*,*real-normed-field*} **shows** w *powr* $(z1 - z2) = w$ *powr* $z1$ / w *powr* $z2$
⟨*proof*⟩

lemma *powr-mult*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow (x * y)$ *powr* $a = (x$ *powr* $a) * (y$ *powr* $a)$
for $a\ x\ y$ :: *real*
⟨*proof*⟩

lemma *powr-ge-pzero* [*simp*]: $0 \leq x$ *powr* y
for $x\ y$ :: *real*
⟨*proof*⟩

lemma *powr-divide*: $0 < x \Longrightarrow 0 < y \Longrightarrow (x / y)$ *powr* $a = (x$ *powr* $a) / (y$ *powr* $a)$
for $a\ b\ x$ :: *real*
⟨*proof*⟩

lemma *powr-add*: x *powr* $(a + b) = (x$ *powr* $a) * (x$ *powr* $b)$
for $a\ b\ x$ :: $'a$::{*ln*,*real-normed-field*}
⟨*proof*⟩

lemma *powr-mult-base*: $0 < x \Longrightarrow x * x$ *powr* $y = x$ *powr* $(1 + y)$
for x :: *real*
⟨*proof*⟩

lemma *powr-powr*: $(x$ *powr* $a)$ *powr* $b = x$ *powr* $(a * b)$
for $a\ b\ x$ :: *real*
⟨*proof*⟩

lemma *powr-powr-swap*: $(x$ *powr* $a)$ *powr* $b = (x$ *powr* $b)$ *powr* a
for $a\ b\ x$ :: *real*
⟨*proof*⟩

lemma *powr-minus*: x *powr* $(- a) =$ *inverse* $(x$ *powr* $a)$
for $a\ x$ :: $'a$::{*ln*,*real-normed-field*}
⟨*proof*⟩

lemma *powr-minus-divide*: $x\ powr\ (-\ a) = 1/(x\ powr\ a)$
for $x\ a$ :: *real*
⟨*proof*⟩

lemma *divide-powr-uminus*: $a\ /\ b\ powr\ c = a * b\ powr\ (-\ c)$
for $a\ b\ c$ :: *real*
⟨*proof*⟩

lemma *powr-less-mono*: $a < b \Longrightarrow 1 < x \Longrightarrow x\ powr\ a < x\ powr\ b$
for $a\ b\ x$ :: *real*
⟨*proof*⟩

lemma *powr-less-cancel*: $x\ powr\ a < x\ powr\ b \Longrightarrow 1 < x \Longrightarrow a < b$
for $a\ b\ x$ :: *real*
⟨*proof*⟩

lemma *powr-less-cancel-iff* [*simp*]: $1 < x \Longrightarrow x\ powr\ a < x\ powr\ b \longleftrightarrow a < b$
for $a\ b\ x$ :: *real*
⟨*proof*⟩

lemma *powr-le-cancel-iff* [*simp*]: $1 < x \Longrightarrow x\ powr\ a \leq x\ powr\ b \longleftrightarrow a \leq b$
for $a\ b\ x$ :: *real*
⟨*proof*⟩

lemma *powr-realpow*: $0 < x \Longrightarrow x\ powr\ (real\ n) = x\hat{}n$
⟨*proof*⟩

lemma *log-ln*: $ln\ x = log\ (exp(1))\ x$
⟨*proof*⟩

lemma *DERIV-log*:
assumes $x > 0$
shows $DERIV\ (\lambda y.\ log\ b\ y)\ x :> 1\ /\ (ln\ b * x)$
⟨*proof*⟩

lemmas *DERIV-log*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-log*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemma *powr-log-cancel* [*simp*]: $0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow a\ powr\ (log\ a\ x)$
$= x$
⟨*proof*⟩

lemma *log-powr-cancel* [*simp*]: $0 < a \Longrightarrow a \neq 1 \Longrightarrow log\ a\ (a\ powr\ y) = y$
⟨*proof*⟩

lemma *log-mult*:
$0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow$
$log\ a\ (x * y) = log\ a\ x + log\ a\ y$
⟨*proof*⟩

lemma *log-eq-div-ln-mult-log*:
$0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow$
$log\ a\ x = (ln\ b/ln\ a) * log\ b\ x$
⟨*proof*⟩

Base 10 logarithms

lemma *log-base-10-eq1*: $0 < x \Longrightarrow log\ 10\ x = (ln\ (exp\ 1)\ /\ ln\ 10) * ln\ x$
⟨*proof*⟩

lemma *log-base-10-eq2*: $0 < x \Longrightarrow log\ 10\ x = (log\ 10\ (exp\ 1)) * ln\ x$
⟨*proof*⟩

lemma *log-one* [*simp*]: $log\ a\ 1 = 0$
⟨*proof*⟩

lemma *log-eq-one* [*simp*]: $0 < a \Longrightarrow a \neq 1 \Longrightarrow log\ a\ a = 1$
⟨*proof*⟩

lemma *log-inverse*: $0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow log\ a\ (inverse\ x) = -\ log\ a\ x$
⟨*proof*⟩

lemma *log-divide*: $0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow log\ a\ (x/y) = log\ a\ x - log\ a\ y$
⟨*proof*⟩

lemma *powr-gt-zero* [*simp*]: $0 < x\ powr\ a \longleftrightarrow x \neq 0$
for $a\ x$ :: *real*
⟨*proof*⟩

lemma *log-add-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow log\ b\ x + y = log\ b\ (x * b\ powr\ y)$
and *add-log-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow y + log\ b\ x = log\ b\ (b\ powr\ y * x)$
and *log-minus-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow log\ b\ x - y = log\ b\ (x * b\ powr\ -y)$
and *minus-log-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow y - log\ b\ x = log\ b\ (b\ powr\ y\ /\ x)$
⟨*proof*⟩

lemma *log-less-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow log\ a\ x < log\ a\ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *log-inj*:
assumes $1 < b$
shows *inj-on* (*log b*) $\{0 <..\}$
⟨*proof*⟩

lemma *log-le-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow \log a\ x \leq \log a\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *zero-less-log-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 < \log a\ x \longleftrightarrow 1 < x$
⟨*proof*⟩

lemma *zero-le-log-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 \leq \log a\ x \longleftrightarrow 1 \leq x$
⟨*proof*⟩

lemma *log-less-zero-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow \log a\ x < 0 \longleftrightarrow x < 1$
⟨*proof*⟩

lemma *log-le-zero-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow \log a\ x \leq 0 \longleftrightarrow x \leq 1$
⟨*proof*⟩

lemma *one-less-log-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 1 < \log a\ x \longleftrightarrow a < x$
⟨*proof*⟩

lemma *one-le-log-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 1 \leq \log a\ x \longleftrightarrow a \leq x$
⟨*proof*⟩

lemma *log-less-one-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow \log a\ x < 1 \longleftrightarrow x < a$
⟨*proof*⟩

lemma *log-le-one-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow \log a\ x \leq 1 \longleftrightarrow x \leq a$
⟨*proof*⟩

lemma *le-log-iff*:
fixes $b\ x\ y$ :: *real*
assumes $1 < b\ x > 0$
shows $y \leq \log b\ x \longleftrightarrow b\ powr\ y \leq x$
⟨*proof*⟩

lemma *less-log-iff*:
assumes $1 < b\ x > 0$
shows $y < \log b\ x \longleftrightarrow b\ powr\ y < x$
⟨*proof*⟩

lemma
assumes $1 < b\ x > 0$
shows *log-less-iff*: $\log b\ x < y \longleftrightarrow x < b\ powr\ y$
and *log-le-iff*: $\log b\ x \leq y \longleftrightarrow x \leq b\ powr\ y$
⟨*proof*⟩

lemmas *powr-le-iff* = *le-log-iff* [*symmetric*]
 and *powr-less-iff* = *less-log-iff* [*symmetric*]
 and *less-powr-iff* = *log-less-iff* [*symmetric*]
 and *le-powr-iff* = *log-le-iff* [*symmetric*]

lemma *le-log-of-power*:
 assumes $b \wedge n \leq m$ $1 < b$
 shows $n \leq log\ b\ m$
⟨*proof*⟩

lemma *le-log2-of-power*: $2 \wedge n \leq m \Longrightarrow n \leq log\ 2\ m$ **for** $m\ n :: nat$
⟨*proof*⟩

lemma *log-of-power-le*: $[\![m \leq b \wedge n; b > 1; m > 0]\!] \Longrightarrow log\ b\ (real\ m) \leq n$
⟨*proof*⟩

lemma *log2-of-power-le*: $[\![m \leq 2 \wedge n; m > 0]\!] \Longrightarrow log\ 2\ m \leq n$ **for** $m\ n :: nat$
⟨*proof*⟩

lemma *log-of-power-less*: $[\![m < b \wedge n; b > 1; m > 0]\!] \Longrightarrow log\ b\ (real\ m) < n$
⟨*proof*⟩

lemma *log2-of-power-less*: $[\![m < 2 \wedge n; m > 0]\!] \Longrightarrow log\ 2\ m < n$ **for** $m\ n :: nat$
⟨*proof*⟩

lemma *less-log-of-power*:
 assumes $b \wedge n < m$ $1 < b$
 shows $n < log\ b\ m$
⟨*proof*⟩

lemma *less-log2-of-power*: $2 \wedge n < m \Longrightarrow n < log\ 2\ m$ **for** $m\ n :: nat$
⟨*proof*⟩

lemma *gr-one-powr*[*simp*]:
 fixes $x\ y :: real$ **shows** $[\![x > 1; y > 0]\!] \Longrightarrow 1 < x\ powr\ y$
⟨*proof*⟩

lemma *floor-log-eq-powr-iff*: $x > 0 \Longrightarrow b > 1 \Longrightarrow \lfloor log\ b\ x \rfloor = k \longleftrightarrow b\ powr\ k \leq x \wedge x < b\ powr\ (k + 1)$
 ⟨*proof*⟩

lemma *floor-log-nat-eq-powr-iff*: **fixes** $b\ n\ k :: nat$
 shows $[\![b \geq 2; k > 0]\!] \Longrightarrow$
 $floor\ (log\ b\ (real\ k)) = n \longleftrightarrow b \hat{} n \leq k \wedge k < b \hat{} (n+1)$
⟨*proof*⟩

lemma *floor-log-nat-eq-if*: **fixes** $b\ n\ k :: nat$
 assumes $b \hat{} n \leq k$ $k < b \hat{} (n+1)$ $b \geq 2$

shows *floor* $(log\ b\ (real\ k)) = n$
⟨*proof*⟩

lemma *ceiling-log-eq-powr-iff*: $[\![\ x > 0;\ b > 1\ ]\!]$
$\Longrightarrow \lceil log\ b\ x\rceil = int\ k + 1 \longleftrightarrow b\ powr\ k < x \wedge x \leq b\ powr\ (k + 1)$
⟨*proof*⟩

lemma *ceiling-log-nat-eq-powr-iff*: **fixes** $b\ n\ k :: nat$
shows $[\![\ b \geq 2;\ k > 0\ ]\!] \Longrightarrow$
$ceiling\ (log\ b\ (real\ k)) = int\ n + 1 \longleftrightarrow (b\hat{}n < k \wedge k \leq b\hat{}(n+1))$
⟨*proof*⟩

lemma *ceiling-log-nat-eq-if*: **fixes** $b\ n\ k :: nat$
assumes $b\hat{}n < k\ k \leq b\hat{}(n+1)\ b \geq 2$
shows *ceiling* $(log\ b\ (real\ k)) = int\ n + 1$
⟨*proof*⟩

lemma *floor-log2-div2*: **fixes** $n :: nat$ **assumes** $n \geq 2$
shows *floor*$(log\ 2\ n) =$ *floor*$(log\ 2\ (n\ div\ 2)) + 1$
⟨*proof*⟩

lemma *ceiling-log2-div2*: **assumes** $n \geq 2$
shows *ceiling*$(log\ 2\ (real\ n)) =$ *ceiling*$(log\ 2\ ((n-1)\ div\ 2 + 1)) + 1$
⟨*proof*⟩

lemma *powr-real-of-int*:
$x > 0 \Longrightarrow x\ powr\ real\text{-}of\text{-}int\ n = (if\ n \geq 0\ then\ x\ \hat{}\ nat\ n\ else\ inverse\ (x\ \hat{}\ nat\ (-\ n)))$
⟨*proof*⟩

lemma *powr-numeral* [*simp*]: $0 < x \Longrightarrow x\ powr\ (numeral\ n :: real) = x\ \hat{}\ (numeral\ n)$
⟨*proof*⟩

lemma *powr-int*:
assumes $x > 0$
shows $x\ powr\ i = (if\ i \geq 0\ then\ x\ \hat{}\ nat\ i\ else\ 1\ /\ x\ \hat{}\ nat\ (-i))$
⟨*proof*⟩

lemma *compute-powr*[*code*]:
fixes $i :: real$
shows $b\ powr\ i =$
$(if\ b \leq 0\ then\ Code.abort\ (STR\ ''op\ powr\ with\ nonpositive\ base'')\ (\lambda\text{-}.\ b\ powr\ i)$
$else\ if\ \lfloor i\rfloor = i\ then\ (if\ 0 \leq i\ then\ b\ \hat{}\ nat\ \lfloor i\rfloor\ else\ 1\ /\ b\ \hat{}\ nat\ \lfloor -\ i\rfloor)$
$else\ Code.abort\ (STR\ ''op\ powr\ with\ non-integer\ exponent'')\ (\lambda\text{-}.\ b\ powr\ i))$
⟨*proof*⟩

lemma *powr-one*: $0 \leq x \Longrightarrow x\ powr\ 1 = x$

for x :: *real*
⟨*proof*⟩

lemma *powr-neg-one*: $0 < x \Longrightarrow x$ *powr* $- 1 = 1 / x$
for x :: *real*
⟨*proof*⟩

lemma *powr-neg-numeral*: $0 < x \Longrightarrow x$ *powr* $-$ *numeral* $n = 1 / x$ ˆ *numeral* n
for x :: *real*
⟨*proof*⟩

lemma *root-powr-inverse*: $0 < n \Longrightarrow 0 < x \Longrightarrow$ *root* n $x = x$ *powr* $(1/n)$
⟨*proof*⟩

lemma *ln-powr*: $x \neq 0 \Longrightarrow$ *ln* (x *powr* y) $= y * $ *ln* x
for x :: *real*
⟨*proof*⟩

lemma *ln-root*: $n > 0 \Longrightarrow b > 0 \Longrightarrow$ *ln* (*root* n b) $=$ *ln* b / n
⟨*proof*⟩

lemma *ln-sqrt*: $0 < x \Longrightarrow$ *ln* (*sqrt* x) $=$ *ln* x / 2
⟨*proof*⟩

lemma *log-root*: $n > 0 \Longrightarrow a > 0 \Longrightarrow$ *log* b (*root* n a) $=$ *log* b a / n
⟨*proof*⟩

lemma *log-powr*: $x \neq 0 \Longrightarrow$ *log* b (x *powr* y) $= y *$ *log* b x
⟨*proof*⟩

lemma *log-nat-power*: $0 < x \Longrightarrow$ *log* b (xˆn) $=$ *real* $n *$ *log* b x
⟨*proof*⟩

lemma *log-of-power-eq*:
assumes $m = b$ ˆ n $b > 1$
shows $n =$ *log* b (*real* m)
⟨*proof*⟩

lemma *log2-of-power-eq*: $m = 2$ ˆ $n \Longrightarrow n =$ *log* 2 m **for** m n :: *nat*
⟨*proof*⟩

lemma *log-base-change*: $0 < a \Longrightarrow a \neq 1 \Longrightarrow$ *log* b $x =$ *log* a x / *log* a b
⟨*proof*⟩

lemma *log-base-pow*: $0 < a \Longrightarrow$ *log* (a ˆ n) $x =$ *log* a x / n
⟨*proof*⟩

lemma *log-base-powr*: $a \neq 0 \Longrightarrow$ *log* (a *powr* b) $x =$ *log* a x / b

⟨*proof*⟩

lemma *log-base-root*: $n > 0 \Longrightarrow b > 0 \Longrightarrow log\ (root\ n\ b)\ x = n * (log\ b\ x)$
⟨*proof*⟩

lemma *ln-bound*: $1 \leq x \Longrightarrow ln\ x \leq x$
for x :: *real*
⟨*proof*⟩

lemma *powr-mono*: $a \leq b \Longrightarrow 1 \leq x \Longrightarrow x\ powr\ a \leq x\ powr\ b$
for x :: *real*
⟨*proof*⟩

lemma *ge-one-powr-ge-zero*: $1 \leq x \Longrightarrow 0 \leq a \Longrightarrow 1 \leq x\ powr\ a$
for x :: *real*
⟨*proof*⟩

lemma *powr-less-mono2*: $0 < a \Longrightarrow 0 \leq x \Longrightarrow x < y \Longrightarrow x\ powr\ a < y\ powr\ a$
for x :: *real*
⟨*proof*⟩

lemma *powr-less-mono2-neg*: $a < 0 \Longrightarrow 0 < x \Longrightarrow x < y \Longrightarrow y\ powr\ a < x\ powr\ a$
for x :: *real*
⟨*proof*⟩

lemma *powr-mono2*: $x\ powr\ a \leq y\ powr\ a$ **if** $0 \leq a$ $0 \leq x$ $x \leq y$
for x :: *real*
⟨*proof*⟩

lemma *powr-le1*: $0 \leq a \Longrightarrow 0 \leq x \Longrightarrow x \leq 1 \Longrightarrow x\ powr\ a \leq 1$
for x :: *real*
⟨*proof*⟩

lemma *powr-mono2'*:
fixes $a\ x\ y$ :: *real*
assumes $a \leq 0$ $x > 0$ $x \leq y$
shows $x\ powr\ a \geq y\ powr\ a$
⟨*proof*⟩

lemma *powr-mono-both*:
fixes x :: *real*
assumes $0 \leq a$ $a \leq b$ $1 \leq x$ $x \leq y$
shows $x\ powr\ a \leq y\ powr\ b$
⟨*proof*⟩

lemma *powr-inj*: $0 < a \Longrightarrow a \neq 1 \Longrightarrow a\ powr\ x = a\ powr\ y \longleftrightarrow x = y$
for x :: *real*
⟨*proof*⟩

lemma *powr-half-sqrt*: $0 \leq x \Longrightarrow x\ powr\ (1/2) = sqrt\ x$
⟨*proof*⟩

lemma *ln-powr-bound*: $1 \leq x \Longrightarrow 0 < a \Longrightarrow ln\ x \leq (x\ powr\ a)\ /\ a$
for $x :: real$
⟨*proof*⟩

lemma *ln-powr-bound2*:
fixes $x :: real$
assumes $1 < x$ **and** $0 < a$
shows $(ln\ x)\ powr\ a \leq (a\ powr\ a) * x$
⟨*proof*⟩

lemma *tendsto-powr*:
fixes $a\ b :: real$
assumes f: $(f \longrightarrow a)\ F$
and g: $(g \longrightarrow b)\ F$
and a: $a \neq 0$
shows $((\lambda x.\ f\ x\ powr\ g\ x) \longrightarrow a\ powr\ b)\ F$
⟨*proof*⟩

lemma *tendsto-powr'*[*tendsto-intros*]:
fixes $a :: real$
assumes f: $(f \longrightarrow a)\ F$
and g: $(g \longrightarrow b)\ F$
and a: $a \neq 0 \vee (b > 0 \wedge eventually\ (\lambda x.\ f\ x \geq 0)\ F)$
shows $((\lambda x.\ f\ x\ powr\ g\ x) \longrightarrow a\ powr\ b)\ F$
⟨*proof*⟩

lemma *continuous-powr*:
assumes *continuous F f*
and *continuous F g*
and $f\ (Lim\ F\ (\lambda x.\ x)) \neq 0$
shows $continuous\ F\ (\lambda x.\ (f\ x)\ powr\ (g\ x :: real))$
⟨*proof*⟩

lemma *continuous-at-within-powr*[*continuous-intros*]:
fixes $f\ g :: - \Rightarrow real$
assumes *continuous* (*at a within s*) *f*
and *continuous* (*at a within s*) *g*
and $f\ a \neq 0$
shows $continuous\ (at\ a\ within\ s)\ (\lambda x.\ (f\ x)\ powr\ (g\ x))$
⟨*proof*⟩

lemma *isCont-powr*[*continuous-intros*, *simp*]:
fixes $f\ g :: - \Rightarrow real$
assumes $isCont\ f\ a\ isCont\ g\ a\ f\ a \neq 0$
shows $isCont\ (\lambda x.\ (f\ x)\ powr\ g\ x)\ a$

⟨*proof*⟩

lemma *continuous-on-powr*[*continuous-intros*]:
fixes *f g* :: - ⇒ *real*
assumes *continuous-on s f continuous-on s g* **and** $\forall x \in s.\ f\ x \neq 0$
shows *continuous-on s* (λ*x*. (*f x*) *powr* (*g x*))
⟨*proof*⟩

lemma *tendsto-powr2*:
fixes *a* :: *real*
assumes *f*: (*f* ⟶ *a*) *F*
and *g*: (*g* ⟶ *b*) *F*
and $\forall_F\ x$ *in F*. $0 \leq f\ x$
and *b*: $0 < b$
shows ((λ*x*. *f x powr g x*) ⟶ *a powr b*) *F*
⟨*proof*⟩

lemma *DERIV-powr*:
fixes *r* :: *real*
assumes *g*: *DERIV g x* :> *m*
and *pos*: $g\ x > 0$
and *f*: *DERIV f x* :> *r*
shows *DERIV* (λ*x*. *g x powr f x*) *x* :> (*g x powr f x*) ∗ (*r* ∗ *ln* (*g x*) + *m* ∗ *f x* / *g x*)
⟨*proof*⟩

lemma *DERIV-fun-powr*:
fixes *r* :: *real*
assumes *g*: *DERIV g x* :> *m*
and *pos*: $g\ x > 0$
shows *DERIV* (λ*x*. (*g x*) *powr r*) *x* :> *r* ∗ (*g x*) *powr* (*r* − *of-nat 1*) ∗ *m*
⟨*proof*⟩

lemma *has-real-derivative-powr*:
assumes $z > 0$
shows ((λ*z*. *z powr r*) *has-real-derivative r* ∗ *z powr* (*r* − *1*)) (*at z*)
⟨*proof*⟩

declare *has-real-derivative-powr*[*THEN DERIV-chain2*, *derivative-intros*]

lemma *tendsto-zero-powrI*:
assumes (*f* ⟶ (*0*::*real*)) *F* (*g* ⟶ *b*) *F* $\forall_F\ x$ *in F*. $0 \leq f\ x$ $0 < b$
shows ((λ*x*. *f x powr g x*) ⟶ *0*) *F*
⟨*proof*⟩

lemma *continuous-on-powr'*:
fixes *f g* :: - ⇒ *real*
assumes *continuous-on s f continuous-on s g*
and $\forall x \in s.\ f\ x \geq 0 \land (f\ x = 0 \longrightarrow g\ x > 0)$

shows *continuous-on s* ($\lambda x.$ (*f x*) *powr* (*g x*))
⟨*proof*⟩

lemma *tendsto-neg-powr*:
assumes $s < 0$
and *f*: *LIM x F. f x* :> *at-top*
shows (($\lambda x.$ *f x powr s*) ⟶ (*0*::*real*)) *F*
⟨*proof*⟩

lemma *tendsto-exp-limit-at-right*: (($\lambda y.$ (*1* + *x* ∗ *y*) *powr* (*1* / *y*)) ⟶ *exp x*) (*at-right 0*)
for *x* :: *real*
⟨*proof*⟩

lemma *tendsto-exp-limit-at-top*: (($\lambda y.$ (*1* + *x* / *y*) *powr y*) ⟶ *exp x*) *at-top*
for *x* :: *real*
⟨*proof*⟩

lemma *tendsto-exp-limit-sequentially*: ($\lambda n.$ (*1* + *x* / *n*) ^ *n*) ⟶ *exp x*
for *x* :: *real*
⟨*proof*⟩

105.10 Sine and Cosine

definition *sin-coeff* :: *nat* ⇒ *real*
where *sin-coeff* = ($\lambda n.$ *if even n then 0 else* (− *1*) ^ ((*n* − *Suc 0*) *div 2*) / (*fact n*))

definition *cos-coeff* :: *nat* ⇒ *real*
where *cos-coeff* = ($\lambda n.$ *if even n then* ((− *1*) ^ (*n div 2*)) / (*fact n*) *else 0*)

definition *sin* :: *'a* ⇒ *'a*::{*real-normed-algebra-1*,*banach*}
where *sin* = ($\lambda x.$ $\sum$ *n. sin-coeff n* $*_R$ *x*^*n*)

definition *cos* :: *'a* ⇒ *'a*::{*real-normed-algebra-1*,*banach*}
where *cos* = ($\lambda x.$ $\sum$ *n. cos-coeff n* $*_R$ *x*^*n*)

lemma *sin-coeff-0* [*simp*]: *sin-coeff 0* = *0*
⟨*proof*⟩

lemma *cos-coeff-0* [*simp*]: *cos-coeff 0* = *1*
⟨*proof*⟩

lemma *sin-coeff-Suc*: *sin-coeff* (*Suc n*) = *cos-coeff n* / *real* (*Suc n*)
⟨*proof*⟩

lemma *cos-coeff-Suc*: *cos-coeff* (*Suc n*) = − *sin-coeff n* / *real* (*Suc n*)
⟨*proof*⟩

lemma *summable-norm-sin*: *summable* $(\lambda n.\ norm\ (sin\text{-}coeff\ n *_R x\hat{\ }n))$
for x :: $'a$::{*real-normed-algebra-1*,*banach*}
⟨*proof*⟩

lemma *summable-norm-cos*: *summable* $(\lambda n.\ norm\ (cos\text{-}coeff\ n *_R x\hat{\ }n))$
for x :: $'a$::{*real-normed-algebra-1*,*banach*}
⟨*proof*⟩

lemma *sin-converges*: $(\lambda n.\ sin\text{-}coeff\ n *_R x\hat{\ }n)$ *sums sin* x
⟨*proof*⟩

lemma *cos-converges*: $(\lambda n.\ cos\text{-}coeff\ n *_R x\hat{\ }n)$ *sums cos* x
⟨*proof*⟩

lemma *sin-of-real*: *sin* (*of-real* x) = *of-real* (*sin* x)
for x :: *real*
⟨*proof*⟩

corollary *sin-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow sin\ z \in \mathbb{R}$
⟨*proof*⟩

lemma *cos-of-real*: *cos* (*of-real* x) = *of-real* (*cos* x)
for x :: *real*
⟨*proof*⟩

corollary *cos-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow cos\ z \in \mathbb{R}$
⟨*proof*⟩

lemma *diffs-sin-coeff*: *diffs sin-coeff* = *cos-coeff*
⟨*proof*⟩

lemma *diffs-cos-coeff*: *diffs cos-coeff* = $(\lambda n.\ -\ sin\text{-}coeff\ n)$
⟨*proof*⟩

lemma *sin-int-times-real*: *sin* (*of-int* m $*$ *of-real* x) = *of-real* (*sin* (*of-int* m $*$ x))
⟨*proof*⟩

lemma *cos-int-times-real*: *cos* (*of-int* m $*$ *of-real* x) = *of-real* (*cos* (*of-int* m $*$ x))
⟨*proof*⟩

Now at last we can get the derivatives of exp, sin and cos.

lemma *DERIV-sin* [*simp*]: *DERIV sin x :> cos x*
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

declare *DERIV-sin*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-sin*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemma *DERIV-cos* [*simp*]: *DERIV cos x :>* $-$ *sin x*

for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

declare *DERIV-cos*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-cos*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemma *isCont-sin*: *isCont sin x*
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-cos*: *isCont cos x*
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-sin'* [*simp*]: *isCont f a* $\Longrightarrow$ *isCont* (λx. *sin* (*f x*)) *a*
for f :: - $\Rightarrow$ $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-cos'* [*simp*]: *isCont f a* $\Longrightarrow$ *isCont* (λx. *cos* (*f x*)) *a*
for f :: - $\Rightarrow$ $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *tendsto-sin* [*tendsto-intros*]: ($f \longrightarrow a$) $F \Longrightarrow$ ((λx. *sin* (*f x*)) $\longrightarrow$ *sin a*) F
for f :: - $\Rightarrow$ $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *tendsto-cos* [*tendsto-intros*]: ($f \longrightarrow a$) $F \Longrightarrow$ ((λx. *cos* (*f x*)) $\longrightarrow$ *cos a*) F
for f :: - $\Rightarrow$ $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-sin* [*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* (λx. *sin* (*f x*))
for f :: - $\Rightarrow$ $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-on-sin* [*continuous-intros*]: *continuous-on s f* $\Longrightarrow$ *continuous-on s* (λx. *sin* (*f x*))
for f :: - $\Rightarrow$ $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-within-sin*: *continuous* (*at z within s*) *sin*
for z :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-cos* [*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* (λx.

cos (*f x*))
  for *f* :: - ⇒ *'a*::{*real-normed-field*,*banach*}
  ⟨*proof*⟩

lemma *continuous-on-cos* [*continuous-intros*]: *continuous-on s f* ⟹ *continuous-on s* ($\lambda x.$ *cos* (*f x*))
  for *f* :: - ⇒ *'a*::{*real-normed-field*,*banach*}
  ⟨*proof*⟩

lemma *continuous-within-cos*: *continuous* (*at z within s*) *cos*
  for *z* :: *'a*::{*real-normed-field*,*banach*}
  ⟨*proof*⟩

105.11 Properties of Sine and Cosine

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

lemma *cos-zero* [*simp*]: *cos 0* = *1*
  ⟨*proof*⟩

lemma *DERIV-fun-sin*: *DERIV g x* :> *m* ⟹ *DERIV* ($\lambda x.$ *sin* (*g x*)) *x* :> *cos* (*g x*) ∗ *m*
  ⟨*proof*⟩

lemma *DERIV-fun-cos*: *DERIV g x* :> *m* ⟹ *DERIV* ($\lambda x.$ *cos*(*g x*)) *x* :> − *sin* (*g x*) ∗ *m*
  ⟨*proof*⟩

105.12 Deriving the Addition Formulas

The product of two cosine series.

lemma *cos-x-cos-y*:
  fixes *x* :: *'a*::{*real-normed-field*,*banach*}
  shows
    ($\lambda p.$ $\sum$ *n*≤*p*.
        if even p ∧ *even n*
        then ((−*1*) ^ (*p div 2*) ∗ (*p choose n*) / (*fact p*)) $*_R$ (*x*^*n*) ∗ *y*^(*p*−*n*) *else 0*)
      sums (*cos x* ∗ *cos y*)
⟨*proof*⟩

The product of two sine series.

lemma *sin-x-sin-y*:
  fixes *x* :: *'a*::{*real-normed-field*,*banach*}
  shows
    ($\lambda p.$ $\sum$ *n*≤*p*.
        if even p ∧ *odd n*
        then − ((−*1*) ^ (*p div 2*) ∗ (*p choose n*) / (*fact p*)) $*_R$ (*x*^*n*) ∗ *y*^(*p*−*n*)

else 0)
sums (*sin x* $*$ *sin y*)
⟨*proof*⟩

lemma *sums-cos-x-plus-y*:
fixes x :: *'a*::{*real-normed-field*,*banach*}
shows
($\lambda p.\ \sum n \le p.$
if even p
then ((−*1*) ˆ (*p div 2*) $*$ (*p choose n*) / (*fact p*)) $*_R$ (*x*ˆ*n*) $*$ *y*ˆ(*p*−*n*)
else 0)
sums cos (*x* + *y*)
⟨*proof*⟩

theorem *cos-add*:
fixes x :: *'a*::{*real-normed-field*,*banach*}
shows *cos* (*x* + *y*) = *cos x* $*$ *cos y* − *sin x* $*$ *sin y*
⟨*proof*⟩

lemma *sin-minus-converges*: ($\lambda n.$ − (*sin-coeff n* $*_R$ (−*x*) ˆ*n*)) *sums sin x*
⟨*proof*⟩

lemma *sin-minus* [*simp*]: *sin* (− *x*) = − *sin x*
for x :: *'a*::{*real-normed-algebra-1*,*banach*}
⟨*proof*⟩

lemma *cos-minus-converges*: ($\lambda n.$ (*cos-coeff n* $*_R$ (−*x*) ˆ*n*)) *sums cos x*
⟨*proof*⟩

lemma *cos-minus* [*simp*]: *cos* (−*x*) = *cos x*
for x :: *'a*::{*real-normed-algebra-1*,*banach*}
⟨*proof*⟩

lemma *sin-cos-squared-add* [*simp*]: $(sin\ x)^2 + (cos\ x)^2 = 1$
for x :: *'a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-cos-squared-add2* [*simp*]: $(cos\ x)^2 + (sin\ x)^2 = 1$
for x :: *'a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-cos-squared-add3* [*simp*]: *cos x* $*$ *cos x* + *sin x* $*$ *sin x* = *1*
for x :: *'a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-squared-eq*: $(sin\ x)^2 = 1 - (cos\ x)^2$
for x :: *'a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *cos-squared-eq*: $(cos\ x)^2 = 1 - (sin\ x)^2$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *abs-sin-le-one* [*simp*]: $|sin\ x| \le 1$
for x :: *real*
⟨*proof*⟩

lemma *sin-ge-minus-one* [*simp*]: $-\ 1 \le sin\ x$
for x :: *real*
⟨*proof*⟩

lemma *sin-le-one* [*simp*]: $sin\ x \le 1$
for x :: *real*
⟨*proof*⟩

lemma *abs-cos-le-one* [*simp*]: $|cos\ x| \le 1$
for x :: *real*
⟨*proof*⟩

lemma *cos-ge-minus-one* [*simp*]: $-\ 1 \le cos\ x$
for x :: *real*
⟨*proof*⟩

lemma *cos-le-one* [*simp*]: $cos\ x \le 1$
for x :: *real*
⟨*proof*⟩

lemma *cos-diff*: $cos\ (x - y) = cos\ x * cos\ y + sin\ x * sin\ y$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *cos-double*: $cos(2{*}x) = (cos\ x)^2 - (sin\ x)^2$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-cos-le1*: $|sin\ x * sin\ y + cos\ x * cos\ y| \le 1$
for x :: *real*
⟨*proof*⟩

lemma *DERIV-fun-pow*: $DERIV\ g\ x :> m \Longrightarrow DERIV\ (\lambda x.\ (g\ x)\ \hat{}\ n)\ x :> real\ n * (g\ x)\ \hat{}\ (n - 1) * m$
⟨*proof*⟩

lemma *DERIV-fun-exp*: $DERIV\ g\ x :> m \Longrightarrow DERIV\ (\lambda x.\ exp\ (g\ x))\ x :> exp\ (g\ x) * m$
⟨*proof*⟩

105.13 The Constant Pi

definition *pi* :: *real*
  where $pi = 2 * (THE\ x.\ 0 \leq x \wedge x \leq 2 \wedge cos\ x = 0)$

Show that there's a least positive x with $cos\ x = (0::'a)$; hence define pi.

lemma *sin-paired*: $(\lambda n.\ (-\ 1)\ \hat{}\ n\ /\ (fact\ (2 * n + 1)) * x\ \hat{}\ (2 * n + 1))$ *sums* *sin x*
  for *x* :: *real*
⟨*proof*⟩

lemma *sin-gt-zero-02*:
  fixes *x* :: *real*
  assumes $0 < x$ **and** $x < 2$
  shows $0 < sin\ x$
⟨*proof*⟩

lemma *cos-double-less-one*: $0 < x \Longrightarrow x < 2 \Longrightarrow cos\ (2 * x) < 1$
  for *x* :: *real*
  ⟨*proof*⟩

lemma *cos-paired*: $(\lambda n.\ (-\ 1)\ \hat{}\ n\ /\ (fact\ (2 * n)) * x\ \hat{}\ (2 * n))$ *sums cos x*
  for *x* :: *real*
⟨*proof*⟩

lemmas *realpow-num-eq-if* = *power-eq-if*

lemma *sumr-pos-lt-pair*:
  fixes *f* :: *nat* ⇒ *real*
  shows *summable f* ⟹
    $(\bigwedge d.\ 0 < f\ (k + (Suc(Suc\ 0) * d)) + f\ (k + ((Suc\ (Suc\ 0) * d) + 1))) \Longrightarrow$
    $sum\ f\ \{..<k\} < suminf\ f$
  ⟨*proof*⟩

lemma *cos-two-less-zero* [*simp*]: $cos\ 2 < (0::real)$
⟨*proof*⟩

lemmas *cos-two-neq-zero* [*simp*] = *cos-two-less-zero* [*THEN less-imp-neq*]
lemmas *cos-two-le-zero* [*simp*] = *cos-two-less-zero* [*THEN order-less-imp-le*]

lemma *cos-is-zero*: $\exists!x::real.\ 0 \leq x \wedge x \leq 2 \wedge cos\ x = 0$
⟨*proof*⟩

lemma *pi-half*: $pi/2 = (THE\ x.\ 0 \leq x \wedge x \leq 2 \wedge cos\ x = 0)$
  ⟨*proof*⟩

lemma *cos-pi-half* [*simp*]: $cos\ (pi\ /\ 2) = 0$
  ⟨*proof*⟩

lemma *cos-of-real-pi-half* [*simp*]: $cos\ ((of\text{-}real\ pi\ /\ 2) :: 'a) = 0$

if *SORT-CONSTRAINT*('*a*::{*real-field*,*banach*,*real-normed-algebra-1*})
⟨*proof*⟩

lemma *pi-half-gt-zero* [*simp*]: $0 < pi / 2$
⟨*proof*⟩

lemmas *pi-half-neq-zero* [*simp*] = *pi-half-gt-zero* [*THEN less-imp-neq*, *symmetric*]
lemmas *pi-half-ge-zero* [*simp*] = *pi-half-gt-zero* [*THEN order-less-imp-le*]

lemma *pi-half-less-two* [*simp*]: $pi / 2 < 2$
⟨*proof*⟩

lemmas *pi-half-neq-two* [*simp*] = *pi-half-less-two* [*THEN less-imp-neq*]
lemmas *pi-half-le-two* [*simp*] = *pi-half-less-two* [*THEN order-less-imp-le*]

lemma *pi-gt-zero* [*simp*]: $0 < pi$
⟨*proof*⟩

lemma *pi-ge-zero* [*simp*]: $0 \leq pi$
⟨*proof*⟩

lemma *pi-neq-zero* [*simp*]: $pi \neq 0$
⟨*proof*⟩

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

lemma *minus-pi-half-less-zero*: $-(pi/2) < 0$
⟨*proof*⟩

lemma *m2pi-less-pi*: $-\ (2*pi) < pi$
⟨*proof*⟩

lemma *sin-pi-half* [*simp*]: $sin(pi/2) = 1$
⟨*proof*⟩

lemma *sin-of-real-pi-half* [*simp*]: *sin* ((*of-real pi* / *2*) :: '*a*) = *1*
if *SORT-CONSTRAINT*('*a*::{*real-field*,*banach*,*real-normed-algebra-1*})
⟨*proof*⟩

lemma *sin-cos-eq*: *sin x* = *cos* (*of-real pi* / *2* − *x*)
for *x* :: '*a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *minus-sin-cos-eq*: − *sin x* = *cos* (*x* + *of-real pi* / *2*)
for *x* :: '*a*::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *cos-sin-eq*: *cos x* = *sin* (*of-real pi* / *2* − *x*)

for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-add*: *sin* $(x + y) =$ *sin* $x *$ *cos* $y +$ *cos* $x *$ *sin* y
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-diff*: *sin* $(x - y) =$ *sin* $x *$ *cos* $y -$ *cos* $x *$ *sin* y
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-double*: *sin*$(2 * x) = 2 *$ *sin* $x *$ *cos* x
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *cos-of-real-pi* [*simp*]: *cos* (*of-real pi*) $= -1$
⟨*proof*⟩

lemma *sin-of-real-pi* [*simp*]: *sin* (*of-real pi*) $= 0$
⟨*proof*⟩

lemma *cos-pi* [*simp*]: *cos pi* $= -1$
⟨*proof*⟩

lemma *sin-pi* [*simp*]: *sin pi* $= 0$
⟨*proof*⟩

lemma *sin-periodic-pi* [*simp*]: *sin* $(x +$ *pi*$) = -$ *sin* x
⟨*proof*⟩

lemma *sin-periodic-pi2* [*simp*]: *sin* (*pi* $+ x) = -$ *sin* x
⟨*proof*⟩

lemma *cos-periodic-pi* [*simp*]: *cos* $(x +$ *pi*$) = -$ *cos* x
⟨*proof*⟩

lemma *cos-periodic-pi2* [*simp*]: *cos* (*pi* $+ x) = -$ *cos* x
⟨*proof*⟩

lemma *sin-periodic* [*simp*]: *sin* $(x + 2 *$ *pi*$) =$ *sin* x
⟨*proof*⟩

lemma *cos-periodic* [*simp*]: *cos* $(x + 2 *$ *pi*$) =$ *cos* x
⟨*proof*⟩

lemma *cos-npi* [*simp*]: *cos* (*real* $n *$ *pi*) $= (- 1)$ ^ n
⟨*proof*⟩

lemma *cos-npi2* [*simp*]: *cos* (*pi* $*$ *real* $n) = (- 1)$ ^ n

⟨proof⟩

lemma *sin-npi* [*simp*]: *sin* (*real n* ∗ *pi*) = *0*
for *n* :: *nat*
⟨proof⟩

lemma *sin-npi2* [*simp*]: *sin* (*pi* ∗ *real n*) = *0*
for *n* :: *nat*
⟨proof⟩

lemma *cos-two-pi* [*simp*]: *cos* (*2* ∗ *pi*) = *1*
⟨proof⟩

lemma *sin-two-pi* [*simp*]: *sin* (*2* ∗ *pi*) = *0*
⟨proof⟩

lemma *sin-times-sin*: *sin w* ∗ *sin z* = (*cos* (*w* − *z*) − *cos* (*w* + *z*)) / *2*
for *w* :: *'a*::{*real-normed-field*,*banach*}
⟨proof⟩

lemma *sin-times-cos*: *sin w* ∗ *cos z* = (*sin* (*w* + *z*) + *sin* (*w* − *z*)) / *2*
for *w* :: *'a*::{*real-normed-field*,*banach*}
⟨proof⟩

lemma *cos-times-sin*: *cos w* ∗ *sin z* = (*sin* (*w* + *z*) − *sin* (*w* − *z*)) / *2*
for *w* :: *'a*::{*real-normed-field*,*banach*}
⟨proof⟩

lemma *cos-times-cos*: *cos w* ∗ *cos z* = (*cos* (*w* − *z*) + *cos* (*w* + *z*)) / *2*
for *w* :: *'a*::{*real-normed-field*,*banach*}
⟨proof⟩

lemma *sin-plus-sin*: *sin w* + *sin z* = *2* ∗ *sin* ((*w* + *z*) / *2*) ∗ *cos* ((*w* − *z*) / *2*)
for *w* :: *'a*::{*real-normed-field*,*banach*,*field*}
⟨proof⟩

lemma *sin-diff-sin*: *sin w* − *sin z* = *2* ∗ *sin* ((*w* − *z*) / *2*) ∗ *cos* ((*w* + *z*) / *2*)
for *w* :: *'a*::{*real-normed-field*,*banach*,*field*}
⟨proof⟩

lemma *cos-plus-cos*: *cos w* + *cos z* = *2* ∗ *cos* ((*w* + *z*) / *2*) ∗ *cos* ((*w* − *z*) / *2*)
for *w* :: *'a*::{*real-normed-field*,*banach*,*field*}
⟨proof⟩

lemma *cos-diff-cos*: *cos w* − *cos z* = *2* ∗ *sin* ((*w* + *z*) / *2*) ∗ *sin* ((*z* − *w*) / *2*)
for *w* :: *'a*::{*real-normed-field*,*banach*,*field*}
⟨proof⟩

lemma *cos-double-cos*: *cos* (*2* ∗ *z*) = *2* ∗ *cos z* ^ *2* − *1*

for z :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *cos-double-sin*: $cos\ (2 * z) = 1 - 2 * sin\ z\ \hat{}\ 2$
for z :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *sin-pi-minus* [*simp*]: $sin\ (pi - x) = sin\ x$
⟨*proof*⟩

lemma *cos-pi-minus* [*simp*]: $cos\ (pi - x) = -\ (cos\ x)$
⟨*proof*⟩

lemma *sin-minus-pi* [*simp*]: $sin\ (x - pi) = -\ (sin\ x)$
⟨*proof*⟩

lemma *cos-minus-pi* [*simp*]: $cos\ (x - pi) = -\ (cos\ x)$
⟨*proof*⟩

lemma *sin-2pi-minus* [*simp*]: $sin\ (2 * pi - x) = -\ (sin\ x)$
⟨*proof*⟩

lemma *cos-2pi-minus* [*simp*]: $cos\ (2 * pi - x) = cos\ x$
⟨*proof*⟩

lemma *sin-gt-zero2*: $0 < x \Longrightarrow x < pi/2 \Longrightarrow 0 < sin\ x$
⟨*proof*⟩

lemma *sin-less-zero*:
assumes $-\ pi/2 < x$ **and** $x < 0$
shows $sin\ x < 0$
⟨*proof*⟩

lemma *pi-less-4*: $pi < 4$
⟨*proof*⟩

lemma *cos-gt-zero*: $0 < x \Longrightarrow x < pi/2 \Longrightarrow 0 < cos\ x$
⟨*proof*⟩

lemma *cos-gt-zero-pi*: $-(pi/2) < x \Longrightarrow x < pi/2 \Longrightarrow 0 < cos\ x$
⟨*proof*⟩

lemma *cos-ge-zero*: $-(pi/2) \leq x \Longrightarrow x \leq pi/2 \Longrightarrow 0 \leq cos\ x$
⟨*proof*⟩

lemma *sin-gt-zero*: $0 < x \Longrightarrow x < pi \Longrightarrow 0 < sin\ x$
⟨*proof*⟩

lemma *sin-lt-zero*: $pi < x \Longrightarrow x < 2 * pi \Longrightarrow sin\ x < 0$

⟨*proof*⟩

lemma *pi-ge-two*: $2 \leq pi$
⟨*proof*⟩

lemma *sin-ge-zero*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow 0 \leq sin\ x$
⟨*proof*⟩

lemma *sin-le-zero*: $pi \leq x \Longrightarrow x < 2 * pi \Longrightarrow sin\ x \leq 0$
⟨*proof*⟩

lemma *sin-pi-divide-n-ge-0* [*simp*]:
assumes $n \neq 0$
shows $0 \leq sin\ (pi\ /\ real\ n)$
⟨*proof*⟩

lemma *sin-pi-divide-n-gt-0*:
assumes $2 \leq n$
shows $0 < sin\ (pi\ /\ real\ n)$
⟨*proof*⟩

lemma *cos-total*:
assumes y: $-\ 1 \leq y\ y \leq 1$
shows $\exists!x.\ 0 \leq x \wedge x \leq pi \wedge cos\ x = y$
⟨*proof*⟩

lemma *sin-total*:
assumes y: $-1 \leq y\ y \leq 1$
shows $\exists!x.\ -\ (pi/2) \leq x \wedge x \leq pi/2 \wedge sin\ x = y$
⟨*proof*⟩

lemma *cos-zero-lemma*:
assumes $0 \leq x\ cos\ x = 0$
shows $\exists\, n.\ odd\ n \wedge x = of\text{-}nat\ n * (pi/2) \wedge n > 0$
⟨*proof*⟩

lemma *sin-zero-lemma*: $0 \leq x \Longrightarrow sin\ x = 0 \Longrightarrow \exists\, n{::}nat.\ even\ n \wedge x = real\ n * (pi/2)$
⟨*proof*⟩

lemma *cos-zero-iff*:
$cos\ x = 0 \longleftrightarrow ((\exists\, n.\ odd\ n \wedge x = real\ n * (pi/2)) \vee (\exists\, n.\ odd\ n \wedge x = -\ (real\ n * (pi/2))))$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *sin-zero-iff*:
$sin\ x = 0 \longleftrightarrow ((\exists\, n.\ even\ n \wedge x = real\ n * (pi/2)) \vee (\exists\, n.\ even\ n \wedge x = -$

$(real\ n * (pi/2))))$
  (**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *cos-zero-iff-int*: $cos\ x = 0 \longleftrightarrow (\exists\, n.\ odd\ n \wedge x = of\text{-}int\ n * (pi/2))$
⟨*proof*⟩

lemma *sin-zero-iff-int*: $sin\ x = 0 \longleftrightarrow (\exists\, n.\ even\ n \wedge x = of\text{-}int\ n * (pi/2))$
⟨*proof*⟩

lemma *sin-zero-iff-int2*: $sin\ x = 0 \longleftrightarrow (\exists\, n{::}int.\ x = of\text{-}int\ n * pi)$
  ⟨*proof*⟩

lemma *sin-npi-int* [*simp*]: $sin\ (pi * of\text{-}int\ n) = 0$
  ⟨*proof*⟩

lemma *cos-monotone-0-pi*:
  assumes $0 \le y$ **and** $y < x$ **and** $x \le pi$
  shows $cos\ x < cos\ y$
⟨*proof*⟩

lemma *cos-monotone-0-pi-le*:
  assumes $0 \le y$ **and** $y \le x$ **and** $x \le pi$
  shows $cos\ x \le cos\ y$
⟨*proof*⟩

lemma *cos-monotone-minus-pi-0*:
  assumes $-\ pi \le y$ **and** $y < x$ **and** $x \le 0$
  shows $cos\ y < cos\ x$
⟨*proof*⟩

lemma *cos-monotone-minus-pi-0'*:
  assumes $-\ pi \le y$ **and** $y \le x$ **and** $x \le 0$
  shows $cos\ y \le cos\ x$
⟨*proof*⟩

lemma *sin-monotone-2pi*:
  assumes $-\ (pi/2) \le y$ **and** $y < x$ **and** $x \le pi/2$
  shows $sin\ y < sin\ x$
  ⟨*proof*⟩

lemma *sin-monotone-2pi-le*:
  assumes $-\ (pi\ /\ 2) \le y$ **and** $y \le x$ **and** $x \le pi\ /\ 2$
  shows $sin\ y \le sin\ x$
  ⟨*proof*⟩

lemma *sin-x-le-x*:
  fixes $x :: real$
  assumes x: $x \ge 0$

shows *sin x* $\leq$ *x*
$\langle proof \rangle$

lemma *sin-x-ge-neg-x*:
fixes *x* :: *real*
assumes *x*: *x* $\geq$ *0*
shows *sin x* $\geq - x$
$\langle proof \rangle$

lemma *abs-sin-x-le-abs-x*: $|sin\ x| \leq |x|$
for *x* :: *real*
$\langle proof \rangle$

105.14 More Corollaries about Sine and Cosine

lemma *sin-cos-npi* [*simp*]: $sin\ (real\ (Suc\ (2 * n)) * pi\ /\ 2) = (-1)\ \hat{}\ n$
$\langle proof \rangle$

lemma *cos-2npi* [*simp*]: $cos\ (2 * real\ n * pi) = 1$
for *n* :: *nat*
$\langle proof \rangle$

lemma *cos-3over2-pi* [*simp*]: $cos\ (3/2{*}pi) = 0$
$\langle proof \rangle$

lemma *sin-2npi* [*simp*]: $sin\ (2 * real\ n * pi) = 0$
for *n* :: *nat*
$\langle proof \rangle$

lemma *sin-3over2-pi* [*simp*]: $sin\ (3/2{*}pi) = -\ 1$
$\langle proof \rangle$

lemma *cos-pi-eq-zero* [*simp*]: $cos\ (pi * real\ (Suc\ (2 * m))\ /\ 2) = 0$
$\langle proof \rangle$

lemma *DERIV-cos-add* [*simp*]: $DERIV\ (\lambda x.\ cos\ (x + k))\ xa :> -\ sin\ (xa + k)$
$\langle proof \rangle$

lemma *sin-zero-norm-cos-one*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}
assumes *sin x* = *0*
shows *norm* (*cos x*) = *1*
$\langle proof \rangle$

lemma *sin-zero-abs-cos-one*: $sin\ x = 0 \Longrightarrow |cos\ x| = (1{::}real)$
$\langle proof \rangle$

lemma *cos-one-sin-zero*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}

assumes *cos x = 1*
shows *sin x = 0*
⟨*proof*⟩

lemma *sin-times-pi-eq-0*: $sin\ (x * pi) = 0 \longleftrightarrow x \in \mathbb{Z}$
⟨*proof*⟩

lemma *cos-one-2pi*: $cos\ x = 1 \longleftrightarrow (\exists\, n{::}nat.\ x = n * 2 * pi) \mid (\exists\, n{::}nat.\ x = -(n * 2 * pi))$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *cos-one-2pi-int*: $cos\ x = 1 \longleftrightarrow (\exists\, n{::}int.\ x = n * 2 * pi)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *cos-npi-int* [*simp*]:
fixes *n::int* **shows** $cos\ (pi * of\text{-}int\ n) = (if\ even\ n\ then\ 1\ else\ -1)$
⟨*proof*⟩

lemma *sin-cos-sqrt*: $0 \le sin\ x \Longrightarrow sin\ x = sqrt\ (1 - (cos(x)\ \hat{}\ 2))$
⟨*proof*⟩

lemma *sin-eq-0-pi*: $-\ pi < x \Longrightarrow x < pi \Longrightarrow sin\ x = 0 \Longrightarrow x = 0$
⟨*proof*⟩

lemma *cos-treble-cos*: $cos\ (3 * x) = 4 * cos\ x\ \hat{}\ 3 - 3 * cos\ x$
for *x :: 'a::{real-normed-field,banach}*
⟨*proof*⟩

lemma *cos-45*: $cos\ (pi\ /\ 4) = sqrt\ 2\ /\ 2$
⟨*proof*⟩

lemma *cos-30*: $cos\ (pi\ /\ 6) = sqrt\ 3/2$
⟨*proof*⟩

lemma *sin-45*: $sin\ (pi\ /\ 4) = sqrt\ 2\ /\ 2$
⟨*proof*⟩

lemma *sin-60*: $sin\ (pi\ /\ 3) = sqrt\ 3/2$
⟨*proof*⟩

lemma *cos-60*: $cos\ (pi\ /\ 3) = 1\ /\ 2$
⟨*proof*⟩

lemma *sin-30*: $sin\ (pi\ /\ 6) = 1\ /\ 2$
⟨*proof*⟩

lemma *cos-integer-2pi*: $n \in \mathbb{Z} \Longrightarrow cos(2 * pi * n) = 1$
⟨*proof*⟩

lemma *sin-integer-2pi*: $n \in \mathbb{Z} \Longrightarrow sin(2 * pi * n) = 0$
⟨*proof*⟩

lemma *cos-int-2npi* [*simp*]: $cos\ (2 * of\text{-}int\ n * pi) = 1$
for $n :: int$
⟨*proof*⟩

lemma *sin-int-2npi* [*simp*]: $sin\ (2 * of\text{-}int\ n * pi) = 0$
for $n :: int$
⟨*proof*⟩

lemma *sincos-principal-value*: $\exists\, y.\ (-\ pi < y \wedge y \leq pi) \wedge (sin\ y = sin\ x \wedge cos\ y = cos\ x)$
⟨*proof*⟩

105.15 Tangent

definition $tan :: 'a \Rightarrow 'a::\{real\text{-}normed\text{-}field,banach\}$
where $tan = (\lambda x.\ sin\ x\ /\ cos\ x)$

lemma *tan-of-real*: $of\text{-}real\ (tan\ x) = (tan\ (of\text{-}real\ x) :: 'a::\{real\text{-}normed\text{-}field,banach\})$
⟨*proof*⟩

lemma *tan-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow tan\ z \in \mathbb{R}$
for $z :: 'a::\{real\text{-}normed\text{-}field,banach\}$
⟨*proof*⟩

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

lemma *tan-pi* [*simp*]: $tan\ pi = 0$
⟨*proof*⟩

lemma *tan-npi* [*simp*]: $tan\ (real\ n * pi) = 0$
for $n :: nat$
⟨*proof*⟩

lemma *tan-minus* [*simp*]: $tan\ (-\ x) = -\ tan\ x$
⟨*proof*⟩

lemma *tan-periodic* [*simp*]: $tan\ (x + 2 * pi) = tan\ x$
⟨*proof*⟩

lemma *lemma-tan-add1*: $cos\ x \neq 0 \Longrightarrow cos\ y \neq 0 \Longrightarrow 1 - tan\ x * tan\ y = cos\ (x + y)/(cos\ x * cos\ y)$
⟨*proof*⟩

lemma *add-tan-eq*: $cos\ x \neq 0 \Longrightarrow cos\ y \neq 0 \Longrightarrow tan\ x + tan\ y = sin(x + y)/(cos$

$x * cos\ y$)
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *tan-add*:
$cos\ x \neq 0 \Longrightarrow cos\ y \neq 0 \Longrightarrow cos\ (x + y) \neq 0 \Longrightarrow tan\ (x + y) = (tan\ x + tan\ y)/(1 - tan\ x * tan\ y)$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *tan-double*: $cos\ x \neq 0 \Longrightarrow cos\ (2 * x) \neq 0 \Longrightarrow tan\ (2 * x) = (2 * tan\ x)\ /\ (1 - (tan\ x)^2)$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *tan-gt-zero*: $0 < x \Longrightarrow x < pi/2 \Longrightarrow 0 < tan\ x$
⟨*proof*⟩

lemma *tan-less-zero*:
assumes $-\ pi/2 < x$ **and** $x < 0$
shows $tan\ x < 0$
⟨*proof*⟩

lemma *tan-half*: $tan\ x = sin\ (2 * x)\ /\ (cos\ (2 * x) + 1)$
for x :: $'a$::{*real-normed-field*,*banach*,*field*}
⟨*proof*⟩

lemma *tan-30*: $tan\ (pi\ /\ 6) = 1\ /\ sqrt\ 3$
⟨*proof*⟩

lemma *tan-45*: $tan\ (pi\ /\ 4) = 1$
⟨*proof*⟩

lemma *tan-60*: $tan\ (pi\ /\ 3) = sqrt\ 3$
⟨*proof*⟩

lemma *DERIV-tan* [*simp*]: $cos\ x \neq 0 \Longrightarrow DERIV\ tan\ x :> inverse\ ((cos\ x)^2)$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-tan*: $cos\ x \neq 0 \Longrightarrow isCont\ tan\ x$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-tan'* [*simp*,*continuous-intros*]:
fixes a :: $'a$::{*real-normed-field*,*banach*} **and** f :: $'a \Rightarrow 'a$
shows $isCont\ f\ a \Longrightarrow cos\ (f\ a) \neq 0 \Longrightarrow isCont\ (\lambda x.\ tan\ (f\ x))\ a$
⟨*proof*⟩

lemma *tendsto-tan* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'a::\{real\text{-}normed\text{-}field,banach\}$
shows $(f \longrightarrow a)\ F \Longrightarrow cos\ a \neq 0 \Longrightarrow ((\lambda x.\ tan\ (f\ x)) \longrightarrow tan\ a)\ F$
⟨*proof*⟩

lemma *continuous-tan*:
fixes $f :: 'a \Rightarrow 'a::\{real\text{-}normed\text{-}field,banach\}$
shows $continuous\ F\ f \Longrightarrow cos\ (f\ (Lim\ F\ (\lambda x.\ x))) \neq 0 \Longrightarrow continuous\ F\ (\lambda x.\ tan\ (f\ x))$
⟨*proof*⟩

lemma *continuous-on-tan* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a::\{real\text{-}normed\text{-}field,banach\}$
shows $continuous\text{-}on\ s\ f \Longrightarrow (\forall x\in s.\ cos\ (f\ x) \neq 0) \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ tan\ (f\ x))$
⟨*proof*⟩

lemma *continuous-within-tan* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a::\{real\text{-}normed\text{-}field,banach\}$
shows $continuous\ (at\ x\ within\ s)\ f \Longrightarrow$
$cos\ (f\ x) \neq 0 \Longrightarrow continuous\ (at\ x\ within\ s)\ (\lambda x.\ tan\ (f\ x))$
⟨*proof*⟩

lemma *LIM-cos-div-sin*: $(\lambda x.\ cos(x)/sin(x))\ {-pi/2}{\rightarrow}\ 0$
⟨*proof*⟩

lemma *lemma-tan-total*: $0 < y \Longrightarrow \exists x.\ 0 < x \wedge x < pi/2 \wedge y < tan\ x$
⟨*proof*⟩

lemma *tan-total-pos*: $0 \leq y \Longrightarrow \exists x.\ 0 \leq x \wedge x < pi/2 \wedge tan\ x = y$
⟨*proof*⟩

lemma *lemma-tan-total1*: $\exists x.\ -(pi/2) < x \wedge x < (pi/2) \wedge tan\ x = y$
⟨*proof*⟩

lemma *tan-total*: $\exists!\ x.\ -(pi/2) < x \wedge x < (pi/2) \wedge tan\ x = y$
⟨*proof*⟩

lemma *tan-monotone*:
assumes $-\ (pi\ /\ 2) < y$ **and** $y < x$ **and** $x < pi\ /\ 2$
shows $tan\ y < tan\ x$
⟨*proof*⟩

lemma *tan-monotone'*:
assumes $-\ (pi\ /\ 2) < y$
and $y < pi\ /\ 2$
and $-\ (pi\ /\ 2) < x$
and $x < pi\ /\ 2$
shows $y < x \longleftrightarrow tan\ y < tan\ x$

⟨*proof*⟩

lemma *tan-inverse*: $1 \ / \ (tan \ y) = tan \ (pi \ / \ 2 - y)$
⟨*proof*⟩

lemma *tan-periodic-pi*[*simp*]: $tan \ (x + pi) = tan \ x$
⟨*proof*⟩

lemma *tan-periodic-nat*[*simp*]: $tan \ (x + real \ n * pi) = tan \ x$
for $n :: nat$
⟨*proof*⟩

lemma *tan-periodic-int*[*simp*]: $tan \ (x + of\text{-}int \ i * pi) = tan \ x$
⟨*proof*⟩

lemma *tan-periodic-n*[*simp*]: $tan \ (x + numeral \ n * pi) = tan \ x$
⟨*proof*⟩

lemma *tan-minus-45*: $tan \ (-(pi/4)) = -1$
⟨*proof*⟩

lemma *tan-diff*:
$cos \ x \neq 0 \Longrightarrow cos \ y \neq 0 \Longrightarrow cos \ (x - y) \neq 0 \Longrightarrow tan \ (x - y) = (tan \ x - tan \ y)/(1 + tan \ x * tan \ y)$
for $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
⟨*proof*⟩

lemma *tan-pos-pi2-le*: $0 \leq x \Longrightarrow x < pi/2 \Longrightarrow 0 \leq tan \ x$
⟨*proof*⟩

lemma *cos-tan*: $|x| < pi/2 \Longrightarrow cos \ x = 1 \ / \ sqrt \ (1 + tan \ x \ \hat{} \ 2)$
⟨*proof*⟩

lemma *sin-tan*: $|x| < pi/2 \Longrightarrow sin \ x = tan \ x \ / \ sqrt \ (1 + tan \ x \ \hat{} \ 2)$
⟨*proof*⟩

lemma *tan-mono-le*: $-(pi/2) < x \Longrightarrow x \leq y \Longrightarrow y < pi/2 \Longrightarrow tan \ x \leq tan \ y$
⟨*proof*⟩

lemma *tan-mono-lt-eq*:
$-(pi/2) < x \Longrightarrow x < pi/2 \Longrightarrow -(pi/2) < y \Longrightarrow y < pi/2 \Longrightarrow tan \ x < tan \ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *tan-mono-le-eq*:
$-(pi/2) < x \Longrightarrow x < pi/2 \Longrightarrow -(pi/2) < y \Longrightarrow y < pi/2 \Longrightarrow tan \ x \leq tan \ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *tan-bound-pi2*: $|x| < pi/4 \Longrightarrow |tan\ x| < 1$
⟨*proof*⟩

lemma *tan-cot*: $tan(pi/2 - x) = inverse(tan\ x)$
⟨*proof*⟩

105.16 Cotangent

definition *cot* :: $'a \Rightarrow 'a::\{real\text{-}normed\text{-}field,banach\}$
where $cot = (\lambda x.\ cos\ x\ /\ sin\ x)$

lemma *cot-of-real*: $of\text{-}real\ (cot\ x) = (cot\ (of\text{-}real\ x) :: 'a::\{real\text{-}normed\text{-}field,banach\})$
⟨*proof*⟩

lemma *cot-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow cot\ z \in \mathbb{R}$
for $z :: 'a::\{real\text{-}normed\text{-}field,banach\}$
⟨*proof*⟩

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

lemma *cot-pi* [*simp*]: $cot\ pi = 0$
⟨*proof*⟩

lemma *cot-npi* [*simp*]: $cot\ (real\ n * pi) = 0$
for $n :: nat$
⟨*proof*⟩

lemma *cot-minus* [*simp*]: $cot\ (-\ x) = -\ cot\ x$
⟨*proof*⟩

lemma *cot-periodic* [*simp*]: $cot\ (x + 2 * pi) = cot\ x$
⟨*proof*⟩

lemma *cot-altdef*: $cot\ x = inverse\ (tan\ x)$
⟨*proof*⟩

lemma *tan-altdef*: $tan\ x = inverse\ (cot\ x)$
⟨*proof*⟩

lemma *tan-cot'*: $tan\ (pi/2 - x) = cot\ x$
⟨*proof*⟩

lemma *cot-gt-zero*: $0 < x \Longrightarrow x < pi/2 \Longrightarrow 0 < cot\ x$
⟨*proof*⟩

lemma *cot-less-zero*:
assumes *lb*: $-\ pi/2 < x$ **and** $x < 0$
shows $cot\ x < 0$

⟨*proof*⟩

lemma *DERIV-cot* [*simp*]: $sin\ x \neq 0 \Longrightarrow DERIV\ cot\ x :> -inverse\ ((sin\ x)^2)$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-cot*: $sin\ x \neq 0 \Longrightarrow isCont\ cot\ x$
for x :: $'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *isCont-cot'* [*simp*,*continuous-intros*]:
$isCont\ f\ a \Longrightarrow sin\ (f\ a) \neq 0 \Longrightarrow isCont\ (\lambda x.\ cot\ (f\ x))\ a$
for a :: $'a$::{*real-normed-field*,*banach*} **and** f :: $'a \Rightarrow 'a$
⟨*proof*⟩

lemma *tendsto-cot* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow sin\ a \neq 0 \Longrightarrow ((\lambda x.\ cot\ (f\ x)) \longrightarrow cot\ a)\ F$
for f :: $'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-cot*:
$continuous\ F\ f \Longrightarrow sin\ (f\ (Lim\ F\ (\lambda x.\ x))) \neq 0 \Longrightarrow continuous\ F\ (\lambda x.\ cot\ (f\ x))$
for f :: $'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
⟨*proof*⟩

lemma *continuous-on-cot* [*continuous-intros*]:
fixes f :: $'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
shows $continuous\text{-}on\ s\ f \Longrightarrow (\forall x \in s.\ sin\ (f\ x) \neq 0) \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ cot\ (f\ x))$
⟨*proof*⟩

lemma *continuous-within-cot* [*continuous-intros*]:
fixes f :: $'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
shows $continuous\ (at\ x\ within\ s)\ f \Longrightarrow sin\ (f\ x) \neq 0 \Longrightarrow continuous\ (at\ x\ within\ s)\ (\lambda x.\ cot\ (f\ x))$
⟨*proof*⟩

105.17 Inverse Trigonometric Functions

definition *arcsin* :: $real \Rightarrow real$
where $arcsin\ y = (THE\ x.\ -(pi/2) \leq x \wedge x \leq pi/2 \wedge sin\ x = y)$

definition *arccos* :: $real \Rightarrow real$
where $arccos\ y = (THE\ x.\ 0 \leq x \wedge x \leq pi \wedge cos\ x = y)$

definition *arctan* :: $real \Rightarrow real$
where $arctan\ y = (THE\ x.\ -(pi/2) < x \wedge x < pi/2 \wedge tan\ x = y)$

lemma *arcsin*: $- 1 \leq y \Longrightarrow y \leq 1 \Longrightarrow - (pi/2) \leq arcsin\ y \wedge arcsin\ y \leq pi/2 \wedge sin\ (arcsin\ y) = y$
⟨*proof*⟩

lemma *arcsin-pi*: $- 1 \leq y \Longrightarrow y \leq 1 \Longrightarrow - (pi/2) \leq arcsin\ y \wedge arcsin\ y \leq pi \wedge sin\ (arcsin\ y) = y$
⟨*proof*⟩

lemma *sin-arcsin* [*simp*]: $- 1 \leq y \Longrightarrow y \leq 1 \Longrightarrow sin\ (arcsin\ y) = y$
⟨*proof*⟩

lemma *arcsin-bounded*: $- 1 \leq y \Longrightarrow y \leq 1 \Longrightarrow - (pi/2) \leq arcsin\ y \wedge arcsin\ y \leq pi/2$
⟨*proof*⟩

lemma *arcsin-lbound*: $- 1 \leq y \Longrightarrow y \leq 1 \Longrightarrow - (pi/2) \leq arcsin\ y$
⟨*proof*⟩

lemma *arcsin-ubound*: $- 1 \leq y \Longrightarrow y \leq 1 \Longrightarrow arcsin\ y \leq pi/2$
⟨*proof*⟩

lemma *arcsin-lt-bounded*: $- 1 < y \Longrightarrow y < 1 \Longrightarrow - (pi/2) < arcsin\ y \wedge arcsin\ y < pi/2$
⟨*proof*⟩

lemma *arcsin-sin*: $- (pi/2) \leq x \Longrightarrow x \leq pi/2 \Longrightarrow arcsin\ (sin\ x) = x$
⟨*proof*⟩

lemma *arcsin-0* [*simp*]: $arcsin\ 0 = 0$
⟨*proof*⟩

lemma *arcsin-1* [*simp*]: $arcsin\ 1 = pi/2$
⟨*proof*⟩

lemma *arcsin-minus-1* [*simp*]: $arcsin\ (- 1) = - (pi/2)$
⟨*proof*⟩

lemma *arcsin-minus*: $- 1 \leq x \Longrightarrow x \leq 1 \Longrightarrow arcsin\ (- x) = - arcsin\ x$
⟨*proof*⟩

lemma *arcsin-eq-iff*: $|x| \leq 1 \Longrightarrow |y| \leq 1 \Longrightarrow arcsin\ x = arcsin\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *cos-arcsin-nonzero*: $- 1 < x \Longrightarrow x < 1 \Longrightarrow cos\ (arcsin\ x) \neq 0$
⟨*proof*⟩

lemma *arccos*: $- 1 \leq y \Longrightarrow y \leq 1 \Longrightarrow 0 \leq arccos\ y \wedge arccos\ y \leq pi \wedge cos\ (arccos\ y) = y$
⟨*proof*⟩

lemma *cos-arccos* [*simp*]: $-\ 1 \le y \Longrightarrow y \le 1 \Longrightarrow cos\ (arccos\ y) = y$
⟨*proof*⟩

lemma *arccos-bounded*: $-\ 1 \le y \Longrightarrow y \le 1 \Longrightarrow 0 \le arccos\ y \wedge arccos\ y \le pi$
⟨*proof*⟩

lemma *arccos-lbound*: $-\ 1 \le y \Longrightarrow y \le 1 \Longrightarrow 0 \le arccos\ y$
⟨*proof*⟩

lemma *arccos-ubound*: $-\ 1 \le y \Longrightarrow y \le 1 \Longrightarrow arccos\ y \le pi$
⟨*proof*⟩

lemma *arccos-lt-bounded*: $-\ 1 < y \Longrightarrow y < 1 \Longrightarrow 0 < arccos\ y \wedge arccos\ y < pi$
⟨*proof*⟩

lemma *arccos-cos*: $0 \le x \Longrightarrow x \le pi \Longrightarrow arccos\ (cos\ x) = x$
⟨*proof*⟩

lemma *arccos-cos2*: $x \le 0 \Longrightarrow -\ pi \le x \Longrightarrow arccos\ (cos\ x) = -x$
⟨*proof*⟩

lemma *cos-arcsin*: $-\ 1 \le x \Longrightarrow x \le 1 \Longrightarrow cos\ (arcsin\ x) = sqrt\ (1 - x^2)$
⟨*proof*⟩

lemma *sin-arccos*: $-\ 1 \le x \Longrightarrow x \le 1 \Longrightarrow sin\ (arccos\ x) = sqrt\ (1 - x^2)$
⟨*proof*⟩

lemma *arccos-0* [*simp*]: $arccos\ 0 = pi/2$
⟨*proof*⟩

lemma *arccos-1* [*simp*]: $arccos\ 1 = 0$
⟨*proof*⟩

lemma *arccos-minus-1* [*simp*]: $arccos\ (-\ 1) = pi$
⟨*proof*⟩

lemma *arccos-minus*: $-1 \le x \Longrightarrow x \le 1 \Longrightarrow arccos\ (-\ x) = pi - arccos\ x$
⟨*proof*⟩

corollary *arccos-minus-abs*:
assumes $|x| \le 1$
shows $arccos\ (-\ x) = pi - arccos\ x$
⟨*proof*⟩

lemma *sin-arccos-nonzero*: $-\ 1 < x \Longrightarrow x < 1 \Longrightarrow sin\ (arccos\ x) \ne 0$
⟨*proof*⟩

lemma *arctan*: $-\ (pi/2) < arctan\ y \wedge arctan\ y < pi/2 \wedge tan\ (arctan\ y) = y$

⟨proof⟩

lemma *tan-arctan*: $tan\ (arctan\ y) = y$
⟨proof⟩

lemma *arctan-bounded*: $-\ (pi/2) < arctan\ y \wedge arctan\ y < pi/2$
⟨proof⟩

lemma *arctan-lbound*: $-\ (pi/2) < arctan\ y$
⟨proof⟩

lemma *arctan-ubound*: $arctan\ y < pi/2$
⟨proof⟩

lemma *arctan-unique*:
assumes $-(pi/2) < x$
and $x < pi/2$
and $tan\ x = y$
shows $arctan\ y = x$
⟨proof⟩

lemma *arctan-tan*: $-(pi/2) < x \Longrightarrow x < pi/2 \Longrightarrow arctan\ (tan\ x) = x$
⟨proof⟩

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

lemma *arctan-minus*: $arctan\ (-\ x) = -\ arctan\ x$
⟨proof⟩

lemma *cos-arctan-not-zero* [*simp*]: $cos\ (arctan\ x) \neq 0$
⟨proof⟩

lemma *cos-arctan*: $cos\ (arctan\ x) = 1\ /\ sqrt\ (1 + x^2)$
⟨proof⟩

lemma *sin-arctan*: $sin\ (arctan\ x) = x\ /\ sqrt\ (1 + x^2)$
⟨proof⟩

lemma *tan-sec*: $cos\ x \neq 0 \Longrightarrow 1 + (tan\ x)^2 = (inverse\ (cos\ x))^2$
for x :: $'a$::{*real-normed-field*,*banach*,*field*}
⟨proof⟩

lemma *arctan-less-iff*: $arctan\ x < arctan\ y \longleftrightarrow x < y$
⟨proof⟩

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

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

lemma *zero-less-arctan-iff* [*simp*]: $0 < arctan\ x \longleftrightarrow 0 < x$
⟨*proof*⟩

lemma *arctan-less-zero-iff* [*simp*]: $arctan\ x < 0 \longleftrightarrow x < 0$
⟨*proof*⟩

lemma *zero-le-arctan-iff* [*simp*]: $0 \leq arctan\ x \longleftrightarrow 0 \leq x$
⟨*proof*⟩

lemma *arctan-le-zero-iff* [*simp*]: $arctan\ x \leq 0 \longleftrightarrow x \leq 0$
⟨*proof*⟩

lemma *arctan-eq-zero-iff* [*simp*]: $arctan\ x = 0 \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *continuous-on-arcsin'*: *continuous-on* $\{-1\ ..\ 1\}$ *arcsin*
⟨*proof*⟩

lemma *continuous-on-arcsin* [*continuous-intros*]:
continuous-on $s\ f \Longrightarrow (\forall x{\in}s.\ -1 \leq f\ x \wedge f\ x \leq 1) \Longrightarrow$ *continuous-on* $s\ (\lambda x.$ $arcsin\ (f\ x))$
⟨*proof*⟩

lemma *isCont-arcsin*: $-1 < x \Longrightarrow x < 1 \Longrightarrow isCont\ arcsin\ x$
⟨*proof*⟩

lemma *continuous-on-arccos'*: *continuous-on* $\{-1\ ..\ 1\}$ *arccos*
⟨*proof*⟩

lemma *continuous-on-arccos* [*continuous-intros*]:
continuous-on $s\ f \Longrightarrow (\forall x{\in}s.\ -1 \leq f\ x \wedge f\ x \leq 1) \Longrightarrow$ *continuous-on* $s\ (\lambda x.$ $arccos\ (f\ x))$
⟨*proof*⟩

lemma *isCont-arccos*: $-1 < x \Longrightarrow x < 1 \Longrightarrow isCont\ arccos\ x$
⟨*proof*⟩

lemma *isCont-arctan*: $isCont\ arctan\ x$
⟨*proof*⟩

lemma *tendsto-arctan* [*tendsto-intros*]: $(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ arctan\ (f\ x)) \longrightarrow arctan\ x)\ F$
⟨*proof*⟩

lemma *continuous-arctan* [*continuous-intros*]: $continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ arctan\ (f\ x))$

⟨*proof*⟩

lemma *continuous-on-arctan* [*continuous-intros*]:
continuous-on s f $\Longrightarrow$ *continuous-on s* ($\lambda x.$ *arctan* (*f x*))
⟨*proof*⟩

lemma *DERIV-arcsin*: $-$ *1* $<$ *x* $\Longrightarrow$ *x* $<$ *1* $\Longrightarrow$ *DERIV arcsin x* :> *inverse* (*sqrt* (*1* $-$ x^2))
⟨*proof*⟩

lemma *DERIV-arccos*: $-$ *1* $<$ *x* $\Longrightarrow$ *x* $<$ *1* $\Longrightarrow$ *DERIV arccos x* :> *inverse* ($-$ *sqrt* (*1* $-$ x^2))
⟨*proof*⟩

lemma *DERIV-arctan*: *DERIV arctan x* :> *inverse* (*1* + x^2)
⟨*proof*⟩

declare
DERIV-arcsin[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-arcsin[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]
DERIV-arccos[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-arccos[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]
DERIV-arctan[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-arctan[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemma *filterlim-tan-at-right*: *filterlim tan at-bot* (*at-right* ($-$ (*pi*/*2*)))
⟨*proof*⟩

lemma *filterlim-tan-at-left*: *filterlim tan at-top* (*at-left* (*pi*/*2*))
⟨*proof*⟩

lemma *tendsto-arctan-at-top*: (*arctan* $\longrightarrow$ (*pi*/*2*)) *at-top*
⟨*proof*⟩

lemma *tendsto-arctan-at-bot*: (*arctan* $\longrightarrow$ $-$ (*pi*/*2*)) *at-bot*
⟨*proof*⟩

105.18 Prove Totality of the Trigonometric Functions

lemma *cos-arccos-abs*: $|y| \leq$ *1* $\Longrightarrow$ *cos* (*arccos y*) = *y*
⟨*proof*⟩

lemma *sin-arccos-abs*: $|y| \leq$ *1* $\Longrightarrow$ *sin* (*arccos y*) = *sqrt* (*1* $-$ y^2)
⟨*proof*⟩

lemma *sin-mono-less-eq*:
$-$ (*pi*/*2*) $\leq$ *x* $\Longrightarrow$ *x* $\leq$ *pi*/*2* $\Longrightarrow$ $-$ (*pi*/*2*) $\leq$ *y* $\Longrightarrow$ *y* $\leq$ *pi*/*2* $\Longrightarrow$ *sin x* $<$ *sin y* $\longleftrightarrow$ *x* $<$ *y*
⟨*proof*⟩

lemma *sin-mono-le-eq*:
$-(pi/2) \leq x \Longrightarrow x \leq pi/2 \Longrightarrow -(pi/2) \leq y \Longrightarrow y \leq pi/2 \Longrightarrow sin\ x \leq sin\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *sin-inj-pi*:
$-(pi/2) \leq x \Longrightarrow x \leq pi/2 \Longrightarrow -(pi/2) \leq y \Longrightarrow y \leq pi/2 \Longrightarrow sin\ x = sin\ y \Longrightarrow x = y$
⟨*proof*⟩

lemma *cos-mono-less-eq*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow 0 \leq y \Longrightarrow y \leq pi \Longrightarrow cos\ x < cos\ y \longleftrightarrow y < x$
⟨*proof*⟩

lemma *cos-mono-le-eq*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow 0 \leq y \Longrightarrow y \leq pi \Longrightarrow cos\ x \leq cos\ y \longleftrightarrow y \leq x$
⟨*proof*⟩

lemma *cos-inj-pi*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow 0 \leq y \Longrightarrow y \leq pi \Longrightarrow cos\ x = cos\ y \Longrightarrow x = y$
⟨*proof*⟩

lemma *arccos-le-pi2*: $[\![0 \leq y;\ y \leq 1]\!] \Longrightarrow arccos\ y \leq pi/2$
⟨*proof*⟩

lemma *sincos-total-pi-half*:
assumes $0 \leq x\ 0 \leq y\ x^2 + y^2 = 1$
shows $\exists t.\ 0 \leq t \wedge t \leq pi/2 \wedge x = cos\ t \wedge y = sin\ t$
⟨*proof*⟩

lemma *sincos-total-pi*:
assumes $0 \leq y\ x^2 + y^2 = 1$
shows $\exists t.\ 0 \leq t \wedge t \leq pi \wedge x = cos\ t \wedge y = sin\ t$
⟨*proof*⟩

lemma *sincos-total-2pi-le*:
assumes $x^2 + y^2 = 1$
shows $\exists t.\ 0 \leq t \wedge t \leq 2 * pi \wedge x = cos\ t \wedge y = sin\ t$
⟨*proof*⟩

lemma *sincos-total-2pi*:
assumes $x^2 + y^2 = 1$
obtains t **where** $0 \leq t\ t < 2{*}pi\ x = cos\ t\ y = sin\ t$
⟨*proof*⟩

lemma *arcsin-less-mono*: $|x| \leq 1 \Longrightarrow |y| \leq 1 \Longrightarrow arcsin\ x < arcsin\ y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *arcsin-le-mono*: $|x| \le 1 \Longrightarrow |y| \le 1 \Longrightarrow arcsin\ x \le arcsin\ y \longleftrightarrow x \le y$
⟨*proof*⟩

lemma *arcsin-less-arcsin*: $-1 \le x \Longrightarrow x < y \Longrightarrow y \le 1 \Longrightarrow arcsin\ x < arcsin\ y$
⟨*proof*⟩

lemma *arcsin-le-arcsin*: $-1 \le x \Longrightarrow x \le y \Longrightarrow y \le 1 \Longrightarrow arcsin\ x \le arcsin\ y$
⟨*proof*⟩

lemma *arccos-less-mono*: $|x| \le 1 \Longrightarrow |y| \le 1 \Longrightarrow arccos\ x < arccos\ y \longleftrightarrow y < x$
⟨*proof*⟩

lemma *arccos-le-mono*: $|x| \le 1 \Longrightarrow |y| \le 1 \Longrightarrow arccos\ x \le arccos\ y \longleftrightarrow y \le x$
⟨*proof*⟩

lemma *arccos-less-arccos*: $-1 \le x \Longrightarrow x < y \Longrightarrow y \le 1 \Longrightarrow arccos\ y < arccos\ x$
⟨*proof*⟩

lemma *arccos-le-arccos*: $-1 \le x \Longrightarrow x \le y \Longrightarrow y \le 1 \Longrightarrow arccos\ y \le arccos\ x$
⟨*proof*⟩

lemma *arccos-eq-iff*: $|x| \le 1 \land |y| \le 1 \Longrightarrow arccos\ x = arccos\ y \longleftrightarrow x = y$
⟨*proof*⟩

105.19 Machin's formula

lemma *arctan-one*: $arctan\ 1 = pi\ /\ 4$
⟨*proof*⟩

lemma *tan-total-pi4*:
assumes $|x| < 1$
shows $\exists z.\ -(pi\ /\ 4) < z \land z < pi\ /\ 4 \land tan\ z = x$
⟨*proof*⟩

lemma *arctan-add*:
assumes $|x| \le 1$ $|y| < 1$
shows $arctan\ x + arctan\ y = arctan\ ((x + y)\ /\ (1 - x * y))$
⟨*proof*⟩

lemma *arctan-double*: $|x| < 1 \Longrightarrow 2 * arctan\ x = arctan\ ((2 * x)\ /\ (1 - x^2))$
⟨*proof*⟩

theorem *machin*: $pi\ /\ 4 = 4 * arctan\ (1\ /\ 5) - arctan\ (1\ /\ 239)$
⟨*proof*⟩

lemma *machin-Euler*: $5 * arctan\ (1\ /\ 7) + 2 * arctan\ (3\ /\ 79) = pi\ /\ 4$
⟨*proof*⟩

105.20 Introducing the inverse tangent power series

lemma *monoseq-arctan-series*:
 fixes x :: *real*
 assumes $|x| \leq 1$
 shows *monoseq* $(\lambda n.\ 1\ /\ real\ (n * 2 + 1) * x\hat{}(n * 2 + 1))$
  (**is** *monoseq ?a*)
⟨*proof*⟩

lemma *zeroseq-arctan-series*:
 fixes x :: *real*
 assumes $|x| \leq 1$
 shows $(\lambda n.\ 1\ /\ real\ (n * 2 + 1) * x\hat{}(n * 2 + 1)) \longrightarrow 0$
  (**is** $?a \longrightarrow 0$)
⟨*proof*⟩

lemma *summable-arctan-series*:
 fixes n :: *nat*
 assumes $|x| \leq 1$
 shows *summable* $(\lambda\ k.\ (-1)\ \hat{}k * (1\ /\ real\ (k*2+1) * x\ \hat{}\ (k*2+1)))$
  (**is** *summable* $(?c\ x)$)
 ⟨*proof*⟩

lemma *DERIV-arctan-series*:
 assumes $|x| < 1$
 shows *DERIV* $(\lambda x'.\ \sum k.\ (-1)\ \hat{}k * (1\ /\ real\ (k * 2 + 1) * x'\ \hat{}\ (k * 2 + 1)))$
x :>
  $(\sum k.\ (-1)\ \hat{}k * x\hat{}(k * 2))$
  (**is** *DERIV ?arctan - :> ?Int*)
⟨*proof*⟩

lemma *arctan-series*:
 assumes $|x| \leq 1$
 shows *arctan* $x = (\sum k.\ (-1)\ \hat{}k * (1\ /\ real\ (k * 2 + 1) * x\ \hat{}\ (k * 2 + 1)))$
  (**is** - = *suminf* $(\lambda\ n.\ ?c\ x\ n)$)
⟨*proof*⟩

lemma *arctan-half*: *arctan* $x = 2 *$ *arctan* $(x\ /\ (1 + sqrt(1 + x^2)))$
 for x :: *real*
⟨*proof*⟩

lemma *arctan-monotone*: $x < y \Longrightarrow$ *arctan* $x <$ *arctan* y
 ⟨*proof*⟩

lemma *arctan-monotone'*: $x \leq y \Longrightarrow$ *arctan* $x \leq$ *arctan* y
 ⟨*proof*⟩

lemma *arctan-inverse*:
 assumes $x \neq 0$
 shows *arctan* $(1\ /\ x) = sgn\ x * pi\ /\ 2 -$ *arctan* x

⟨*proof*⟩

theorem *pi-series*: $pi\ /\ 4 = (\sum k.\ (-1)\ \hat{}k * 1\ /\ real\ (k * 2 + 1))$
(**is** - = *?SUM*)
⟨*proof*⟩

105.21 Existence of Polar Coordinates

lemma *cos-x-y-le-one*: $|x\ /\ sqrt\ (x^2 + y^2)| \leq 1$
⟨*proof*⟩

lemmas *cos-arccos-lemma1* = *cos-arccos-abs* [*OF cos-x-y-le-one*]

lemmas *sin-arccos-lemma1* = *sin-arccos-abs* [*OF cos-x-y-le-one*]

lemma *polar-Ex*: $\exists\, r{::}real.\ \exists\, a.\ x = r * cos\ a \wedge y = r * sin\ a$
⟨*proof*⟩

105.22 Basics about polynomial functions: products, extremal behaviour and root counts

lemma *pairs-le-eq-Sigma*: $\{(i, j).\ i + j \leq m\} = Sigma\ (atMost\ m)\ (\lambda r.\ atMost\ (m - r))$
for $m :: nat$
⟨*proof*⟩

lemma *sum-up-index-split*: $(\sum k{\leq}m + n.\ f\ k) = (\sum k{\leq}m.\ f\ k) + (\sum k = Suc\ m..m + n.\ f\ k)$
⟨*proof*⟩

lemma *Sigma-interval-disjoint*: $(SIGMA\ i{:}A.\ \{..v\ i\}) \cap (SIGMA\ i{:}A.\{v\ i{<}..w\}) = \{\}$
for $w :: 'a{::}order$
⟨*proof*⟩

lemma *product-atMost-eq-Un*: $A \times \{..m\} = (SIGMA\ i{:}A.\{..m - i\}) \cup (SIGMA\ i{:}A.\{m - i{<}..m\})$
for $m :: nat$
⟨*proof*⟩

lemma *polynomial-product*:
fixes $x :: 'a{::}idom$
assumes m: $\bigwedge i.\ i > m \Longrightarrow a\ i = 0$
and n: $\bigwedge j.\ j > n \Longrightarrow b\ j = 0$
shows $(\sum i{\leq}m.\ (a\ i) * x\ \hat{}\ i) * (\sum j{\leq}n.\ (b\ j) * x\ \hat{}\ j) =$
$(\sum r{\leq}m + n.\ (\sum k{\leq}r.\ (a\ k) * (b\ (r - k))) * x\ \hat{}\ r)$
⟨*proof*⟩

lemma *polynomial-product-nat*:

fixes x :: *nat*
assumes m: $\bigwedge i.\ i > m \Longrightarrow a\ i = 0$
and n: $\bigwedge j.\ j > n \Longrightarrow b\ j = 0$
shows $(\sum i{\le}m.\ (a\ i) * x\ \hat{}\ i) * (\sum j{\le}n.\ (b\ j) * x\ \hat{}\ j) =$
$(\sum r{\le}m + n.\ (\sum k{\le}r.\ (a\ k) * (b\ (r - k))) * x\ \hat{}\ r)$
⟨*proof*⟩

lemma *polyfun-diff*:
fixes x :: $'a$::*idom*
assumes $1 \le n$
shows $(\sum i{\le}n.\ a\ i * x\hat{}i) - (\sum i{\le}n.\ a\ i * y\hat{}i) =$
$(x - y) * (\sum j{<}n.\ (\sum i{=}Suc\ j..n.\ a\ i * y\hat{}(i - j - 1)) * x\hat{}j)$
⟨*proof*⟩

lemma *polyfun-diff-alt*:
fixes x :: $'a$::*idom*
assumes $1 \le n$
shows $(\sum i{\le}n.\ a\ i * x\hat{}i) - (\sum i{\le}n.\ a\ i * y\hat{}i) =$
$(x - y) * ((\sum j{<}n.\ \sum k{<}n{-}j.\ a(j + k + 1) * y\hat{}k * x\hat{}j))$
⟨*proof*⟩

lemma *polyfun-linear-factor*:
fixes a :: $'a$::*idom*
shows $\exists\, b.\ \forall\, z.\ (\sum i{\le}n.\ c(i) * z\hat{}i) = (z - a) * (\sum i{<}n.\ b(i) * z\hat{}i) + (\sum i{\le}n.\ c(i) * a\hat{}i)$
⟨*proof*⟩

lemma *polyfun-linear-factor-root*:
fixes a :: $'a$::*idom*
assumes $(\sum i{\le}n.\ c(i) * a\hat{}i) = 0$
obtains b **where** $\bigwedge z.\ (\sum i{\le}n.\ c\ i * z\hat{}i) = (z - a) * (\sum i{<}n.\ b\ i * z\hat{}i)$
⟨*proof*⟩

lemma *isCont-polynom*: *isCont* $(\lambda w.\ \sum i{\le}n.\ c\ i * w\hat{}i)\ a$
for c :: *nat* $\Rightarrow$ $'a$::*real-normed-div-algebra*
⟨*proof*⟩

lemma *zero-polynom-imp-zero-coeffs*:
fixes c :: *nat* $\Rightarrow$ $'a$::{*ab-semigroup-mult*,*real-normed-div-algebra*}
assumes $\bigwedge w.\ (\sum i{\le}n.\ c\ i * w\hat{}i) = 0\ \ k \le n$
shows $c\ k = 0$
⟨*proof*⟩

lemma *polyfun-rootbound*:
fixes c :: *nat* $\Rightarrow$ $'a$::{*idom*,*real-normed-div-algebra*}
assumes $c\ k \ne 0\ k{\le}n$
shows *finite* $\{z.\ (\sum i{\le}n.\ c(i) * z\hat{}i) = 0\} \wedge$ *card* $\{z.\ (\sum i{\le}n.\ c(i) * z\hat{}i) = 0\}$

$\leq n$
⟨*proof*⟩

lemma
fixes $c :: nat \Rightarrow 'a::\{idom,real\text{-}normed\text{-}div\text{-}algebra\}$
assumes $c\ k \neq 0\ k \leq n$
shows *polyfun-roots-finite*: $finite\ \{z.\ (\sum i \leq n.\ c(i) * z\hat{\ }i) = 0\}$
and *polyfun-roots-card*: $card\ \{z.\ (\sum i \leq n.\ c(i) * z\hat{\ }i) = 0\} \leq n$
⟨*proof*⟩

lemma *polyfun-finite-roots*:
fixes $c :: nat \Rightarrow 'a::\{idom,real\text{-}normed\text{-}div\text{-}algebra\}$
shows $finite\ \{x.\ (\sum i \leq n.\ c\ i * x\hat{\ }i) = 0\} \longleftrightarrow (\exists\, i \leq n.\ c\ i \neq 0)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *polyfun-eq-0*: $(\forall\, x.\ (\sum i \leq n.\ c\ i * x\hat{\ }i) = 0) \longleftrightarrow (\forall\, i \leq n.\ c\ i = 0)$
for $c :: nat \Rightarrow 'a::\{idom,real\text{-}normed\text{-}div\text{-}algebra\}$

⟨*proof*⟩

lemma *polyfun-eq-coeffs*: $(\forall\, x.\ (\sum i \leq n.\ c\ i * x\hat{\ }i) = (\sum i \leq n.\ d\ i * x\hat{\ }i)) \longleftrightarrow (\forall\, i \leq n.\ c\ i = d\ i)$
for $c :: nat \Rightarrow 'a::\{idom,real\text{-}normed\text{-}div\text{-}algebra\}$
⟨*proof*⟩

lemma *polyfun-eq-const*:
fixes $c :: nat \Rightarrow 'a::\{idom,real\text{-}normed\text{-}div\text{-}algebra\}$
shows $(\forall\, x.\ (\sum i \leq n.\ c\ i * x\hat{\ }i) = k) \longleftrightarrow c\ 0 = k \wedge (\forall\, i \in \{1..n\}.\ c\ i = 0)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *root-polyfun*:
fixes $z :: 'a::idom$
assumes $1 \leq n$
shows $z\hat{\ }n = a \longleftrightarrow (\sum i \leq n.\ (if\ i = 0\ then\ -a\ else\ if\ i=n\ then\ 1\ else\ 0) * z\hat{\ }i) = 0$
⟨*proof*⟩

lemma
assumes $SORT\text{-}CONSTRAINT('a::\{idom,real\text{-}normed\text{-}div\text{-}algebra\})$
and $1 \leq n$
shows *finite-roots-unity*: $finite\ \{z::'a.\ z\hat{\ }n = 1\}$
and *card-roots-unity*: $card\ \{z::'a.\ z\hat{\ }n = 1\} \leq n$
⟨*proof*⟩

105.23 Simprocs for root and power literals

lemma *numeral-powr-numeral-real* [*simp*]:

numeral m powr numeral n = (*numeral m* ^ *numeral n* :: *real*)
⟨*proof*⟩

context
begin

private lemma *sqrt-numeral-simproc-aux*:
assumes $m * m \equiv n$
shows *sqrt* (*numeral n* :: *real*) ≡ *numeral m*
⟨*proof*⟩ **lemma** *root-numeral-simproc-aux*:
assumes *Num.pow m n* ≡ *x*
shows *root* (*numeral n*) (*numeral x* :: *real*) ≡ *numeral m*
⟨*proof*⟩ **lemma** *powr-numeral-simproc-aux*:
assumes *Num.pow y n* = *x*
shows *numeral x powr* (*m* / *numeral n* :: *real*) ≡ *numeral y powr m*
⟨*proof*⟩ **lemma** *numeral-powr-inverse-eq*:
numeral x powr (*inverse* (*numeral n*)) = *numeral x powr* (*1* / *numeral n* :: *real*)
⟨*proof*⟩

⟨*ML*⟩

end

⟨*ML*⟩

lemma *root 100 1267650600228229401496703205376* = *2*
⟨*proof*⟩

lemma *sqrt 196* = *14*
⟨*proof*⟩

lemma *256 powr* (*7* / *4* :: *real*) = *16384*
⟨*proof*⟩

lemma *27 powr* (*inverse 3*) = (*3*::*real*)
⟨*proof*⟩

end

106 Complex Numbers: Rectangular and Polar Representations

theory *Complex*
imports *Transcendental*
begin

We use the **codatatype** command to define the type of complex numbers.

This allows us to use **primcorec** to define complex functions by defining their real and imaginary result separately.

codatatype *complex* = *Complex* (*Re*: *real*) (*Im*: *real*)

lemma *complex-surj*: *Complex* (*Re z*) (*Im z*) = *z*
⟨*proof*⟩

lemma *complex-eqI* [*intro?*]: *Re x* = *Re y* ⟹ *Im x* = *Im y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *complex-eq-iff*: *x* = *y* ⟷ *Re x* = *Re y* ∧ *Im x* = *Im y*
⟨*proof*⟩

106.1 Addition and Subtraction

instantiation *complex* :: *ab-group-add*
begin

primcorec *zero-complex*
where
Re 0 = *0*
| *Im 0* = *0*

primcorec *plus-complex*
where
Re (*x* + *y*) = *Re x* + *Re y*
| *Im* (*x* + *y*) = *Im x* + *Im y*

primcorec *uminus-complex*
where
Re (− *x*) = − *Re x*
| *Im* (− *x*) = − *Im x*

primcorec *minus-complex*
where
Re (*x* − *y*) = *Re x* − *Re y*
| *Im* (*x* − *y*) = *Im x* − *Im y*

instance
⟨*proof*⟩

end

106.2 Multiplication and Division

instantiation *complex* :: *field*
begin

primcorec *one-complex*

where
$Re\ 1 = 1$
$|\ Im\ 1 = 0$

primcorec *times-complex*
where
$Re\ (x * y) = Re\ x * Re\ y - Im\ x * Im\ y$
$|\ Im\ (x * y) = Re\ x * Im\ y + Im\ x * Re\ y$

primcorec *inverse-complex*
where
$Re\ (inverse\ x) = Re\ x\ /\ ((Re\ x)^2 + (Im\ x)^2)$
$|\ Im\ (inverse\ x) = -\ Im\ x\ /\ ((Re\ x)^2 + (Im\ x)^2)$

definition $x\ div\ y = x * inverse\ y$ **for** $x\ y :: complex$

instance
⟨*proof*⟩

end

lemma *Re-divide*: $Re\ (x\ /\ y) = (Re\ x * Re\ y + Im\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2)$
⟨*proof*⟩

lemma *Im-divide*: $Im\ (x\ /\ y) = (Im\ x * Re\ y - Re\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2)$
⟨*proof*⟩

lemma *Complex-divide*:
$(x\ /\ y) = Complex\ ((Re\ x * Re\ y + Im\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2))$
$((Im\ x * Re\ y - Re\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2))$
⟨*proof*⟩

lemma *Re-power2*: $Re\ (x\ \hat{}\ 2) = (Re\ x)\hat{}2 - (Im\ x)\hat{}2$
⟨*proof*⟩

lemma *Im-power2*: $Im\ (x\ \hat{}\ 2) = 2 * Re\ x * Im\ x$
⟨*proof*⟩

lemma *Re-power-real* [*simp*]: $Im\ x = 0 \Longrightarrow Re\ (x\ \hat{}\ n) = Re\ x\ \hat{}\ n$
⟨*proof*⟩

lemma *Im-power-real* [*simp*]: $Im\ x = 0 \Longrightarrow Im\ (x\ \hat{}\ n) = 0$
⟨*proof*⟩

106.3 Scalar Multiplication

instantiation *complex* :: *real-field*

begin

primcorec *scaleR-complex*
  where
    Re (*scaleR r x*) = *r* ∗ *Re x*
  | *Im* (*scaleR r x*) = *r* ∗ *Im x*

instance
⟨*proof*⟩

end

106.4 Numerals, Arithmetic, and Embedding from R

abbreviation *complex-of-real* :: *real* ⇒ *complex*
  where *complex-of-real* ≡ *of-real*

declare [[*coercion of-real* :: *real* ⇒ *complex*]]
declare [[*coercion of-rat* :: *rat* ⇒ *complex*]]
declare [[*coercion of-int* :: *int* ⇒ *complex*]]
declare [[*coercion of-nat* :: *nat* ⇒ *complex*]]

lemma *complex-Re-of-nat* [*simp*]: *Re* (*of-nat n*) = *of-nat n*
  ⟨*proof*⟩

lemma *complex-Im-of-nat* [*simp*]: *Im* (*of-nat n*) = *0*
  ⟨*proof*⟩

lemma *complex-Re-of-int* [*simp*]: *Re* (*of-int z*) = *of-int z*
  ⟨*proof*⟩

lemma *complex-Im-of-int* [*simp*]: *Im* (*of-int z*) = *0*
  ⟨*proof*⟩

lemma *complex-Re-numeral* [*simp*]: *Re* (*numeral v*) = *numeral v*
  ⟨*proof*⟩

lemma *complex-Im-numeral* [*simp*]: *Im* (*numeral v*) = *0*
  ⟨*proof*⟩

lemma *Re-complex-of-real* [*simp*]: *Re* (*complex-of-real z*) = *z*
  ⟨*proof*⟩

lemma *Im-complex-of-real* [*simp*]: *Im* (*complex-of-real z*) = *0*
  ⟨*proof*⟩

lemma *Re-divide-numeral* [*simp*]: *Re* (*z* / *numeral w*) = *Re z* / *numeral w*
  ⟨*proof*⟩

lemma *Im-divide-numeral* [*simp*]: *Im* (*z* / *numeral w*) = *Im z* / *numeral w*
⟨*proof*⟩

lemma *Re-divide-of-nat* [*simp*]: *Re* (*z* / *of-nat n*) = *Re z* / *of-nat n*
⟨*proof*⟩

lemma *Im-divide-of-nat* [*simp*]: *Im* (*z* / *of-nat n*) = *Im z* / *of-nat n*
⟨*proof*⟩

lemma *of-real-Re* [*simp*]: $z \in \mathbb{R} \Longrightarrow$ *of-real* (*Re z*) = *z*
⟨*proof*⟩

lemma *complex-Re-fact* [*simp*]: *Re* (*fact n*) = *fact n*
⟨*proof*⟩

lemma *complex-Im-fact* [*simp*]: *Im* (*fact n*) = *0*
⟨*proof*⟩

106.5 The Complex Number *i*

primcorec *imaginary-unit* :: *complex* (i)
where
Re i = *0*
| *Im* i = *1*

lemma *Complex-eq*: *Complex a b* = *a* + i ∗ *b*
⟨*proof*⟩

lemma *complex-eq*: *a* = *Re a* + i ∗ *Im a*
⟨*proof*⟩

lemma *fun-complex-eq*: *f* = (λ*x*. *Re* (*f x*) + i ∗ *Im* (*f x*))
⟨*proof*⟩

lemma *i-squared* [*simp*]: i ∗ i = −*1*
⟨*proof*⟩

lemma *power2-i* [*simp*]: i^2 = −*1*
⟨*proof*⟩

lemma *inverse-i* [*simp*]: *inverse* i = − i
⟨*proof*⟩

lemma *divide-i* [*simp*]: *x* / i = − i ∗ *x*
⟨*proof*⟩

lemma *complex-i-mult-minus* [*simp*]: i ∗ (i ∗ *x*) = − *x*
⟨*proof*⟩

lemma *complex-i-not-zero* [*simp*]: i $\neq$ *0*
⟨*proof*⟩

lemma *complex-i-not-one* [*simp*]: i $\neq$ *1*
⟨*proof*⟩

lemma *complex-i-not-numeral* [*simp*]: i $\neq$ *numeral w*
⟨*proof*⟩

lemma *complex-i-not-neg-numeral* [*simp*]: i $\neq$ − *numeral w*
⟨*proof*⟩

lemma *complex-split-polar*: $\exists$ *r a*. *z* = *complex-of-real r* ∗ (*cos a* + i ∗ *sin a*)
⟨*proof*⟩

lemma *i-even-power* [*simp*]: i ^ (*n* ∗ *2*) = (−*1*) ^ *n*
⟨*proof*⟩

lemma *Re-i-times* [*simp*]: *Re* (i ∗ *z*) = − *Im z*
⟨*proof*⟩

lemma *Im-i-times* [*simp*]: *Im* (i ∗ *z*) = *Re z*
⟨*proof*⟩

lemma *i-times-eq-iff*: i ∗ *w* = *z* ⟷ *w* = − (i ∗ *z*)
⟨*proof*⟩

lemma *divide-numeral-i* [*simp*]: *z* / (*numeral n* ∗ i) = − (i ∗ *z*) / *numeral n*
⟨*proof*⟩

lemma *imaginary-eq-real-iff* [*simp*]:
assumes *y* ∈ *Reals x* ∈ *Reals*
shows i ∗ *y* = *x* ⟷ *x=0* ∧ *y=0*
⟨*proof*⟩

lemma *real-eq-imaginary-iff* [*simp*]:
assumes *y* ∈ *Reals x* ∈ *Reals*
shows *x* = i ∗ *y* ⟷ *x=0* ∧ *y=0*
⟨*proof*⟩

106.6 Vector Norm

instantiation *complex* :: *real-normed-field*
begin

definition *norm z* = *sqrt* ((*Re z*)^2 + (*Im z*)^2)

abbreviation *cmod* :: *complex* ⇒ *real*
where *cmod* ≡ *norm*

definition *complex-sgn-def*: $sgn\ x = x\ /_R\ cmod\ x$

definition *dist-complex-def*: $dist\ x\ y = cmod\ (x - y)$

definition *uniformity-complex-def* [*code del*]:
$(uniformity :: (complex \times complex)\ filter) = (INF\ e{:}\{0\ <..\}.\ principal\ \{(x,\ y).\ dist\ x\ y < e\})$

definition *open-complex-def* [*code del*]:
$open\ (U :: complex\ set) \longleftrightarrow (\forall x{\in}U.\ eventually\ (\lambda(x',\ y).\ x' = x \longrightarrow y \in U)\ uniformity)$

instance
⟨*proof*⟩

end

declare *uniformity-Abort*[**where** $'a = complex$, *code*]

lemma *norm-ii* [*simp*]: $norm\ \mathrm{i} = 1$
⟨*proof*⟩

lemma *cmod-unit-one*: $cmod\ (cos\ a + \mathrm{i} * sin\ a) = 1$
⟨*proof*⟩

lemma *cmod-complex-polar*: $cmod\ (r * (cos\ a + \mathrm{i} * sin\ a)) = |r|$
⟨*proof*⟩

lemma *complex-Re-le-cmod*: $Re\ x \leq cmod\ x$
⟨*proof*⟩

lemma *complex-mod-minus-le-complex-mod*: $-\ cmod\ x \leq cmod\ x$
⟨*proof*⟩

lemma *complex-mod-triangle-ineq2*: $cmod\ (b + a) - cmod\ b \leq cmod\ a$
⟨*proof*⟩

lemma *abs-Re-le-cmod*: $|Re\ x| \leq cmod\ x$
⟨*proof*⟩

lemma *abs-Im-le-cmod*: $|Im\ x| \leq cmod\ x$
⟨*proof*⟩

lemma *cmod-le*: $cmod\ z \leq |Re\ z| + |Im\ z|$
⟨*proof*⟩

lemma *cmod-eq-Re*: $Im\ z = 0 \Longrightarrow cmod\ z = |Re\ z|$
⟨*proof*⟩

lemma *cmod-eq-Im*: $Re\ z = 0 \Longrightarrow cmod\ z = |Im\ z|$
⟨*proof*⟩

lemma *cmod-power2*: $(cmod\ z)^2 = (Re\ z)^2 + (Im\ z)^2$
⟨*proof*⟩

lemma *cmod-plus-Re-le-0-iff*: $cmod\ z + Re\ z \leq 0 \longleftrightarrow Re\ z = -\ cmod\ z$
⟨*proof*⟩

lemma *cmod-Re-le-iff*: $Im\ x = Im\ y \Longrightarrow cmod\ x \leq cmod\ y \longleftrightarrow |Re\ x| \leq |Re\ y|$
⟨*proof*⟩

lemma *cmod-Im-le-iff*: $Re\ x = Re\ y \Longrightarrow cmod\ x \leq cmod\ y \longleftrightarrow |Im\ x| \leq |Im\ y|$
⟨*proof*⟩

lemma *Im-eq-0*: $|Re\ z| = cmod\ z \Longrightarrow Im\ z = 0$
⟨*proof*⟩

lemma *abs-sqrt-wlog*: $(\bigwedge x.\ x \geq 0 \Longrightarrow P\ x\ (x^2)) \Longrightarrow P\ |x|\ (x^2)$
for x::'*a*::*linordered-idom*
⟨*proof*⟩

lemma *complex-abs-le-norm*: $|Re\ z| + |Im\ z| \leq sqrt\ 2 * norm\ z$
⟨*proof*⟩

lemma *complex-unit-circle*: $z \neq 0 \Longrightarrow (Re\ z\ /\ cmod\ z)^2 + (Im\ z\ /\ cmod\ z)^2 = 1$
⟨*proof*⟩

Properties of complex signum.

lemma *sgn-eq*: $sgn\ z = z\ /\ complex\text{-}of\text{-}real\ (cmod\ z)$
⟨*proof*⟩

lemma *Re-sgn* [*simp*]: $Re(sgn\ z) = Re(z)/cmod\ z$
⟨*proof*⟩

lemma *Im-sgn* [*simp*]: $Im(sgn\ z) = Im(z)/cmod\ z$
⟨*proof*⟩

106.7 Absolute value

instantiation *complex* :: *field-abs-sgn*
begin

definition *abs-complex* :: *complex* ⇒ *complex*
where *abs-complex* = *of-real* ∘ *norm*

instance
⟨*proof*⟩

end

106.8 Completeness of the Complexes

lemma *bounded-linear-Re*: *bounded-linear Re*
⟨*proof*⟩

lemma *bounded-linear-Im*: *bounded-linear Im*
⟨*proof*⟩

lemmas *Cauchy-Re* = *bounded-linear.Cauchy* [*OF bounded-linear-Re*]
lemmas *Cauchy-Im* = *bounded-linear.Cauchy* [*OF bounded-linear-Im*]
lemmas *tendsto-Re* [*tendsto-intros*] = *bounded-linear.tendsto* [*OF bounded-linear-Re*]
lemmas *tendsto-Im* [*tendsto-intros*] = *bounded-linear.tendsto* [*OF bounded-linear-Im*]
lemmas *isCont-Re* [*simp*] = *bounded-linear.isCont* [*OF bounded-linear-Re*]
lemmas *isCont-Im* [*simp*] = *bounded-linear.isCont* [*OF bounded-linear-Im*]
lemmas *continuous-Re* [*simp*] = *bounded-linear.continuous* [*OF bounded-linear-Re*]
lemmas *continuous-Im* [*simp*] = *bounded-linear.continuous* [*OF bounded-linear-Im*]
lemmas *continuous-on-Re* [*continuous-intros*] = *bounded-linear.continuous-on*[*OF bounded-linear-Re*]
lemmas *continuous-on-Im* [*continuous-intros*] = *bounded-linear.continuous-on*[*OF bounded-linear-Im*]
lemmas *has-derivative-Re* [*derivative-intros*] = *bounded-linear.has-derivative*[*OF bounded-linear-Re*]
lemmas *has-derivative-Im* [*derivative-intros*] = *bounded-linear.has-derivative*[*OF bounded-linear-Im*]
lemmas *sums-Re* = *bounded-linear.sums* [*OF bounded-linear-Re*]
lemmas *sums-Im* = *bounded-linear.sums* [*OF bounded-linear-Im*]

lemma *tendsto-Complex* [*tendsto-intros*]:
$(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow ((\lambda x.\ Complex\ (f\ x)\ (g\ x)) \longrightarrow Complex\ a\ b)\ F$
⟨*proof*⟩

lemma *tendsto-complex-iff*:
$(f \longrightarrow x)\ F \longleftrightarrow (((\lambda x.\ Re\ (f\ x)) \longrightarrow Re\ x)\ F \wedge ((\lambda x.\ Im\ (f\ x)) \longrightarrow Im\ x)\ F)$
⟨*proof*⟩

lemma *continuous-complex-iff*:
$continuous\ F\ f \longleftrightarrow continuous\ F\ (\lambda x.\ Re\ (f\ x)) \wedge continuous\ F\ (\lambda x.\ Im\ (f\ x))$
⟨*proof*⟩

lemma *continuous-on-of-real-o-iff* [*simp*]:
$continuous\text{-}on\ S\ (\lambda x.\ complex\text{-}of\text{-}real\ (g\ x)) = continuous\text{-}on\ S\ g$
⟨*proof*⟩

lemma *continuous-on-of-real-id* [*simp*]:

continuous-on S (*of-real* :: *real* $\Rightarrow$ *'a::real-normed-algebra-1*)
⟨*proof*⟩

lemma *has-vector-derivative-complex-iff*: (*f has-vector-derivative x*) *F* $\longleftrightarrow$
((λ*x*. *Re* (*f x*)) *has-field-derivative* (*Re x*)) *F* $\wedge$
((λ*x*. *Im* (*f x*)) *has-field-derivative* (*Im x*)) *F*
⟨*proof*⟩

lemma *has-field-derivative-Re*[*derivative-intros*]:
(*f has-vector-derivative D*) *F* $\Longrightarrow$ ((λ*x*. *Re* (*f x*)) *has-field-derivative* (*Re D*)) *F*
⟨*proof*⟩

lemma *has-field-derivative-Im*[*derivative-intros*]:
(*f has-vector-derivative D*) *F* $\Longrightarrow$ ((λ*x*. *Im* (*f x*)) *has-field-derivative* (*Im D*)) *F*
⟨*proof*⟩

instance *complex* :: *banach*
⟨*proof*⟩

declare *DERIV-power*[**where** *'a=complex*, *unfolded of-nat-def*[*symmetric*], *derivative-intros*]

106.9 Complex Conjugation

primcorec *cnj* :: *complex* $\Rightarrow$ *complex*
where
Re (*cnj z*) = *Re z*
| *Im* (*cnj z*) = − *Im z*

lemma *complex-cnj-cancel-iff* [*simp*]: *cnj x* = *cnj y* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

lemma *complex-cnj-cnj* [*simp*]: *cnj* (*cnj z*) = *z*
⟨*proof*⟩

lemma *complex-cnj-zero* [*simp*]: *cnj 0* = *0*
⟨*proof*⟩

lemma *complex-cnj-zero-iff* [*iff*]: *cnj z* = *0* $\longleftrightarrow$ *z* = *0*
⟨*proof*⟩

lemma *complex-cnj-add* [*simp*]: *cnj* (*x* + *y*) = *cnj x* + *cnj y*
⟨*proof*⟩

lemma *cnj-sum* [*simp*]: *cnj* (*sum f s*) = ($\sum$ *x*∈*s*. *cnj* (*f x*))
⟨*proof*⟩

lemma *complex-cnj-diff* [*simp*]: *cnj* (*x* − *y*) = *cnj x* − *cnj y*
⟨*proof*⟩

lemma *complex-cnj-minus* [*simp*]: *cnj* (− *x*) = − *cnj x*
⟨*proof*⟩

lemma *complex-cnj-one* [*simp*]: *cnj 1* = *1*
⟨*proof*⟩

lemma *complex-cnj-mult* [*simp*]: *cnj* (*x* ∗ *y*) = *cnj x* ∗ *cnj y*
⟨*proof*⟩

lemma *cnj-prod* [*simp*]: *cnj* (*prod f s*) = ($\prod$ *x*∈*s*. *cnj* (*f x*))
⟨*proof*⟩

lemma *complex-cnj-inverse* [*simp*]: *cnj* (*inverse x*) = *inverse* (*cnj x*)
⟨*proof*⟩

lemma *complex-cnj-divide* [*simp*]: *cnj* (*x* / *y*) = *cnj x* / *cnj y*
⟨*proof*⟩

lemma *complex-cnj-power* [*simp*]: *cnj* (*x* ^ *n*) = *cnj x* ^ *n*
⟨*proof*⟩

lemma *complex-cnj-of-nat* [*simp*]: *cnj* (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *complex-cnj-of-int* [*simp*]: *cnj* (*of-int z*) = *of-int z*
⟨*proof*⟩

lemma *complex-cnj-numeral* [*simp*]: *cnj* (*numeral w*) = *numeral w*
⟨*proof*⟩

lemma *complex-cnj-neg-numeral* [*simp*]: *cnj* (− *numeral w*) = − *numeral w*
⟨*proof*⟩

lemma *complex-cnj-scaleR* [*simp*]: *cnj* (*scaleR r x*) = *scaleR r* (*cnj x*)
⟨*proof*⟩

lemma *complex-mod-cnj* [*simp*]: *cmod* (*cnj z*) = *cmod z*
⟨*proof*⟩

lemma *complex-cnj-complex-of-real* [*simp*]: *cnj* (*of-real x*) = *of-real x*
⟨*proof*⟩

lemma *complex-cnj-i* [*simp*]: *cnj* i = − i
⟨*proof*⟩

lemma *complex-add-cnj*: *z* + *cnj z* = *complex-of-real* (*2* ∗ *Re z*)
⟨*proof*⟩

lemma *complex-diff-cnj*: *z* − *cnj z* = *complex-of-real* (*2* ∗ *Im z*) ∗ i

⟨*proof*⟩

lemma *complex-mult-cnj*: $z * cnj\ z = complex\text{-}of\text{-}real\ ((Re\ z)^2 + (Im\ z)^2)$
⟨*proof*⟩

lemma *complex-mod-mult-cnj*: $cmod\ (z * cnj\ z) = (cmod\ z)^2$
⟨*proof*⟩

lemma *complex-mod-sqrt-Re-mult-cnj*: $cmod\ z = sqrt\ (Re\ (z * cnj\ z))$
⟨*proof*⟩

lemma *complex-In-mult-cnj-zero* [*simp*]: $Im\ (z * cnj\ z) = 0$
⟨*proof*⟩

lemma *complex-cnj-fact* [*simp*]: $cnj\ (fact\ n) = fact\ n$
⟨*proof*⟩

lemma *complex-cnj-pochhammer* [*simp*]: $cnj\ (pochhammer\ z\ n) = pochhammer\ (cnj\ z)\ n$
⟨*proof*⟩

lemma *bounded-linear-cnj*: *bounded-linear cnj*
⟨*proof*⟩

lemmas *tendsto-cnj* [*tendsto-intros*] = *bounded-linear.tendsto* [*OF bounded-linear-cnj*]
and *isCont-cnj* [*simp*] = *bounded-linear.isCont* [*OF bounded-linear-cnj*]
and *continuous-cnj* [*simp*, *continuous-intros*] = *bounded-linear.continuous* [*OF bounded-linear-cnj*]
and *continuous-on-cnj* [*simp*, *continuous-intros*] = *bounded-linear.continuous-on* [*OF bounded-linear-cnj*]
and *has-derivative-cnj* [*simp*, *derivative-intros*] = *bounded-linear.has-derivative* [*OF bounded-linear-cnj*]

lemma *lim-cnj*: $((\lambda x.\ cnj(f\ x)) \longrightarrow cnj\ l)\ F \longleftrightarrow (f \longrightarrow l)\ F$
⟨*proof*⟩

lemma *sums-cnj*: $((\lambda x.\ cnj(f\ x))\ sums\ cnj\ l) \longleftrightarrow (f\ sums\ l)$
⟨*proof*⟩

106.10 Basic Lemmas

lemma *complex-eq-0*: $z{=}0 \longleftrightarrow (Re\ z)^2 + (Im\ z)^2 = 0$
⟨*proof*⟩

lemma *complex-neq-0*: $z{\neq}0 \longleftrightarrow (Re\ z)^2 + (Im\ z)^2 > 0$
⟨*proof*⟩

lemma *complex-norm-square*: $of\text{-}real\ ((norm\ z)^2) = z * cnj\ z$
⟨*proof*⟩

lemma *complex-div-cnj*: $a\ /\ b = (a * cnj\ b)\ /\ (norm\ b)^2$
⟨*proof*⟩

lemma *Re-complex-div-eq-0*: $Re\ (a\ /\ b) = 0 \longleftrightarrow Re\ (a * cnj\ b) = 0$
⟨*proof*⟩

lemma *Im-complex-div-eq-0*: $Im\ (a\ /\ b) = 0 \longleftrightarrow Im\ (a * cnj\ b) = 0$
⟨*proof*⟩

lemma *complex-div-gt-0*: $(Re\ (a\ /\ b) > 0 \longleftrightarrow Re\ (a * cnj\ b) > 0) \wedge (Im\ (a\ /\ b) > 0 \longleftrightarrow Im\ (a * cnj\ b) > 0)$
⟨*proof*⟩

lemma *Re-complex-div-gt-0*: $Re\ (a\ /\ b) > 0 \longleftrightarrow Re\ (a * cnj\ b) > 0$
and *Im-complex-div-gt-0*: $Im\ (a\ /\ b) > 0 \longleftrightarrow Im\ (a * cnj\ b) > 0$
⟨*proof*⟩

lemma *Re-complex-div-ge-0*: $Re\ (a\ /\ b) \geq 0 \longleftrightarrow Re\ (a * cnj\ b) \geq 0$
⟨*proof*⟩

lemma *Im-complex-div-ge-0*: $Im\ (a\ /\ b) \geq 0 \longleftrightarrow Im\ (a * cnj\ b) \geq 0$
⟨*proof*⟩

lemma *Re-complex-div-lt-0*: $Re\ (a\ /\ b) < 0 \longleftrightarrow Re\ (a * cnj\ b) < 0$
⟨*proof*⟩

lemma *Im-complex-div-lt-0*: $Im\ (a\ /\ b) < 0 \longleftrightarrow Im\ (a * cnj\ b) < 0$
⟨*proof*⟩

lemma *Re-complex-div-le-0*: $Re\ (a\ /\ b) \leq 0 \longleftrightarrow Re\ (a * cnj\ b) \leq 0$
⟨*proof*⟩

lemma *Im-complex-div-le-0*: $Im\ (a\ /\ b) \leq 0 \longleftrightarrow Im\ (a * cnj\ b) \leq 0$
⟨*proof*⟩

lemma *Re-divide-of-real* [*simp*]: $Re\ (z\ /\ of\text{-}real\ r) = Re\ z\ /\ r$
⟨*proof*⟩

lemma *Im-divide-of-real* [*simp*]: $Im\ (z\ /\ of\text{-}real\ r) = Im\ z\ /\ r$
⟨*proof*⟩

lemma *Re-divide-Reals* [*simp*]: $r \in \mathbb{R} \Longrightarrow Re\ (z\ /\ r) = Re\ z\ /\ Re\ r$
⟨*proof*⟩

lemma *Im-divide-Reals* [*simp*]: $r \in \mathbb{R} \Longrightarrow Im\ (z\ /\ r) = Im\ z\ /\ Re\ r$
⟨*proof*⟩

lemma *Re-sum*[*simp*]: $Re\ (sum\ f\ s) = (\sum x \in s.\ Re\ (f\ x))$

⟨*proof*⟩

lemma *Im-sum*[*simp*]: $Im\ (sum\ f\ s) = (\sum x{\in}s.\ Im(f\ x))$
⟨*proof*⟩

lemma *sums-complex-iff*: $f\ sums\ x \longleftrightarrow ((\lambda x.\ Re\ (f\ x))\ sums\ Re\ x) \wedge ((\lambda x.\ Im\ (f\ x))\ sums\ Im\ x)$
⟨*proof*⟩

lemma *summable-complex-iff*: $summable\ f \longleftrightarrow summable\ (\lambda x.\ Re\ (f\ x)) \wedge\ summable\ (\lambda x.\ Im\ (f\ x))$
⟨*proof*⟩

lemma *summable-complex-of-real* [*simp*]: $summable\ (\lambda n.\ complex\text{-}of\text{-}real\ (f\ n)) \longleftrightarrow summable\ f$
⟨*proof*⟩

lemma *summable-Re*: $summable\ f \Longrightarrow summable\ (\lambda x.\ Re\ (f\ x))$
⟨*proof*⟩

lemma *summable-Im*: $summable\ f \Longrightarrow summable\ (\lambda x.\ Im\ (f\ x))$
⟨*proof*⟩

lemma *complex-is-Nat-iff*: $z \in \mathbb{N} \longleftrightarrow Im\ z = 0 \wedge (\exists\, i.\ Re\ z = of\text{-}nat\ i)$
⟨*proof*⟩

lemma *complex-is-Int-iff*: $z \in \mathbb{Z} \longleftrightarrow Im\ z = 0 \wedge (\exists\, i.\ Re\ z = of\text{-}int\ i)$
⟨*proof*⟩

lemma *complex-is-Real-iff*: $z \in \mathbb{R} \longleftrightarrow Im\ z = 0$
⟨*proof*⟩

lemma *Reals-cnj-iff*: $z \in \mathbb{R} \longleftrightarrow cnj\ z = z$
⟨*proof*⟩

lemma *in-Reals-norm*: $z \in \mathbb{R} \Longrightarrow norm\ z = |Re\ z|$
⟨*proof*⟩

lemma *Re-Reals-divide*: $r \in \mathbb{R} \Longrightarrow Re\ (r\ /\ z) = Re\ r * Re\ z\ /\ (norm\ z)^2$
⟨*proof*⟩

lemma *Im-Reals-divide*: $r \in \mathbb{R} \Longrightarrow Im\ (r\ /\ z) = -Re\ r * Im\ z\ /\ (norm\ z)^2$
⟨*proof*⟩

lemma *series-comparison-complex*:
fixes $f{::}\ nat \Rightarrow {'}a{::}banach$
assumes *sg*: *summable g*
and $\bigwedge n.\ g\ n \in \mathbb{R}\ \bigwedge n.\ Re\ (g\ n) \geq 0$
and *fg*: $\bigwedge n.\ n \geq N \Longrightarrow norm(f\ n) \leq norm(g\ n)$

shows *summable f*
⟨*proof*⟩

106.11 Polar Form for Complex Numbers

lemma *complex-unimodular-polar*:
assumes *norm z = 1*
obtains *t* **where** $0 \leq t$ $t < 2 * pi$ $z = Complex\ (cos\ t)\ (sin\ t)$
⟨*proof*⟩

106.11.1 $\cos\theta + i\sin\theta$

primcorec *cis* :: *real* ⇒ *complex*
where
Re (*cis a*) = *cos a*
| *Im* (*cis a*) = *sin a*

lemma *cis-zero* [*simp*]: *cis 0 = 1*
⟨*proof*⟩

lemma *norm-cis* [*simp*]: *norm* (*cis a*) = *1*
⟨*proof*⟩

lemma *sgn-cis* [*simp*]: *sgn* (*cis a*) = *cis a*
⟨*proof*⟩

lemma *cis-neq-zero* [*simp*]: *cis a* ≠ *0*
⟨*proof*⟩

lemma *cis-mult*: *cis a* ∗ *cis b* = *cis* (*a* + *b*)
⟨*proof*⟩

lemma *DeMoivre*: (*cis a*) ˆ *n* = *cis* (*real n* ∗ *a*)
⟨*proof*⟩

lemma *cis-inverse* [*simp*]: *inverse* (*cis a*) = *cis* (− *a*)
⟨*proof*⟩

lemma *cis-divide*: *cis a* / *cis b* = *cis* (*a* − *b*)
⟨*proof*⟩

lemma *cos-n-Re-cis-pow-n*: *cos* (*real n* ∗ *a*) = *Re* (*cis a* ˆ *n*)
⟨*proof*⟩

lemma *sin-n-Im-cis-pow-n*: *sin* (*real n* ∗ *a*) = *Im* (*cis a* ˆ *n*)
⟨*proof*⟩

lemma *cis-pi*: *cis pi* = −*1*
⟨*proof*⟩

106.11.2 $r(\cos\theta + i\sin\theta)$

definition *rcis* :: *real* ⇒ *real* ⇒ *complex*
where *rcis r a* = *complex-of-real r* ∗ *cis a*

lemma *Re-rcis* [*simp*]: *Re*(*rcis r a*) = *r* ∗ *cos a*
⟨*proof*⟩

lemma *Im-rcis* [*simp*]: *Im*(*rcis r a*) = *r* ∗ *sin a*
⟨*proof*⟩

lemma *rcis-Ex*: ∃ *r a*. *z* = *rcis r a*
⟨*proof*⟩

lemma *complex-mod-rcis* [*simp*]: *cmod* (*rcis r a*) = |*r*|
⟨*proof*⟩

lemma *cis-rcis-eq*: *cis a* = *rcis 1 a*
⟨*proof*⟩

lemma *rcis-mult*: *rcis r1 a* ∗ *rcis r2 b* = *rcis* (*r1* ∗ *r2*) (*a* + *b*)
⟨*proof*⟩

lemma *rcis-zero-mod* [*simp*]: *rcis 0 a* = *0*
⟨*proof*⟩

lemma *rcis-zero-arg* [*simp*]: *rcis r 0* = *complex-of-real r*
⟨*proof*⟩

lemma *rcis-eq-zero-iff* [*simp*]: *rcis r a* = *0* ⟷ *r* = *0*
⟨*proof*⟩

lemma *DeMoivre2*: (*rcis r a*) ^ *n* = *rcis* (*r* ^ *n*) (*real n* ∗ *a*)
⟨*proof*⟩

lemma *rcis-inverse*: *inverse*(*rcis r a*) = *rcis* (*1* / *r*) (− *a*)
⟨*proof*⟩

lemma *rcis-divide*: *rcis r1 a* / *rcis r2 b* = *rcis* (*r1* / *r2*) (*a* − *b*)
⟨*proof*⟩

106.11.3 Complex exponential

lemma *cis-conv-exp*: *cis b* = *exp* (i ∗ *b*)
⟨*proof*⟩

lemma *exp-eq-polar*: *exp z* = *exp* (*Re z*) ∗ *cis* (*Im z*)
⟨*proof*⟩

lemma *Re-exp*: *Re* (*exp z*) = *exp* (*Re z*) ∗ *cos* (*Im z*)

$\langle proof \rangle$

lemma *Im-exp*: $Im\ (exp\ z) = exp\ (Re\ z) * sin\ (Im\ z)$
$\langle proof \rangle$

lemma *norm-cos-sin* [*simp*]: $norm\ (Complex\ (cos\ t)\ (sin\ t)) = 1$
$\langle proof \rangle$

lemma *norm-exp-eq-Re* [*simp*]: $norm\ (exp\ z) = exp\ (Re\ z)$
$\langle proof \rangle$

lemma *complex-exp-exists*: $\exists\, a\ r.\ z = complex\text{-}of\text{-}real\ r * exp\ a$
$\langle proof \rangle$

lemma *exp-pi-i* [*simp*]: $exp\ (of\text{-}real\ pi * \mathrm{i}) = -1$
$\langle proof \rangle$

lemma *exp-pi-i'* [*simp*]: $exp\ (\mathrm{i} * of\text{-}real\ pi) = -1$
$\langle proof \rangle$

lemma *exp-two-pi-i* [*simp*]: $exp\ (2 * of\text{-}real\ pi * \mathrm{i}) = 1$
$\langle proof \rangle$

lemma *exp-two-pi-i'* [*simp*]: $exp\ (\mathrm{i} * (of\text{-}real\ pi * 2)) = 1$
$\langle proof \rangle$

106.11.4 Complex argument

definition $arg :: complex \Rightarrow real$
where $arg\ z = (if\ z = 0\ then\ 0\ else\ (SOME\ a.\ sgn\ z = cis\ a \wedge -\ pi < a \wedge a \leq pi))$

lemma *arg-zero*: $arg\ 0 = 0$
$\langle proof \rangle$

lemma *arg-unique*:
assumes $sgn\ z = cis\ x$ **and** $-pi < x$ **and** $x \leq pi$
shows $arg\ z = x$
$\langle proof \rangle$

lemma *arg-correct*:
assumes $z \neq 0$
shows $sgn\ z = cis\ (arg\ z) \wedge -pi < arg\ z \wedge arg\ z \leq pi$
$\langle proof \rangle$

lemma *arg-bounded*: $-\ pi < arg\ z \wedge arg\ z \leq pi$
$\langle proof \rangle$

lemma *cis-arg*: $z \neq 0 \Longrightarrow cis\ (arg\ z) = sgn\ z$

⟨proof⟩

lemma *rcis-cmod-arg*: *rcis* (*cmod z*) (*arg z*) = *z*
⟨proof⟩

lemma *cos-arg-i-mult-zero* [*simp*]: $y \neq 0 \Longrightarrow Re\ y = 0 \Longrightarrow cos\ (arg\ y) = 0$
⟨proof⟩

106.12 Square root of complex numbers

primcorec *csqrt* :: *complex* ⇒ *complex*
where
Re (*csqrt z*) = *sqrt* ((*cmod z* + *Re z*) / *2*)
| *Im* (*csqrt z*) = (*if Im z* = *0 then 1 else sgn* (*Im z*)) ∗ *sqrt* ((*cmod z* − *Re z*) / *2*)

lemma *csqrt-of-real-nonneg* [*simp*]: $Im\ x = 0 \Longrightarrow Re\ x \geq 0 \Longrightarrow csqrt\ x = sqrt\ (Re\ x)$
⟨proof⟩

lemma *csqrt-of-real-nonpos* [*simp*]: $Im\ x = 0 \Longrightarrow Re\ x \leq 0 \Longrightarrow csqrt\ x = \mathrm{i} * sqrt\ |Re\ x|$
⟨proof⟩

lemma *of-real-sqrt*: $x \geq 0 \Longrightarrow of\text{-}real\ (sqrt\ x) = csqrt\ (of\text{-}real\ x)$
⟨proof⟩

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

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

lemma *csqrt-ii* [*simp*]: *csqrt* i = (*1* + i) / *sqrt 2*
⟨proof⟩

lemma *power2-csqrt*[*simp*,*algebra*]: $(csqrt\ z)^2 = z$
⟨proof⟩

lemma *csqrt-eq-0* [*simp*]: $csqrt\ z = 0 \longleftrightarrow z = 0$
⟨proof⟩

lemma *csqrt-eq-1* [*simp*]: $csqrt\ z = 1 \longleftrightarrow z = 1$
⟨proof⟩

lemma *csqrt-principal*: $0 < Re\ (csqrt\ z) \lor Re\ (csqrt\ z) = 0 \land 0 \leq Im\ (csqrt\ z)$
⟨proof⟩

lemma *Re-csqrt*: $0 \leq Re\ (csqrt\ z)$

⟨*proof*⟩

lemma *csqrt-square*:
assumes $0 < Re\ b \lor (Re\ b = 0 \land 0 \le Im\ b)$
shows $csqrt\ (b \hat{} 2) = b$
⟨*proof*⟩

lemma *csqrt-unique*: $w^2 = z \Longrightarrow 0 < Re\ w \lor Re\ w = 0 \land 0 \le Im\ w \Longrightarrow csqrt\ z = w$
⟨*proof*⟩

lemma *csqrt-minus* [*simp*]:
assumes $Im\ x < 0 \lor (Im\ x = 0 \land 0 \le Re\ x)$
shows $csqrt\ (-\ x) = \mathrm{i} * csqrt\ x$
⟨*proof*⟩

Legacy theorem names

lemmas *expand-complex-eq* = *complex-eq-iff*
lemmas *complex-Re-Im-cancel-iff* = *complex-eq-iff*
lemmas *complex-equality* = *complex-eqI*
lemmas *cmod-def* = *norm-complex-def*
lemmas *complex-norm-def* = *norm-complex-def*
lemmas *complex-divide-def* = *divide-complex-def*

lemma *legacy-Complex-simps*:
shows *Complex-eq-0*: $Complex\ a\ b = 0 \longleftrightarrow a = 0 \land b = 0$
and *complex-add*: $Complex\ a\ b + Complex\ c\ d = Complex\ (a + c)\ (b + d)$
and *complex-minus*: $-\ (Complex\ a\ b) = Complex\ (-\ a)\ (-\ b)$
and *complex-diff*: $Complex\ a\ b - Complex\ c\ d = Complex\ (a - c)\ (b - d)$
and *Complex-eq-1*: $Complex\ a\ b = 1 \longleftrightarrow a = 1 \land b = 0$
and *Complex-eq-neg-1*: $Complex\ a\ b = -\ 1 \longleftrightarrow a = -\ 1 \land b = 0$
and *complex-mult*: $Complex\ a\ b * Complex\ c\ d = Complex\ (a * c - b * d)\ (a * d + b * c)$
and *complex-inverse*: $inverse\ (Complex\ a\ b) = Complex\ (a\ /\ (a^2 + b^2))\ (-\ b\ /\ (a^2 + b^2))$
and *Complex-eq-numeral*: $Complex\ a\ b = numeral\ w \longleftrightarrow a = numeral\ w \land b = 0$
and *Complex-eq-neg-numeral*: $Complex\ a\ b = -\ numeral\ w \longleftrightarrow a = -\ numeral\ w \land b = 0$
and *complex-scaleR*: $scaleR\ r\ (Complex\ a\ b) = Complex\ (r * a)\ (r * b)$
and *Complex-eq-i*: $Complex\ x\ y = \mathrm{i} \longleftrightarrow x = 0 \land y = 1$
and *i-mult-Complex*: $\mathrm{i} * Complex\ a\ b = Complex\ (-\ b)\ a$
and *Complex-mult-i*: $Complex\ a\ b * \mathrm{i} = Complex\ (-\ b)\ a$
and *i-complex-of-real*: $\mathrm{i} * complex\text{-}of\text{-}real\ r = Complex\ 0\ r$
and *complex-of-real-i*: $complex\text{-}of\text{-}real\ r * \mathrm{i} = Complex\ 0\ r$
and *Complex-add-complex-of-real*: $Complex\ x\ y + complex\text{-}of\text{-}real\ r = Complex\ (x{+}r)\ y$
and *complex-of-real-add-Complex*: $complex\text{-}of\text{-}real\ r + Complex\ x\ y = Complex\ (r{+}x)\ y$

and *Complex-mult-complex-of-real*: $Complex\ x\ y * complex\text{-}of\text{-}real\ r = Complex\ (x{*}r)\ (y{*}r)$
and *complex-of-real-mult-Complex*: $complex\text{-}of\text{-}real\ r * Complex\ x\ y = Complex\ (r{*}x)\ (r{*}y)$
and *complex-eq-cancel-iff2*: $(Complex\ x\ y = complex\text{-}of\text{-}real\ xa) = (x = xa \wedge y = 0)$
and *complex-cn*: $cnj\ (Complex\ a\ b) = Complex\ a\ (-\ b)$
and *Complex-sum'*: $sum\ (\lambda x.\ Complex\ (f\ x)\ 0)\ s = Complex\ (sum\ f\ s)\ 0$
and *Complex-sum*: $Complex\ (sum\ f\ s)\ 0 = sum\ (\lambda x.\ Complex\ (f\ x)\ 0)\ s$
and *complex-of-real-def*: $complex\text{-}of\text{-}real\ r = Complex\ r\ 0$
and *complex-norm*: $cmod\ (Complex\ x\ y) = sqrt\ (x^2 + y^2)$
⟨*proof*⟩

lemma *Complex-in-Reals*: $Complex\ x\ 0 \in \mathbb{R}$
⟨*proof*⟩

end

107 MacLaurin and Taylor Series

theory *MacLaurin*
imports *Transcendental*
begin

107.1 Maclaurin's Theorem with Lagrange Form of Remainder

This is a very long, messy proof even now that it's been broken down into lemmas.

lemma *Maclaurin-lemma*:
$0 < h \Longrightarrow$
$\exists B{::}real.\ f\ h = (\sum m{<}n.\ (j\ m\ /\ (fact\ m)) * (h\hat{}m)) + (B * ((h\hat{}n)\ /(fact\ n)))$
⟨*proof*⟩

lemma *eq-diff-eq'*: $x = y - z \longleftrightarrow y = x + z$
for $x\ y\ z :: real$
⟨*proof*⟩

lemma *fact-diff-Suc*: $n < Suc\ m \Longrightarrow fact\ (Suc\ m - n) = (Suc\ m - n) * fact\ (m - n)$
⟨*proof*⟩

lemma *Maclaurin-lemma2*:
fixes B
assumes *DERIV*: $\forall m\ t.\ m < n \wedge 0{\le}t \wedge t{\le}h \longrightarrow DERIV\ (diff\ m)\ t :> diff\ (Suc\ m)\ t$
and *INIT*: $n = Suc\ k$
defines $difg \equiv$

(λm *t*::*real*. *diff* m t $-$
(($\sum p<n - m$. *diff* $(m + p)$ *0* / *fact* $p * t$ ^ p) $+ B * (t$ ^ $(n - m)$ / *fact* $(n - m)$))))
(**is** *difg* $\equiv$ (λm t. *diff* m t $-$ *?difg* m t))
shows $\forall\, m\ t.\ m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow$ *DERIV* (*difg* m) t :> *difg* (*Suc* m) t
⟨*proof*⟩

lemma *Maclaurin*:
assumes *h*: $0 < h$
and *n*: $0 < n$
and *diff-0*: *diff* $0 = f$
and *diff-Suc*: $\forall\, m\ t.\ m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow$ *DERIV* (*diff* m) t :> *diff* (*Suc* m) t
shows
$\exists\, t$::*real*. $0 < t \wedge t < h \wedge$
$f\ h =$ *sum* (λm. (*diff* m *0* / *fact* m) $* h$ ^ m) {..<n} + (*diff* n t / *fact* n) $* h$ ^ n
⟨*proof*⟩

lemma *Maclaurin-objl*:
$0 < h \wedge n > 0 \wedge$ *diff* $0 = f \wedge$
($\forall\, m\ t.\ m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow$ *DERIV* (*diff* m) t :> *diff* (*Suc* m) t) $\longrightarrow$
($\exists\, t.\ 0 < t \wedge t < h \wedge f\ h =$ ($\sum m<n$. *diff* m *0* / (*fact* m) $* h$ ^ m) + *diff* n t / *fact* $n * h$ ^ n)
for n :: *nat* **and** h :: *real*
⟨*proof*⟩

lemma *Maclaurin2*:
fixes n :: *nat*
and h :: *real*
assumes *INIT1*: $0 < h$
and *INIT2*: *diff* $0 = f$
and *DERIV*: $\forall\, m\ t.\ m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow$ *DERIV* (*diff* m) t :> *diff* (*Suc* m) t
shows $\exists\, t.\ 0 < t \wedge t \leq h \wedge f\ h =$ ($\sum m<n$. *diff* m *0* / (*fact* m) $* h$ ^ m) + *diff* n t / *fact* $n * h$ ^ n
⟨*proof*⟩

lemma *Maclaurin2-objl*:
$0 < h \wedge$ *diff* $0 = f \wedge$
($\forall\, m\ t.\ m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow$ *DERIV* (*diff* m) t :> *diff* (*Suc* m) t) $\longrightarrow$
($\exists\, t.\ 0 < t \wedge t \leq h \wedge f\ h =$ ($\sum m<n$. *diff* m *0* / *fact* $m * h$ ^ m) + *diff* n t / *fact* $n * h$ ^ n)
for n :: *nat* **and** h :: *real*
⟨*proof*⟩

lemma *Maclaurin-minus*:
fixes n :: *nat* **and** h :: *real*
assumes $h < 0$ $0 < n$ *diff* $0 = f$

and *DERIV*: $\forall m\ t.\ m < n \wedge h \le t \wedge t \le 0 \longrightarrow DERIV\ (diff\ m)\ t :> diff\ (Suc\ m)\ t$
shows $\exists t.\ h < t \wedge t < 0 \wedge f\ h = (\sum m{<}n.\ diff\ m\ 0\ /\ fact\ m * h\ \hat{}\ m) + diff\ n\ t\ /\ fact\ n * h\ \hat{}\ n$
$\langle proof \rangle$

lemma *Maclaurin-minus-objl*:
fixes $n :: nat$ **and** $h :: real$
shows
$h < 0 \wedge n > 0 \wedge diff\ 0 = f \wedge$
$(\forall m\ t.\ m < n\ \&\ h \le t\ \&\ t \le 0\ {-}{-}{>}\ DERIV\ (diff\ m)\ t :> diff\ (Suc\ m)\ t) \longrightarrow$
$(\exists t.\ h < t \wedge t < 0 \wedge f\ h = (\sum m{<}n.\ diff\ m\ 0\ /\ fact\ m * h\ \hat{}\ m) + diff\ n\ t\ /\ fact\ n * h\ \hat{}\ n)$
$\langle proof \rangle$

107.2 More Convenient "Bidirectional" Version.

lemma *Maclaurin-bi-le-lemma*:
$n > 0 \Longrightarrow$
$diff\ 0\ 0 = (\sum m{<}n.\ diff\ m\ 0 * 0\ \hat{}\ m\ /\ (fact\ m)) + diff\ n\ 0 * 0\ \hat{}\ n\ /\ (fact\ n :: real)$
$\langle proof \rangle$

lemma *Maclaurin-bi-le*:
fixes $n :: nat$ **and** $x :: real$
assumes $diff\ 0 = f$
and *DERIV* : $\forall m\ t.\ m < n \wedge |t| \le |x| \longrightarrow DERIV\ (diff\ m)\ t :> diff\ (Suc\ m)\ t$
shows $\exists t.\ |t| \le |x| \wedge f\ x = (\sum m{<}n.\ diff\ m\ 0\ /\ (fact\ m) * x\ \hat{}\ m) + diff\ n\ t\ /\ (fact\ n) * x\ \hat{}\ n$
(**is** $\exists t.\ \text{-} \wedge f\ x = ?f\ x\ t$)
$\langle proof \rangle$

lemma *Maclaurin-all-lt*:
fixes $x :: real$
assumes *INIT1*: $diff\ 0 = f$
and *INIT2*: $0 < n$
and *INIT3*: $x \ne 0$
and *DERIV*: $\forall m\ x.\ DERIV\ (diff\ m)\ x :> diff(Suc\ m)\ x$
shows $\exists t.\ 0 < |t| \wedge |t| < |x| \wedge f\ x =$
$(\sum m{<}n.\ (diff\ m\ 0\ /\ fact\ m) * x\ \hat{}\ m) + (diff\ n\ t\ /\ fact\ n) * x\ \hat{}\ n$
(**is** $\exists t.\ \text{-} \wedge \text{-} \wedge f\ x = ?f\ x\ t$)
$\langle proof \rangle$

lemma *Maclaurin-all-lt-objl*:
fixes $x :: real$
shows

$diff\ 0 = f \wedge (\forall m\ x.\ DERIV\ (diff\ m)\ x :> diff\ (Suc\ m)\ x) \wedge x \neq 0 \wedge n > 0 \longrightarrow$
$(\exists t.\ 0 < |t| \wedge |t| < |x| \wedge$
$f\ x = (\sum m<n.\ (diff\ m\ 0\ /\ fact\ m) * x\ \hat{}\ m) + (diff\ n\ t\ /\ fact\ n) * x\ \hat{}\ n)$
⟨*proof*⟩

lemma *Maclaurin-zero*: $x = 0 \Longrightarrow n \neq 0 \Longrightarrow (\sum m<n.\ (diff\ m\ 0\ /\ fact\ m) * x\ \hat{}\ m) = diff\ 0\ 0$
for $x :: real$ **and** $n :: nat$
⟨*proof*⟩

lemma *Maclaurin-all-le*:
fixes $x :: real$ **and** $n :: nat$
assumes *INIT*: $diff\ 0 = f$
and *DERIV*: $\forall m\ x.\ DERIV\ (diff\ m)\ x :> diff\ (Suc\ m)\ x$
shows $\exists t.\ |t| \leq |x| \wedge f\ x = (\sum m<n.\ (diff\ m\ 0\ /\ fact\ m) * x\ \hat{}\ m) + (diff\ n\ t\ /\ fact\ n) * x\ \hat{}\ n$
(**is** $\exists t.\ \text{-} \wedge f\ x = ?f\ x\ t$)
⟨*proof*⟩

lemma *Maclaurin-all-le-objl*:
$diff\ 0 = f \wedge (\forall m\ x.\ DERIV\ (diff\ m)\ x :> diff\ (Suc\ m)\ x) \longrightarrow$
$(\exists t::real.\ |t| \leq |x| \wedge f\ x = (\sum m<n.\ (diff\ m\ 0\ /\ fact\ m) * x\ \hat{}\ m) + (diff\ n\ t\ /\ fact\ n) * x\ \hat{}\ n)$
for $x :: real$ **and** $n :: nat$
⟨*proof*⟩

107.3 Version for Exponential Function

lemma *Maclaurin-exp-lt*:
fixes $x :: real$ **and** $n :: nat$
shows
$x \neq 0 \Longrightarrow n > 0 \Longrightarrow$
$(\exists t.\ 0 < |t| \wedge |t| < |x| \wedge exp\ x = (\sum m<n.\ (x\ \hat{}\ m)\ /\ fact\ m) + (exp\ t\ /\ fact\ n) * x\ \hat{}\ n)$
⟨*proof*⟩

lemma *Maclaurin-exp-le*:
fixes $x :: real$ **and** $n :: nat$
shows $\exists t.\ |t| \leq |x| \wedge exp\ x = (\sum m<n.\ (x\ \hat{}\ m)\ /\ fact\ m) + (exp\ t\ /\ fact\ n) * x\ \hat{}\ n$
⟨*proof*⟩

corollary *exp-lower-taylor-quadratic*: $0 \leq x \Longrightarrow 1 + x + x^2\ /\ 2 \leq exp\ x$
for $x :: real$
⟨*proof*⟩

corollary *ln-2-less-1*: $ln\ 2 < (1::real)$

⟨*proof*⟩

107.4 Version for Sine Function

lemma *mod-exhaust-less-4*: *m mod 4 = 0 | m mod 4 = 1 | m mod 4 = 2 | m mod 4 = 3*
for *m* :: *nat*
⟨*proof*⟩

lemma *Suc-Suc-mult-two-diff-two* [*simp*]: $n \neq 0 \Longrightarrow Suc\ (Suc\ (2 * n - 2)) = 2 * n$
⟨*proof*⟩

lemma *lemma-Suc-Suc-4n-diff-2* [*simp*]: $n \neq 0 \Longrightarrow Suc\ (Suc\ (4 * n - 2)) = 4 * n$
⟨*proof*⟩

lemma *Suc-mult-two-diff-one* [*simp*]: $n \neq 0 \Longrightarrow Suc\ (2 * n - 1) = 2 * n$
⟨*proof*⟩

It is unclear why so many variant results are needed.

lemma *sin-expansion-lemma*: *sin (x + real (Suc m) ∗ pi / 2) = cos (x + real m ∗ pi / 2)*
⟨*proof*⟩

lemma *Maclaurin-sin-expansion2*:
$\exists t.\ |t| \leq |x| \wedge$
$sin\ x = (\sum m<n.\ sin\text{-}coeff\ m * x\ \hat{}\ m) + (sin\ (t + 1/2 * real\ n * pi) / fact\ n) * x\ \hat{}\ n$
⟨*proof*⟩

lemma *Maclaurin-sin-expansion*:
$\exists t.\ sin\ x = (\sum m<n.\ sin\text{-}coeff\ m * x\ \hat{}\ m) + (sin\ (t + 1/2 * real\ n * pi) / fact\ n) * x\ \hat{}\ n$
⟨*proof*⟩

lemma *Maclaurin-sin-expansion3*:
$n > 0 \Longrightarrow 0 < x \Longrightarrow$
$\exists t.\ 0 < t \wedge t < x \wedge$
$sin\ x = (\sum m<n.\ sin\text{-}coeff\ m * x\ \hat{}\ m) + (sin\ (t + 1/2 * real\ n * pi) / fact\ n) * x\ \hat{}\ n$
⟨*proof*⟩

lemma *Maclaurin-sin-expansion4*:
$0 < x \Longrightarrow$
$\exists t.\ 0 < t \wedge t \leq x \wedge$
$sin\ x = (\sum m<n.\ sin\text{-}coeff\ m * x\ \hat{}\ m) + (sin\ (t + 1/2 * real\ n * pi) / fact\ n) * x\ \hat{}\ n$
⟨*proof*⟩

107.5 Maclaurin Expansion for Cosine Function

lemma *sumr-cos-zero-one* [*simp*]: $(\sum m<Suc\ n.\ cos\text{-}coeff\ m * 0\ \hat{}\ m) = 1$
⟨*proof*⟩

lemma *cos-expansion-lemma*: $cos\ (x + real\ (Suc\ m) * pi\ /\ 2) = -\ sin\ (x + real\ m * pi\ /\ 2)$
⟨*proof*⟩

lemma *Maclaurin-cos-expansion*:
$\exists\, t{::}real.\ |t| \le |x| \wedge$
$cos\ x = (\sum m<n.\ cos\text{-}coeff\ m * x\ \hat{}\ m) + (cos(t + 1/2 * real\ n * pi)\ /\ fact\ n) * x\ \hat{}\ n$
⟨*proof*⟩

lemma *Maclaurin-cos-expansion2*:
$0 < x \Longrightarrow n > 0 \Longrightarrow$
$\exists\, t.\ 0 < t \wedge t < x \wedge$
$cos\ x = (\sum m<n.\ cos\text{-}coeff\ m * x\ \hat{}\ m) + (cos\ (t + 1/2 * real\ n * pi)\ /\ fact\ n) * x\ \hat{}\ n$
⟨*proof*⟩

lemma *Maclaurin-minus-cos-expansion*:
$x < 0 \Longrightarrow n > 0 \Longrightarrow$
$\exists\, t.\ x < t \wedge t < 0 \wedge$
$cos\ x = (\sum m<n.\ cos\text{-}coeff\ m * x\ \hat{}\ m) + ((cos\ (t + 1/2 * real\ n * pi)\ /\ fact\ n) * x\ \hat{}\ n)$
⟨*proof*⟩

lemma *sin-bound-lemma*: $x = y \Longrightarrow |u| \le v \Longrightarrow |(x + u) - y| \le v$
for $x\ y\ u\ v :: real$
⟨*proof*⟩

lemma *Maclaurin-sin-bound*: $|sin\ x - (\sum m<n.\ sin\text{-}coeff\ m * x\ \hat{}\ m)| \le inverse\ (fact\ n) * |x|\ \hat{}\ n$
⟨*proof*⟩

108 Taylor series

We use MacLaurin and the translation of the expansion point *c* to *0* to prove Taylor's theorem.

lemma *taylor-up*:
assumes *INIT*: $n > 0\ diff\ 0 = f$
and *DERIV*: $\forall m\ t.\ m < n \wedge a \le t \wedge t \le b \longrightarrow DERIV\ (diff\ m)\ t :> (diff$

(*Suc m*) *t*)
and *INTERV*: $a \leq c$ $c < b$
shows $\exists$ *t*::*real*. $c < t \wedge t < b \wedge$
$f\ b = (\sum m{<}n.\ (\mathit{diff}\ m\ c\ /\ \mathit{fact}\ m) * (b - c)\hat{}m) + (\mathit{diff}\ n\ t\ /\ \mathit{fact}\ n) * (b - c)\hat{}n$
⟨*proof*⟩

lemma *taylor-down*:
fixes *a* :: *real* **and** *n* :: *nat*
assumes *INIT*: $n > 0$ *diff 0* = *f*
and *DERIV*: $(\forall m\ t.\ m < n \wedge a \leq t \wedge t \leq b \longrightarrow$ *DERIV* (*diff m*) *t* :> *diff* (*Suc m*) *t*)
and *INTERV*: $a < c$ $c \leq b$
shows $\exists$ *t*. $a < t \wedge t < c \wedge$
$f\ a = (\sum m{<}n.\ (\mathit{diff}\ m\ c\ /\ \mathit{fact}\ m) * (a - c)\hat{}m) + (\mathit{diff}\ n\ t\ /\ \mathit{fact}\ n) * (a - c)\hat{}n$
⟨*proof*⟩

theorem *taylor*:
fixes *a* :: *real* **and** *n* :: *nat*
assumes *INIT*: $n > 0$ *diff 0* = *f*
and *DERIV*: $\forall m\ t.\ m < n \wedge a \leq t \wedge t \leq b \longrightarrow$ *DERIV* (*diff m*) *t* :> *diff* (*Suc m*) *t*
and *INTERV*: $a \leq c$ $c \leq b$ $a \leq x$ $x \leq b$ $x \neq c$
shows $\exists$ *t*.
(*if* $x < c$ *then* $x < t \wedge t < c$ *else* $c < t \wedge t < x$) $\wedge$
$f\ x = (\sum m{<}n.\ (\mathit{diff}\ m\ c\ /\ \mathit{fact}\ m) * (x - c)\hat{}m) + (\mathit{diff}\ n\ t\ /\ \mathit{fact}\ n) * (x - c)\hat{}n$
⟨*proof*⟩

end

109 Comprehensive Complex Theory

theory *Complex-Main*
imports
Complex
MacLaurin
begin

end

References

[1] H. Davenport. *The Higher Arithmetic*. Cambridge University Press, 1992.

[2] M. Holz, K. Steffens, and E. Weitz. *Introduction to Cardinal Arithmetic.* Birkhäuser, 1999.

[3] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.