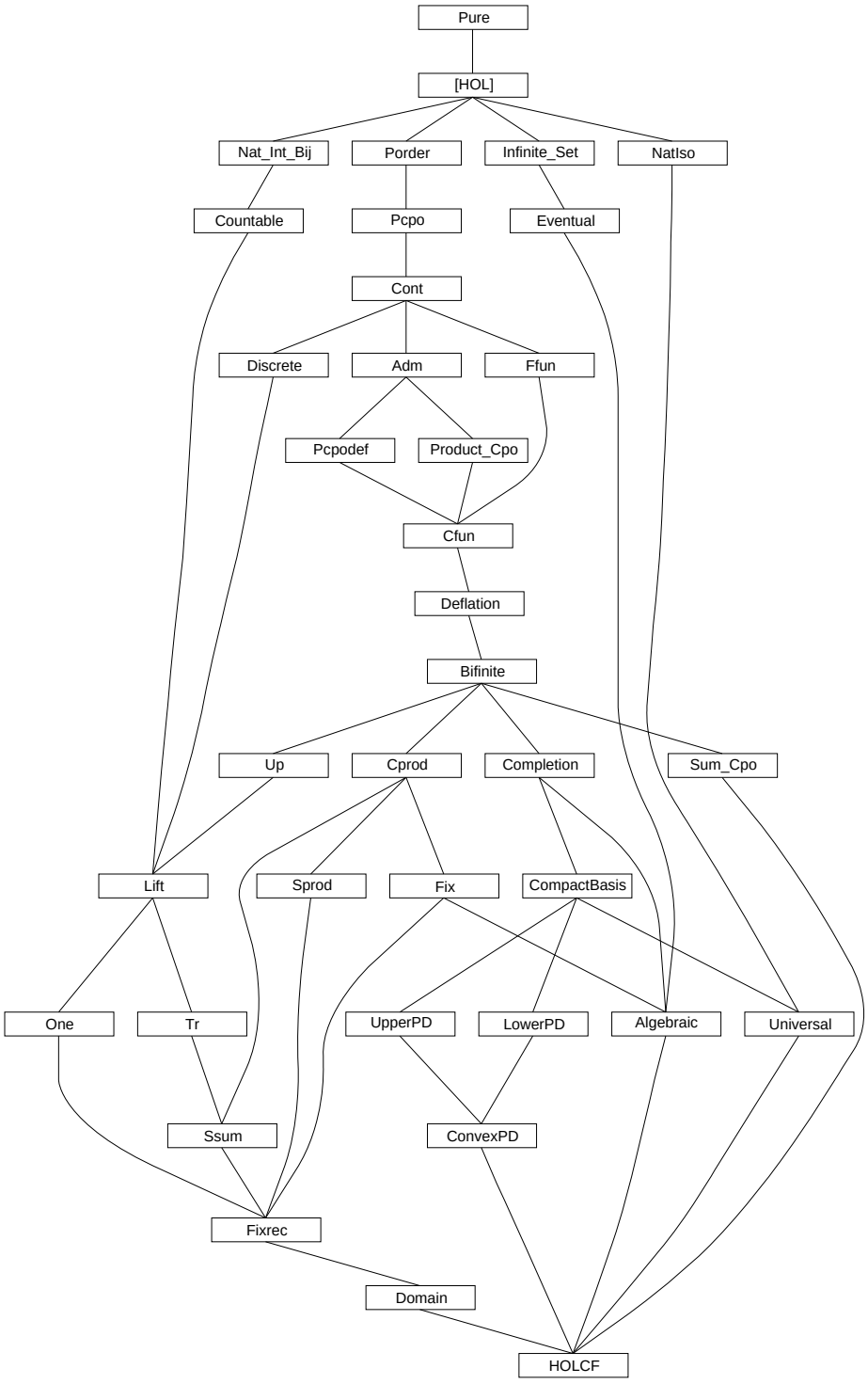
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

April 19, 2009

Contents


Pure
[HOL]
Nat_Int_Bij
Porder
Infinite_Set
NatIso
Countable
Pcpo
Eventual
Cont
Discrete
Adm
Ffun
Pcpodef
Product_Cpo
Cfun
Deflation
Bifinite
Up
Cprod
Completion
Sum_Cpo
Lift
Sprod
Fix
CompactBasis
One
Tr
UpperPD
LowerPD
Algebraic
Universal
Ssum
ConvexPD
Fixrec
Domain
HOLCF

1 Porder: Partial orders

theory *Porder*
imports *Main*
begin

1.1 Type class for partial orders

class *sq-ord* =
  fixes *sq-le* :: $'a \Rightarrow 'a \Rightarrow bool$

notation
  sq-le (**infixl** $<<$ *55*)

notation (*xsymbols*)
  sq-le (**infixl** $\sqsubseteq$ *55*)

class *po* = *sq-ord* +
  assumes *refl-less* [*iff*]: $x \sqsubseteq x$
  assumes *trans-less*: $[\![x \sqsubseteq y;\ y \sqsubseteq z]\!] \Longrightarrow x \sqsubseteq z$
  assumes *antisym-less*: $[\![x \sqsubseteq y;\ y \sqsubseteq x]\!] \Longrightarrow x = y$

minimal fixes least element

lemma *minimal2UU*[*OF allI*] : $\forall x::'a::po.\ uu \sqsubseteq x \Longrightarrow uu = (THE\ u.\ \forall y.\ u \sqsubseteq y)$
⟨*proof*⟩

the reverse law of anti-symmetry of *op* $\sqsubseteq$

lemma *antisym-less-inverse*: $(x::'a::po) = y \Longrightarrow x \sqsubseteq y \wedge y \sqsubseteq x$
⟨*proof*⟩

lemma *box-less*: $[\![(a::'a::po) \sqsubseteq b;\ c \sqsubseteq a;\ b \sqsubseteq d]\!] \Longrightarrow c \sqsubseteq d$
⟨*proof*⟩

lemma *po-eq-conv*: $((x::'a::po) = y) = (x \sqsubseteq y \wedge y \sqsubseteq x)$
⟨*proof*⟩

lemma *rev-trans-less*: $[\![(y::'a::po) \sqsubseteq z;\ x \sqsubseteq y]\!] \Longrightarrow x \sqsubseteq z$
⟨*proof*⟩

lemma *sq-ord-less-eq-trans*: $[\![a \sqsubseteq b;\ b = c]\!] \Longrightarrow a \sqsubseteq c$
⟨*proof*⟩

lemma *sq-ord-eq-less-trans*: $[\![a = b;\ b \sqsubseteq c]\!] \Longrightarrow a \sqsubseteq c$
⟨*proof*⟩

lemmas *HOLCF-trans-rules* [*trans*] =
  trans-less
  antisym-less

sq-ord-less-eq-trans
sq-ord-eq-less-trans

1.2 Upper bounds

definition
is-ub :: [*'a set*, *'a*::*po*] ⇒ *bool* (**infixl** <| *55*) **where**
(*S* <| *x*) = (∀ *y*. *y* ∈ *S* ⟶ *y* ⊑ *x*)

lemma *is-ubI*: (⋀*x*. *x* ∈ *S* ⟹ *x* ⊑ *u*) ⟹ *S* <| *u*
⟨*proof*⟩

lemma *is-ubD*: ⟦*S* <| *u*; *x* ∈ *S*⟧ ⟹ *x* ⊑ *u*
⟨*proof*⟩

lemma *ub-imageI*: (⋀*x*. *x* ∈ *S* ⟹ *f x* ⊑ *u*) ⟹ (λ*x*. *f x*) ‘ *S* <| *u*
⟨*proof*⟩

lemma *ub-imageD*: ⟦*f* ‘ *S* <| *u*; *x* ∈ *S*⟧ ⟹ *f x* ⊑ *u*
⟨*proof*⟩

lemma *ub-rangeI*: (⋀*i*. *S i* ⊑ *x*) ⟹ *range S* <| *x*
⟨*proof*⟩

lemma *ub-rangeD*: *range S* <| *x* ⟹ *S i* ⊑ *x*
⟨*proof*⟩

lemma *is-ub-empty* [*simp*]: {} <| *u*
⟨*proof*⟩

lemma *is-ub-insert* [*simp*]: (*insert x A*) <| *y* = (*x* ⊑ *y* ∧ *A* <| *y*)
⟨*proof*⟩

lemma *is-ub-upward*: ⟦*S* <| *x*; *x* ⊑ *y*⟧ ⟹ *S* <| *y*
⟨*proof*⟩

1.3 Least upper bounds

definition
is-lub :: [*'a set*, *'a*::*po*] ⇒ *bool* (**infixl** <<| *55*) **where**
(*S* <<| *x*) = (*S* <| *x* ∧ (∀ *u*. *S* <| *u* ⟶ *x* ⊑ *u*))

definition
lub :: *'a set* ⇒ *'a*::*po* **where**
lub S = (*THE x*. *S* <<| *x*)

syntax
-BLub :: [*pttrn*, *'a set*, *'b*] ⇒ *'b* ((*3LUB* -:-./ -) [*0*,*0*, *10*] *10*)

syntax (*xsymbols*)

-BLub :: [*pttrn*, *'a set*, *'b*] ⇒ *'b* ((*3*⨆ -∈-./ -) [*0*,*0*, *10*] *10*)

translations
LUB x:*A*. *t* == *CONST lub* ((%*x*. *t*) ‘ *A*)

abbreviation
Lub (**binder** *LUB* *10*) **where**
LUB n. *t n* == *lub* (*range t*)

notation (*xsymbols*)
Lub (**binder** ⨆ *10*)

access to some definition as inference rule

lemma *is-lubD1*: *S* <<| *x* ⟹ *S* <| *x*
⟨*proof*⟩

lemma *is-lub-lub*: ⟦*S* <<| *x*; *S* <| *u*⟧ ⟹ *x* ⊑ *u*
⟨*proof*⟩

lemma *is-lubI*: ⟦*S* <| *x*; ⋀*u*. *S* <| *u* ⟹ *x* ⊑ *u*⟧ ⟹ *S* <<| *x*
⟨*proof*⟩

lubs are unique

lemma *unique-lub*: ⟦*S* <<| *x*; *S* <<| *y*⟧ ⟹ *x* = *y*
⟨*proof*⟩

technical lemmas about *lub* and *op* <<|

lemma *lubI*: *M* <<| *x* ⟹ *M* <<| *lub M*
⟨*proof*⟩

lemma *thelubI*: *M* <<| *l* ⟹ *lub M* = *l*
⟨*proof*⟩

lemma *is-lub-singleton*: {*x*} <<| *x*
⟨*proof*⟩

lemma *lub-singleton* [*simp*]: *lub* {*x*} = *x*
⟨*proof*⟩

lemma *is-lub-bin*: *x* ⊑ *y* ⟹ {*x*, *y*} <<| *y*
⟨*proof*⟩

lemma *lub-bin*: *x* ⊑ *y* ⟹ *lub* {*x*, *y*} = *y*
⟨*proof*⟩

lemma *is-lub-maximal*: ⟦*S* <| *x*; *x* ∈ *S*⟧ ⟹ *S* <<| *x*
⟨*proof*⟩

lemma *lub-maximal*: ⟦*S* <| *x*; *x* ∈ *S*⟧ ⟹ *lub S* = *x*

⟨*proof*⟩

1.4 Countable chains

definition
— Here we use countable chains and I prefer to code them as functions!
chain :: (*nat* ⇒ *'a*::*po*) ⇒ *bool* **where**
chain Y = (∀ *i*. *Y i* ⊑ *Y* (*Suc i*))

lemma *chainI*: (⋀*i*. *Y i* ⊑ *Y* (*Suc i*)) ⟹ *chain Y*
⟨*proof*⟩

lemma *chainE*: *chain Y* ⟹ *Y i* ⊑ *Y* (*Suc i*)
⟨*proof*⟩

chains are monotone functions

lemma *chain-mono-less*: ⟦*chain Y*; *i* < *j*⟧ ⟹ *Y i* ⊑ *Y j*
⟨*proof*⟩

lemma *chain-mono*: ⟦*chain Y*; *i* ≤ *j*⟧ ⟹ *Y i* ⊑ *Y j*
⟨*proof*⟩

lemma *chain-shift*: *chain Y* ⟹ *chain* (λ*i*. *Y* (*i* + *j*))
⟨*proof*⟩

technical lemmas about (least) upper bounds of chains

lemma *is-ub-lub*: *range S* <<| *x* ⟹ *S i* ⊑ *x*
⟨*proof*⟩

lemma *is-ub-range-shift*:
chain S ⟹ *range* (λ*i*. *S* (*i* + *j*)) <| *x* = *range S* <| *x*
⟨*proof*⟩

lemma *is-lub-range-shift*:
chain S ⟹ *range* (λ*i*. *S* (*i* + *j*)) <<| *x* = *range S* <<| *x*
⟨*proof*⟩

the lub of a constant chain is the constant

lemma *chain-const* [*simp*]: *chain* (λ*i*. *c*)
⟨*proof*⟩

lemma *lub-const*: *range* (λ*x*. *c*) <<| *c*
⟨*proof*⟩

lemma *thelub-const* [*simp*]: (⨆ *i*. *c*) = *c*
⟨*proof*⟩

1.5 Finite chains

definition

— finite chains, needed for monotony of continuous functions
max-in-chain :: [*nat*, *nat* ⇒ *'a*::*po*] ⇒ *bool* **where**
max-in-chain i C = (∀ *j*. *i* ≤ *j* ⟶ *C i* = *C j*)

definition
finite-chain :: (*nat* ⇒ *'a*::*po*) ⇒ *bool* **where**
finite-chain C = (*chain C* ∧ (∃ *i*. *max-in-chain i C*))

results about finite chains

lemma *max-in-chainI*: (⋀*j*. *i* ≤ *j* ⟹ *Y i* = *Y j*) ⟹ *max-in-chain i Y*
⟨*proof*⟩

lemma *max-in-chainD*: ⟦*max-in-chain i Y*; *i* ≤ *j*⟧ ⟹ *Y i* = *Y j*
⟨*proof*⟩

lemma *finite-chainI*:
⟦*chain C*; *max-in-chain i C*⟧ ⟹ *finite-chain C*
⟨*proof*⟩

lemma *finite-chainE*:
⟦*finite-chain C*; ⋀*i*. ⟦*chain C*; *max-in-chain i C*⟧ ⟹ *R*⟧ ⟹ *R*
⟨*proof*⟩

lemma *lub-finch1*: ⟦*chain C*; *max-in-chain i C*⟧ ⟹ *range C* <<| *C i*
⟨*proof*⟩

lemma *lub-finch2*:
finite-chain C ⟹ *range C* <<| *C* (*LEAST i*. *max-in-chain i C*)
⟨*proof*⟩

lemma *finch-imp-finite-range*: *finite-chain Y* ⟹ *finite* (*range Y*)
⟨*proof*⟩

lemma *finite-range-has-max*:
fixes *f* :: *nat* ⇒ *'a* **and** *r* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *mono*: ⋀*i j*. *i* ≤ *j* ⟹ *r* (*f i*) (*f j*)
assumes *finite-range*: *finite* (*range f*)
shows ∃ *k*. ∀ *i*. *r* (*f i*) (*f k*)
⟨*proof*⟩

lemma *finite-range-imp-finch*:
⟦*chain Y*; *finite* (*range Y*)⟧ ⟹ *finite-chain Y*
⟨*proof*⟩

lemma *bin-chain*: *x* ⊑ *y* ⟹ *chain* (λ*i*. *if i=0 then x else y*)
⟨*proof*⟩

lemma *bin-chainmax*:
x ⊑ *y* ⟹ *max-in-chain* (*Suc 0*) (λ*i*. *if i=0 then x else y*)

⟨*proof*⟩

lemma *lub-bin-chain*:
$x \sqsubseteq y \Longrightarrow range\ (\lambda i{::}nat.\ if\ i{=}0\ then\ x\ else\ y) <<|\ y$
⟨*proof*⟩

the maximal element in a chain is its lub

lemma *lub-chain-maxelem*: $[\![Y\ i = c;\ \forall\, i.\ Y\ i \sqsubseteq c]\!] \Longrightarrow lub\ (range\ Y) = c$
⟨*proof*⟩

1.6 Directed sets

definition
$directed :: {'a}{::}po\ set \Rightarrow bool$ **where**
$directed\ S = ((\exists\, x.\ x \in S) \wedge (\forall\, x{\in}S.\ \forall\, y{\in}S.\ \exists\, z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z))$

lemma *directedI*:
assumes *1*: $\exists\, z.\ z \in S$
assumes *2*: $\bigwedge x\ y.\ [\![x \in S;\ y \in S]\!] \Longrightarrow \exists\, z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$
shows *directed S*
⟨*proof*⟩

lemma *directedD1*: $directed\ S \Longrightarrow \exists\, z.\ z \in S$
⟨*proof*⟩

lemma *directedD2*: $[\![directed\ S;\ x \in S;\ y \in S]\!] \Longrightarrow \exists\, z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$
⟨*proof*⟩

lemma *directedE1*:
assumes *S*: *directed S*
obtains *z* **where** $z \in S$
⟨*proof*⟩

lemma *directedE2*:
assumes *S*: *directed S*
assumes *x*: $x \in S$ **and** *y*: $y \in S$
obtains *z* **where** $z \in S\ x \sqsubseteq z\ y \sqsubseteq z$
⟨*proof*⟩

lemma *directed-finiteI*:
assumes *U*: $\bigwedge U.\ [\![finite\ U;\ U \subseteq S]\!] \Longrightarrow \exists\, z{\in}S.\ U <|\ z$
shows *directed S*
⟨*proof*⟩

lemma *directed-finiteD*:
assumes *S*: *directed S*
shows $[\![finite\ U;\ U \subseteq S]\!] \Longrightarrow \exists\, z{\in}S.\ U <|\ z$
⟨*proof*⟩

lemma *not-directed-empty* [*simp*]: ¬ *directed* {}
⟨*proof*⟩

lemma *directed-singleton*: *directed* {*x*}
⟨*proof*⟩

lemma *directed-bin*: $x \sqsubseteq y \Longrightarrow$ *directed* {*x*, *y*}
⟨*proof*⟩

lemma *directed-chain*: *chain S* ⟹ *directed* (*range S*)
⟨*proof*⟩

end

2 Pcpo: Classes cpo and pcpo

theory *Pcpo*
imports *Porder*
begin

2.1 Complete partial orders

The class cpo of chain complete partial orders

class *cpo* = *po* +
— class axiom:
assumes *cpo*: *chain S* ⟹ ∃ *x* :: *'a*::*po*. *range S* <<| *x*

in cpo's everthing equal to THE lub has lub properties for every chain

lemma *cpo-lubI*: *chain* (*S*::*nat* ⇒ *'a*::*cpo*) ⟹ *range S* <<| (⨆ *i*. *S i*)
⟨*proof*⟩

lemma *thelubE*: ⟦*chain S*; (⨆ *i*. *S i*) = (*l*::*'a*::*cpo*)⟧ ⟹ *range S* <<| *l*
⟨*proof*⟩

Properties of the lub

lemma *is-ub-thelub*: *chain* (*S*::*nat* ⇒ *'a*::*cpo*) ⟹ *S x* ⊑ (⨆ *i*. *S i*)
⟨*proof*⟩

lemma *is-lub-thelub*:
⟦*chain* (*S*::*nat* ⇒ *'a*::*cpo*); *range S* <| *x*⟧ ⟹ (⨆ *i*. *S i*) ⊑ *x*
⟨*proof*⟩

lemma *lub-range-mono*:
⟦*range X* ⊆ *range Y*; *chain Y*; *chain* (*X*::*nat* ⇒ *'a*::*cpo*)⟧
⟹ (⨆ *i*. *X i*) ⊑ (⨆ *i*. *Y i*)
⟨*proof*⟩

lemma *lub-range-shift*:
chain (*Y*::*nat* ⇒ 'a::*cpo*) ⟹ (⨆ *i*. *Y* (*i* + *j*)) = (⨆ *i*. *Y i*)
⟨*proof*⟩

lemma *maxinch-is-thelub*:
chain Y ⟹ *max-in-chain i Y* = ((⨆ *i*. *Y i*) = ((*Y i*)::'a::*cpo*))
⟨*proof*⟩

the ⊑ relation between two chains is preserved by their lubs

lemma *lub-mono*:
⟦*chain* (*X*::*nat* ⇒ 'a::*cpo*); *chain Y*; ⋀*i*. *X i* ⊑ *Y i*⟧
⟹ (⨆ *i*. *X i*) ⊑ (⨆ *i*. *Y i*)
⟨*proof*⟩

the = relation between two chains is preserved by their lubs

lemma *lub-equal*:
⟦*chain* (*X*::*nat* ⇒ 'a::*cpo*); *chain Y*; ∀ *k*. *X k* = *Y k*⟧
⟹ (⨆ *i*. *X i*) = (⨆ *i*. *Y i*)
⟨*proof*⟩

more results about mono and = of lubs of chains

lemma *lub-mono2*:
⟦∃ *j*. ∀ *i*>*j*. *X i* = *Y i*; *chain* (*X*::*nat* ⇒ 'a::*cpo*); *chain Y*⟧
⟹ (⨆ *i*. *X i*) ⊑ (⨆ *i*. *Y i*)
⟨*proof*⟩

lemma *lub-equal2*:
⟦∃ *j*. ∀ *i*>*j*. *X i* = *Y i*; *chain* (*X*::*nat* ⇒ 'a::*cpo*); *chain Y*⟧
⟹ (⨆ *i*. *X i*) = (⨆ *i*. *Y i*)
⟨*proof*⟩

lemma *lub-mono3*:
⟦*chain* (*Y*::*nat* ⇒ 'a::*cpo*); *chain X*; ∀ *i*. ∃ *j*. *Y i* ⊑ *X j*⟧
⟹ (⨆ *i*. *Y i*) ⊑ (⨆ *i*. *X i*)
⟨*proof*⟩

lemma *ch2ch-lub*:
fixes *Y* :: *nat* ⇒ *nat* ⇒ 'a::*cpo*
assumes *1*: ⋀*j*. *chain* (λ*i*. *Y i j*)
assumes *2*: ⋀*i*. *chain* (λ*j*. *Y i j*)
shows *chain* (λ*i*. ⨆ *j*. *Y i j*)
⟨*proof*⟩

lemma *diag-lub*:
fixes *Y* :: *nat* ⇒ *nat* ⇒ 'a::*cpo*
assumes *1*: ⋀*j*. *chain* (λ*i*. *Y i j*)
assumes *2*: ⋀*i*. *chain* (λ*j*. *Y i j*)
shows (⨆ *i*. ⨆ *j*. *Y i j*) = (⨆ *i*. *Y i i*)
⟨*proof*⟩

lemma *ex-lub*:
fixes $Y :: nat \Rightarrow nat \Rightarrow 'a::cpo$
assumes *1*: $\bigwedge j.\ chain\ (\lambda i.\ Y\ i\ j)$
assumes *2*: $\bigwedge i.\ chain\ (\lambda j.\ Y\ i\ j)$
shows $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) = (\bigsqcup j.\ \bigsqcup i.\ Y\ i\ j)$
⟨*proof*⟩

2.2 Pointed cpos

The class pcpo of pointed cpos

class *pcpo* = *cpo* +
assumes *least*: $\exists x.\ \forall y.\ x \sqsubseteq y$

definition
$UU :: 'a::pcpo$ **where**
$UU = (THE\ x.\ \forall y.\ x \sqsubseteq y)$

notation (*xsymbols*)
UU $(\bot)$

derive the old rule minimal

lemma *UU-least*: $\forall z.\ \bot \sqsubseteq z$
⟨*proof*⟩

lemma *minimal* [*iff*]: $\bot \sqsubseteq x$
⟨*proof*⟩

lemma *UU-reorient*: $(\bot = x) = (x = \bot)$
⟨*proof*⟩

⟨*ML*⟩

useful lemmas about $\bot$

lemma *less-UU-iff* [*simp*]: $(x \sqsubseteq \bot) = (x = \bot)$
⟨*proof*⟩

lemma *eq-UU-iff*: $(x = \bot) = (x \sqsubseteq \bot)$
⟨*proof*⟩

lemma *UU-I*: $x \sqsubseteq \bot \Longrightarrow x = \bot$
⟨*proof*⟩

lemma *not-less2not-eq*: $\neg\ (x::'a::po) \sqsubseteq y \Longrightarrow x \neq y$
⟨*proof*⟩

lemma *chain-UU-I*: $[\![chain\ Y;\ (\bigsqcup i.\ Y\ i) = \bot]\!] \Longrightarrow \forall i.\ Y\ i = \bot$
⟨*proof*⟩

lemma *chain-UU-I-inverse*: $\forall i{::}nat.\ Y\ i = \bot \Longrightarrow (\bigsqcup i.\ Y\ i) = \bot$
⟨*proof*⟩

lemma *chain-UU-I-inverse2*: $(\bigsqcup i.\ Y\ i) \neq \bot \Longrightarrow \exists i{::}nat.\ Y\ i \neq \bot$
⟨*proof*⟩

lemma *notUU-I*: $[\![x \sqsubseteq y;\ x \neq \bot]\!] \Longrightarrow y \neq \bot$
⟨*proof*⟩

lemma *chain-mono2*: $[\![\exists j.\ Y\ j \neq \bot;\ chain\ Y]\!] \Longrightarrow \exists j.\ \forall i{>}j.\ Y\ i \neq \bot$
⟨*proof*⟩

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

class *finite-po* = *finite* + *po*

class *chfin* = *po* +
assumes *chfin*: $chain\ Y \Longrightarrow \exists n.\ max\text{-}in\text{-}chain\ n\ (Y :: nat => 'a{::}po)$

class *flat* = *pcpo* +
assumes *ax-flat*: $(x :: 'a{::}pcpo) \sqsubseteq y \Longrightarrow x = \bot \vee x = y$

finite partial orders are chain-finite

instance *finite-po* < *chfin*
⟨*proof*⟩

some properties for chfin and flat

chfin types are cpo

instance *chfin* < *cpo*
⟨*proof*⟩

flat types are chfin

instance *flat* < *chfin*
⟨*proof*⟩

flat subclass of chfin; *adm-flat* not needed

lemma *flat-less-iff*:
fixes $x\ y :: 'a{::}flat$
shows $(x \sqsubseteq y) = (x = \bot \vee x = y)$
⟨*proof*⟩

lemma *flat-eq*: $(a{::}'a{::}flat) \neq \bot \Longrightarrow a \sqsubseteq b = (a = b)$
⟨*proof*⟩

lemma *chfin2finch*: $chain\ (Y{::}nat \Rightarrow 'a{::}chfin) \Longrightarrow finite\text{-}chain\ Y$

⟨*proof*⟩

Discrete cpos

class *discrete-cpo* = *sq-ord* +
  assumes *discrete-cpo* [*simp*]: $x \sqsubseteq y \longleftrightarrow x = y$

subclass (**in** *discrete-cpo*) *po*
⟨*proof*⟩

In a discrete cpo, every chain is constant

lemma *discrete-chain-const*:
  assumes S: *chain* (S::*nat* ⇒ $'a$::*discrete-cpo*)
  shows $\exists x.\ S = (\lambda i.\ x)$
⟨*proof*⟩

instance *discrete-cpo* < *cpo*
⟨*proof*⟩

lemmata for improved admissibility introdution rule

lemma *infinite-chain-adm-lemma*:
  $[\![chain\ Y;\ \forall i.\ P\ (Y\ i);$
    $\bigwedge Y.\ [\![chain\ Y;\ \forall i.\ P\ (Y\ i);\ \neg\ finite\text{-}chain\ Y]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)]\!]$
      $\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *increasing-chain-adm-lemma*:
  $[\![chain\ Y;\ \forall i.\ P\ (Y\ i);\ \bigwedge Y.\ [\![chain\ Y;\ \forall i.\ P\ (Y\ i);$
    $\forall i.\ \exists j{>}i.\ Y\ i \neq Y\ j \wedge Y\ i \sqsubseteq Y\ j]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)]\!]$
      $\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

end

3 Cont: Continuity and monotonicity

theory *Cont*
imports *Pcpo*
begin

Now we change the default class! Form now on all untyped type variables are of default class po

defaultsort *po*

3.1 Definitions

definition
  monofun :: ($'a$ ⇒ $'b$) ⇒ *bool* — monotonicity **where**

$monofun\ f = (\forall x\ y.\ x \sqsubseteq y \longrightarrow f\ x \sqsubseteq f\ y)$

definition
$contlub :: ('a::cpo \Rightarrow 'b::cpo) \Rightarrow bool$ — first cont. def **where**
$contlub\ f = (\forall\ Y.\ chain\ Y \longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i)))$

definition
$cont :: ('a::cpo \Rightarrow 'b::cpo) \Rightarrow bool$ — secnd cont. def **where**
$cont\ f = (\forall\ Y.\ chain\ Y \longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i))$

lemma *contlubI*:
$[\![\bigwedge Y.\ chain\ Y \Longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i))]\!] \Longrightarrow contlub\ f$
⟨*proof*⟩

lemma *contlubE*:
$[\![contlub\ f;\ chain\ Y]\!] \Longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i))$
⟨*proof*⟩

lemma *contI*:
$[\![\bigwedge Y.\ chain\ Y \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow cont\ f$
⟨*proof*⟩

lemma *contE*:
$[\![cont\ f;\ chain\ Y]\!] \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *monofunI*:
$[\![\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y]\!] \Longrightarrow monofun\ f$
⟨*proof*⟩

lemma *monofunE*:
$[\![monofun\ f;\ x \sqsubseteq y]\!] \Longrightarrow f\ x \sqsubseteq f\ y$
⟨*proof*⟩

3.2 $monofun\ f \wedge contlub\ f \equiv cont\ f$

monotone functions map chains to chains

lemma *ch2ch-monofun*: $[\![monofun\ f;\ chain\ Y]\!] \Longrightarrow chain\ (\lambda i.\ f\ (Y\ i))$
⟨*proof*⟩

monotone functions map upper bound to upper bounds

lemma *ub2ub-monofun*:
$[\![monofun\ f;\ range\ Y <|\ u]\!] \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <|\ f\ u$
⟨*proof*⟩

left to right: $monofun\ f \wedge contlub\ f \Longrightarrow cont\ f$

lemma *monocontlub2cont*: $[\![monofun\ f;\ contlub\ f]\!] \Longrightarrow cont\ f$
⟨*proof*⟩

first a lemma about binary chains

lemma *binchain-cont*:
$[\![cont\ f;\ x \sqsubseteq y]\!] \Longrightarrow range\ (\lambda i{::}nat.\ f\ (if\ i = 0\ then\ x\ else\ y)) <<|\ f\ y$
⟨*proof*⟩

right to left: $cont\ f \Longrightarrow monofun\ f \wedge contlub\ f$

part1: $cont\ f \Longrightarrow monofun\ f$

lemma *cont2mono*: $cont\ f \Longrightarrow monofun\ f$
⟨*proof*⟩

lemmas *cont2monofunE* = *cont2mono* [*THEN monofunE*]

lemmas *ch2ch-cont* = *cont2mono* [*THEN ch2ch-monofun*]

right to left: $cont\ f \Longrightarrow monofun\ f \wedge contlub\ f$

part2: $cont\ f \Longrightarrow contlub\ f$

lemma *cont2contlub*: $cont\ f \Longrightarrow contlub\ f$
⟨*proof*⟩

lemmas *cont2contlubE* = *cont2contlub* [*THEN contlubE*]

lemma *contI2*:
assumes *mono*: $monofun\ f$
assumes *less*: $\bigwedge Y.\ [\![chain\ Y;\ chain\ (\lambda i.\ f\ (Y\ i))]\!]$
$\Longrightarrow f\ (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ f\ (Y\ i))$
shows $cont\ f$
⟨*proof*⟩

3.3 Continuity simproc

⟨*ML*⟩
Given the term *cont f*, the procedure tries to construct the theorem *cont f* ≡ *True*. If this theorem cannot be completely solved by the introduction rules, then the procedure returns a conditional rewrite rule with the unsolved subgoals as premises.

⟨*ML*⟩

3.4 Continuity of basic functions

The identity function is continuous

lemma *cont-id* [*cont2cont*]: $cont\ (\lambda x.\ x)$
⟨*proof*⟩

constant functions are continuous

lemma *cont-const* [*cont2cont*]: $cont\ (\lambda x.\ c)$

⟨*proof*⟩

application of functions is continuous

lemma *cont2cont-apply*:
fixes $f :: 'a::cpo \Rightarrow 'b::cpo \Rightarrow 'c::cpo$ **and** $t :: 'a \Rightarrow 'b$
assumes *f1*: $\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)$
assumes *f2*: $\bigwedge x.\ cont\ (\lambda y.\ f\ x\ y)$
assumes *t*: $cont\ (\lambda x.\ t\ x)$
shows $cont\ (\lambda x.\ (f\ x)\ (t\ x))$
⟨*proof*⟩

lemma *cont2cont-compose*:
$[\![cont\ c;\ cont\ (\lambda x.\ f\ x)]\!] \Longrightarrow cont\ (\lambda x.\ c\ (f\ x))$
⟨*proof*⟩

if-then-else is continuous

lemma *cont-if* [*simp*]:
$[\![cont\ f;\ cont\ g]\!] \Longrightarrow cont\ (\lambda x.\ if\ b\ then\ f\ x\ else\ g\ x)$
⟨*proof*⟩

3.5 Finite chains and flat pcpos

monotone functions map finite chains to finite chains

lemma *monofun-finch2finch*:
$[\![monofun\ f;\ finite\text{-}chain\ Y]\!] \Longrightarrow finite\text{-}chain\ (\lambda n.\ f\ (Y\ n))$
⟨*proof*⟩

The same holds for continuous functions

lemma *cont-finch2finch*:
$[\![cont\ f;\ finite\text{-}chain\ Y]\!] \Longrightarrow finite\text{-}chain\ (\lambda n.\ f\ (Y\ n))$
⟨*proof*⟩

lemma *chfindom-monofun2cont*: $monofun\ f \Longrightarrow cont\ (f::'a::chfin \Rightarrow 'b::cpo)$
⟨*proof*⟩

some properties of flat

lemma *flatdom-strict2mono*: $f\ \bot = \bot \Longrightarrow monofun\ (f::'a::flat \Rightarrow 'b::pcpo)$
⟨*proof*⟩

lemma *flatdom-strict2cont*: $f\ \bot = \bot \Longrightarrow cont\ (f::'a::flat \Rightarrow 'b::pcpo)$
⟨*proof*⟩

functions with discrete domain

lemma *cont-discrete-cpo* [*simp*]: $cont\ (f::'a::discrete\text{-}cpo \Rightarrow 'b::cpo)$
⟨*proof*⟩

end

4 Adm: Admissibility and compactness

theory *Adm*
imports *Cont*
begin

defaultsort *cpo*

4.1 Definitions

definition
$adm :: ('a::cpo \Rightarrow bool) \Rightarrow bool$ **where**
$adm\ P = (\forall Y.\ chain\ Y \longrightarrow (\forall i.\ P\ (Y\ i)) \longrightarrow P\ (\bigsqcup i.\ Y\ i))$

lemma *admI*:
$(\bigwedge Y.\ [\![chain\ Y; \forall i.\ P\ (Y\ i)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)) \Longrightarrow adm\ P$
⟨*proof*⟩

lemma *admD*: $[\![adm\ P; chain\ Y; \bigwedge i.\ P\ (Y\ i)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *admD2*: $[\![adm\ (\lambda x.\ \neg\ P\ x); chain\ Y; P\ (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists i.\ P\ (Y\ i)$
⟨*proof*⟩

lemma *triv-admI*: $\forall x.\ P\ x \Longrightarrow adm\ P$
⟨*proof*⟩

improved admissibility introduction

lemma *admI2*:
$(\bigwedge Y.\ [\![chain\ Y; \forall i.\ P\ (Y\ i); \forall i.\ \exists j{>}i.\ Y\ i \neq Y\ j \wedge Y\ i \sqsubseteq Y\ j]\!]$
$\Longrightarrow P\ (\bigsqcup i.\ Y\ i)) \Longrightarrow adm\ P$
⟨*proof*⟩

4.2 Admissibility on chain-finite types

for chain-finite (easy) types every formula is admissible

lemma *adm-chfin*: $adm\ (P::'a::chfin \Rightarrow bool)$
⟨*proof*⟩

4.3 Admissibility of special formulae and propagation

lemma *adm-not-free*: $adm\ (\lambda x.\ t)$
⟨*proof*⟩

lemma *adm-conj*: $[\![adm\ P; adm\ Q]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \wedge Q\ x)$
⟨*proof*⟩

lemma *adm-all*: $(\bigwedge y.\ adm\ (\lambda x.\ P\ x\ y)) \Longrightarrow adm\ (\lambda x.\ \forall y.\ P\ x\ y)$
⟨*proof*⟩

lemma *adm-ball*: $(\bigwedge y.\ y \in A \Longrightarrow adm\ (\lambda x.\ P\ x\ y)) \Longrightarrow adm\ (\lambda x.\ \forall\, y{\in}A.\ P\ x\ y)$
⟨*proof*⟩

Admissibility for disjunction is hard to prove. It takes 5 Lemmas

lemma *adm-disj-lemma1*:
$[\![chain\ (Y{::}nat \Rightarrow {'a}{::}cpo);\ \forall\, i.\ \exists\, j{\geq}i.\ P\ (Y\ j)]\!]$
$\Longrightarrow chain\ (\lambda i.\ Y\ (LEAST\ j.\ i \leq j \wedge P\ (Y\ j)))$
⟨*proof*⟩

lemmas *adm-disj-lemma2* = *LeastI-ex* [*of* $\lambda j.\ i \leq j \wedge P\ (Y\ j)$, *standard*]

lemma *adm-disj-lemma3*:
$[\![chain\ (Y{::}nat \Rightarrow {'a}{::}cpo);\ \forall\, i.\ \exists\, j{\geq}i.\ P\ (Y\ j)]\!] \Longrightarrow$
$(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ Y\ (LEAST\ j.\ i \leq j \wedge P\ (Y\ j)))$
⟨*proof*⟩

lemma *adm-disj-lemma4*:
$[\![adm\ P;\ chain\ Y;\ \forall\, i.\ \exists\, j{\geq}i.\ P\ (Y\ j)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *adm-disj-lemma5*:
$\forall\, n{::}nat.\ P\ n \vee Q\ n \Longrightarrow (\forall\, i.\ \exists\, j{\geq}i.\ P\ j) \vee (\forall\, i.\ \exists\, j{\geq}i.\ Q\ j)$
⟨*proof*⟩

lemma *adm-disj*: $[\![adm\ P;\ adm\ Q]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \vee Q\ x)$
⟨*proof*⟩

lemma *adm-imp*: $[\![adm\ (\lambda x.\ \neg\ P\ x);\ adm\ Q]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \longrightarrow Q\ x)$
⟨*proof*⟩

lemma *adm-iff*:
$[\![adm\ (\lambda x.\ P\ x \longrightarrow Q\ x);\ adm\ (\lambda x.\ Q\ x \longrightarrow P\ x)]\!]$
$\Longrightarrow adm\ (\lambda x.\ P\ x = Q\ x)$
⟨*proof*⟩

lemma *adm-not-conj*:
$[\![adm\ (\lambda x.\ \neg\ P\ x);\ adm\ (\lambda x.\ \neg\ Q\ x)]\!] \Longrightarrow adm\ (\lambda x.\ \neg\ (P\ x \wedge Q\ x))$
⟨*proof*⟩

admissibility and continuity

lemma *adm-less*: $[\![cont\ u;\ cont\ v]\!] \Longrightarrow adm\ (\lambda x.\ u\ x \sqsubseteq v\ x)$
⟨*proof*⟩

lemma *adm-eq*: $[\![cont\ u;\ cont\ v]\!] \Longrightarrow adm\ (\lambda x.\ u\ x = v\ x)$
⟨*proof*⟩

lemma *adm-subst*: $[\![cont\ t;\ adm\ P]\!] \Longrightarrow adm\ (\lambda x.\ P\ (t\ x))$
⟨*proof*⟩

lemma *adm-not-less*: *cont t* $\Longrightarrow$ *adm* $(\lambda x.\ \neg\ t\ x \sqsubseteq u)$
⟨*proof*⟩

4.4 Compactness

definition
compact :: $'a$::*cpo* $\Rightarrow$ *bool* **where**
compact k = *adm* $(\lambda x.\ \neg\ k \sqsubseteq x)$

lemma *compactI*: *adm* $(\lambda x.\ \neg\ k \sqsubseteq x)$ $\Longrightarrow$ *compact k*
⟨*proof*⟩

lemma *compactD*: *compact k* $\Longrightarrow$ *adm* $(\lambda x.\ \neg\ k \sqsubseteq x)$
⟨*proof*⟩

lemma *compactI2*:
$(\bigwedge Y.\ [\![chain\ Y;\ x \sqsubseteq (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists\, i.\ x \sqsubseteq Y\ i) \Longrightarrow$ *compact x*
⟨*proof*⟩

lemma *compactD2*:
$[\![compact\ x;\ chain\ Y;\ x \sqsubseteq (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists\, i.\ x \sqsubseteq Y\ i$
⟨*proof*⟩

lemma *compact-chfin* [*simp*]: *compact* $(x::'a::chfin)$
⟨*proof*⟩

lemma *compact-imp-max-in-chain*:
$[\![chain\ Y;\ compact\ (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists\, i.\ max\text{-}in\text{-}chain\ i\ Y$
⟨*proof*⟩

admissibility and compactness

lemma *adm-compact-not-less*: $[\![compact\ k;\ cont\ t]\!] \Longrightarrow adm\ (\lambda x.\ \neg\ k \sqsubseteq t\ x)$
⟨*proof*⟩

lemma *adm-neq-compact*: $[\![compact\ k;\ cont\ t]\!] \Longrightarrow adm\ (\lambda x.\ t\ x \neq k)$
⟨*proof*⟩

lemma *adm-compact-neq*: $[\![compact\ k;\ cont\ t]\!] \Longrightarrow adm\ (\lambda x.\ k \neq t\ x)$
⟨*proof*⟩

lemma *compact-UU* [*simp*, *intro*]: *compact* $\bot$
⟨*proof*⟩

lemma *adm-not-UU*: *cont t* $\Longrightarrow$ *adm* $(\lambda x.\ t\ x \neq \bot)$
⟨*proof*⟩

Any upward-closed predicate is admissible.

lemma *adm-upward*:

assumes P: $\bigwedge x\ y.\ [\![P\ x;\ x \sqsubseteq y]\!] \Longrightarrow P\ y$
shows *adm P*
⟨*proof*⟩

lemmas *adm-lemmas* [*simp*] =
adm-not-free adm-conj adm-all adm-ball adm-disj adm-imp adm-iff
adm-less adm-eq adm-not-less
adm-compact-not-less adm-compact-neq adm-neq-compact adm-not-UU

end

5 Pcpodef: Subtypes of pcpos

theory *Pcpodef*
imports *Adm*
uses (*Tools/pcpodef-package.ML*)
begin

5.1 Proving a subtype is a partial order

A subtype of a partial order is itself a partial order, if the ordering is defined in the standard way.

⟨*ML*⟩

theorem *typedef-po*:
fixes *Abs* :: $'a$::*po* ⇒ $'b$::*type*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq \equiv \lambda x\ y.$ *Rep x* $\sqsubseteq$ *Rep y*
shows *OFCLASS*($'b$, *po-class*)
⟨*proof*⟩

⟨*ML*⟩

5.2 Proving a subtype is finite

lemma *typedef-finite-UNIV*:
fixes *Abs* :: $'a$::*type* ⇒ $'b$::*type*
assumes *type*: *type-definition Rep Abs A*
shows *finite A* ⟹ *finite* (*UNIV* :: $'b$ *set*)
⟨*proof*⟩

theorem *typedef-finite-po*:
fixes *Abs* :: $'a$::*finite-po* ⇒ $'b$::*po*
assumes *type*: *type-definition Rep Abs A*
shows *OFCLASS*($'b$, *finite-po-class*)
⟨*proof*⟩

5.3 Proving a subtype is chain-finite

lemma *monofun-Rep*:
assumes *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows *monofun Rep*
⟨*proof*⟩

lemmas *ch2ch-Rep* = *ch2ch-monofun* [*OF monofun-Rep*]
lemmas *ub2ub-Rep* = *ub2ub-monofun* [*OF monofun-Rep*]

theorem *typedef-chfin*:
fixes *Abs* :: $'a$::*chfin* ⇒ $'b$::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows *OFCLASS*($'b$, *chfin-class*)
⟨*proof*⟩

5.4 Proving a subtype is complete

A subtype of a cpo is itself a cpo if the ordering is defined in the standard way, and the defining subset is closed with respect to limits of chains. A set is closed if and only if membership in the set is an admissible predicate.

lemma *Abs-inverse-lub-Rep*:
fixes *Abs* :: $'a$::*cpo* ⇒ $'b$::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: *adm* $(\lambda x.\ x \in A)$
shows *chain S* ⟹ $Rep\ (Abs\ (\bigsqcup i.\ Rep\ (S\ i))) = (\bigsqcup i.\ Rep\ (S\ i))$
⟨*proof*⟩

theorem *typedef-lub*:
fixes *Abs* :: $'a$::*cpo* ⇒ $'b$::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: *adm* $(\lambda x.\ x \in A)$
shows *chain S* ⟹ *range S* <<| *Abs* $(\bigsqcup i.\ Rep\ (S\ i))$
⟨*proof*⟩

lemmas *typedef-thelub* = *typedef-lub* [*THEN thelubI*, *standard*]

theorem *typedef-cpo*:
fixes *Abs* :: $'a$::*cpo* ⇒ $'b$::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: *adm* $(\lambda x.\ x \in A)$
shows *OFCLASS*($'b$, *cpo-class*)
⟨*proof*⟩

5.4.1 Continuity of *Rep* and *Abs*

For any sub-cpo, the *Rep* function is continuous.

theorem *typedef-cont-Rep*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *cont Rep*
⟨*proof*⟩

For a sub-cpo, we can make the *Abs* function continuous only if we restrict its domain to the defining subset by composing it with another continuous function.

theorem *typedef-is-lubI*:
assumes *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
shows *range* (λ*i*. *Rep* (*S i*)) <<| *Rep x* ⟹ *range S* <<| *x*
⟨*proof*⟩

theorem *typedef-cont-Abs*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
fixes *f* :: *'c*::*cpo* ⇒ *'a*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
and *f-in-A*: ⋀*x*. *f x* ∈ *A*
and *cont-f*: *cont f*
shows *cont* (λ*x*. *Abs* (*f x*))
⟨*proof*⟩

5.5 Proving subtype elements are compact

theorem *typedef-compact*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *compact* (*Rep k*) ⟹ *compact k*
⟨*proof*⟩

5.6 Proving a subtype is pointed

A subtype of a cpo has a least element if and only if the defining subset has a least element.

theorem *typedef-pcpo-generic*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*

and *z-in-A*: $z \in A$
and *z-least*: $\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x$
shows *OFCLASS*('*b*, *pcpo-class*)
⟨*proof*⟩

As a special case, a subtype of a pcpo has a least element if the defining subset contains $\bot$.

theorem *typedef-pcpo*:
fixes *Abs* :: '*a*::*pcpo* ⇒ '*b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows *OFCLASS*('*b*, *pcpo-class*)
⟨*proof*⟩

5.6.1 Strictness of *Rep* and *Abs*

For a sub-pcpo where $\bot$ is a member of the defining subset, *Rep* and *Abs* are both strict.

theorem *typedef-Abs-strict*:
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $Abs\ \bot = \bot$
⟨*proof*⟩

theorem *typedef-Rep-strict*:
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $Rep\ \bot = \bot$
⟨*proof*⟩

theorem *typedef-Abs-strict-iff*:
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $x \in A \Longrightarrow (Abs\ x = \bot) = (x = \bot)$
⟨*proof*⟩

theorem *typedef-Rep-strict-iff*:
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $(Rep\ x = \bot) = (x = \bot)$
⟨*proof*⟩

theorem *typedef-Abs-defined*:

assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $[\![x \neq \bot;\ x \in A]\!] \Longrightarrow Abs\ x \neq \bot$
⟨*proof*⟩

theorem *typedef-Rep-defined*:
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $x \neq \bot \Longrightarrow Rep\ x \neq \bot$
⟨*proof*⟩

5.7 Proving a subtype is flat

theorem *typedef-flat*:
fixes *Abs* :: $'a$::*flat* ⇒ $'b$::*pcpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows *OFCLASS*($'b$, *flat-class*)
⟨*proof*⟩

5.8 HOLCF type definition package

⟨*ML*⟩

end

6 Ffun: Class instances for the full function space

theory *Ffun*
imports *Cont*
begin

6.1 Full function space is a partial order

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

definition
less-fun-def: $(op \sqsubseteq) \equiv (\lambda f\ g.\ \forall x.\ f\ x \sqsubseteq g\ x)$

instance ⟨*proof*⟩
end

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

make the symbol << accessible for type fun

lemma *expand-fun-less*: $(f \sqsubseteq g) = (\forall x.\ f\ x \sqsubseteq g\ x)$
⟨*proof*⟩

lemma *less-fun-ext*: $(\bigwedge x.\ f\ x \sqsubseteq g\ x) \Longrightarrow f \sqsubseteq g$
⟨*proof*⟩

6.2 Full function space is chain complete

function application is monotone

lemma *monofun-app*: *monofun* $(\lambda f.\ f\ x)$
⟨*proof*⟩

chains of functions yield chains in the po range

lemma *ch2ch-fun*: *chain* $S \Longrightarrow$ *chain* $(\lambda i.\ S\ i\ x)$
⟨*proof*⟩

lemma *ch2ch-lambda*: $(\bigwedge x.$ *chain* $(\lambda i.\ S\ i\ x)) \Longrightarrow$ *chain* S
⟨*proof*⟩

upper bounds of function chains yield upper bound in the po range

lemma *ub2ub-fun*:
range $S <|\ u \Longrightarrow$ *range* $(\lambda i.\ S\ i\ x) <|\ u\ x$
⟨*proof*⟩

Type $'a \Rightarrow 'b$ is chain complete

lemma *is-lub-lambda*:
assumes f: $\bigwedge x.$ *range* $(\lambda i.\ Y\ i\ x) <<|\ f\ x$
shows *range* $Y <<|\ f$
⟨*proof*⟩

lemma *lub-fun*:
chain $(S{::}nat \Rightarrow 'a{::}type \Rightarrow 'b{::}cpo)$
$\Longrightarrow$ *range* $S <<|\ (\lambda x.\ \bigsqcup i.\ S\ i\ x)$
⟨*proof*⟩

lemma *thelub-fun*:
chain $(S{::}nat \Rightarrow 'a{::}type \Rightarrow 'b{::}cpo)$
$\Longrightarrow (\bigsqcup i.\ S\ i) = (\lambda x.\ \bigsqcup i.\ S\ i\ x)$
⟨*proof*⟩

lemma *cpo-fun*:
chain $(S{::}nat \Rightarrow 'a{::}type \Rightarrow 'b{::}cpo) \Longrightarrow \exists x.$ *range* $S <<|\ x$
⟨*proof*⟩

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

instance *fun* :: (*finite*, *finite-po*) *finite-po* ⟨*proof*⟩

instance *fun* :: (*type*, *discrete-cpo*) *discrete-cpo*
⟨*proof*⟩

chain-finite function spaces

lemma *maxinch2maxinch-lambda*:
(⋀*x*. *max-in-chain n* (λ*i*. *S i x*)) ⟹ *max-in-chain n S*
⟨*proof*⟩

lemma *maxinch-mono*:
⟦*max-in-chain i Y*; *i* ≤ *j*⟧ ⟹ *max-in-chain j Y*
⟨*proof*⟩

instance *fun* :: (*finite*, *chfin*) *chfin*
⟨*proof*⟩

6.3 Full function space is pointed

lemma *minimal-fun*: (λ*x*. ⊥) ⊑ *f*
⟨*proof*⟩

lemma *least-fun*: ∃ *x*::'*a*::*type* ⇒ '*b*::*pcpo*. ∀ *y*. *x* ⊑ *y*
⟨*proof*⟩

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

for compatibility with old HOLCF-Version

lemma *inst-fun-pcpo*: ⊥ = (λ*x*. ⊥)
⟨*proof*⟩

function application is strict in the left argument

lemma *app-strict* [*simp*]: ⊥ *x* = ⊥
⟨*proof*⟩

The following results are about application for functions in '*a* ⇒ '*b*

lemma *monofun-fun-fun*: *f* ⊑ *g* ⟹ *f x* ⊑ *g x*
⟨*proof*⟩

lemma *monofun-fun-arg*: ⟦*monofun f*; *x* ⊑ *y*⟧ ⟹ *f x* ⊑ *f y*
⟨*proof*⟩

lemma *monofun-fun*: ⟦*monofun f*; *monofun g*; *f* ⊑ *g*; *x* ⊑ *y*⟧ ⟹ *f x* ⊑ *g y*
⟨*proof*⟩

6.4 Propagation of monotonicity and continuity

the lub of a chain of monotone functions is monotone

lemma *monofun-lub-fun*:
$[\![chain\ (F{::}nat \Rightarrow 'a \Rightarrow 'b{::}cpo);\ \forall i.\ monofun\ (F\ i)]\!]$
$\Longrightarrow monofun\ (\bigsqcup i.\ F\ i)$
⟨*proof*⟩

the lub of a chain of continuous functions is continuous

lemma *contlub-lub-fun*:
$[\![chain\ F;\ \forall i.\ cont\ (F\ i)]\!] \Longrightarrow contlub\ (\bigsqcup i.\ F\ i)$
⟨*proof*⟩

lemma *cont-lub-fun*:
$[\![chain\ F;\ \forall i.\ cont\ (F\ i)]\!] \Longrightarrow cont\ (\bigsqcup i.\ F\ i)$
⟨*proof*⟩

lemma *cont2cont-lub*:
$[\![chain\ F;\ \bigwedge i.\ cont\ (F\ i)]\!] \Longrightarrow cont\ (\lambda x.\ \bigsqcup i.\ F\ i\ x)$
⟨*proof*⟩

lemma *mono2mono-fun*: $monofun\ f \Longrightarrow monofun\ (\lambda x.\ f\ x\ y)$
⟨*proof*⟩

lemma *cont2cont-fun*: $cont\ f \Longrightarrow cont\ (\lambda x.\ f\ x\ y)$
⟨*proof*⟩

Note $(\lambda x.\ \lambda y.\ f\ x\ y) = f$

lemma *mono2mono-lambda*:
assumes f: $\bigwedge y.\ monofun\ (\lambda x.\ f\ x\ y)$ **shows** $monofun\ f$
⟨*proof*⟩

lemma *cont2cont-lambda* [*simp*]:
assumes f: $\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)$ **shows** $cont\ f$
⟨*proof*⟩

What D.A.Schmidt calls continuity of abstraction; never used here

lemma *contlub-lambda*:
$(\bigwedge x{::}'a{::}type.\ chain\ (\lambda i.\ S\ i\ x{::}'b{::}cpo))$
$\Longrightarrow (\lambda x.\ \bigsqcup i.\ S\ i\ x) = (\bigsqcup i.\ (\lambda x.\ S\ i\ x))$
⟨*proof*⟩

lemma *contlub-abstraction*:
$[\![chain\ Y;\ \forall y.\ cont\ (\lambda x.(c{::}'a{::}cpo \Rightarrow 'b{::}type \Rightarrow 'c{::}cpo)\ x\ y)]\!] \Longrightarrow$
$(\lambda y.\ \bigsqcup i.\ c\ (Y\ i)\ y) = (\bigsqcup i.\ (\lambda y.\ c\ (Y\ i)\ y))$
⟨*proof*⟩

lemma *mono2mono-app*:
$[\![monofun\ f;\ \forall x.\ monofun\ (f\ x);\ monofun\ t]\!] \Longrightarrow monofun\ (\lambda x.\ (f\ x)\ (t\ x))$
⟨*proof*⟩

lemma *cont2contlub-app*:

$[\![cont\ f;\ \forall x.\ cont\ (f\ x);\ cont\ t]\!] \Longrightarrow contlub\ (\lambda x.\ (f\ x)\ (t\ x))$
⟨*proof*⟩

lemma *cont2cont-app*:
$[\![cont\ f;\ \forall x.\ cont\ (f\ x);\ cont\ t]\!] \Longrightarrow cont\ (\lambda x.\ (f\ x)\ (t\ x))$
⟨*proof*⟩

lemmas *cont2cont-app2* = *cont2cont-app* [*rule-format*]

lemma *cont2cont-app3*: $[\![cont\ f;\ cont\ t]\!] \Longrightarrow cont\ (\lambda x.\ f\ (t\ x))$
⟨*proof*⟩

end

7 Product-Cpo: The cpo of cartesian products

theory *Product-Cpo*
imports *Adm*
begin

defaultsort *cpo*

7.1 Type *unit* is a pcpo

instantiation *unit* :: *sq-ord*
begin

definition
less-unit-def [*simp*]: $x \sqsubseteq (y::unit) \equiv True$

instance ⟨*proof*⟩
end

instance *unit* :: *discrete-cpo*
⟨*proof*⟩

instance *unit* :: *finite-po* ⟨*proof*⟩

instance *unit* :: *pcpo*
⟨*proof*⟩

7.2 Product type is a partial order

instantiation $*$ :: (*sq-ord*, *sq-ord*) *sq-ord*
begin

definition

less-cprod-def: $(op \sqsubseteq) \equiv \lambda p1\ p2.\ (fst\ p1 \sqsubseteq fst\ p2 \wedge snd\ p1 \sqsubseteq snd\ p2)$

instance ⟨*proof*⟩
end

instance $*$ :: (*po*, *po*) *po*
⟨*proof*⟩

7.3 Monotonicity of (-,-), *fst*, *snd*

lemma *prod-lessI*: $[\![fst\ p \sqsubseteq fst\ q;\ snd\ p \sqsubseteq snd\ q]\!] \Longrightarrow p \sqsubseteq q$
⟨*proof*⟩

lemma *Pair-less-iff* [*simp*]: $(a, b) \sqsubseteq (c, d) \longleftrightarrow a \sqsubseteq c \wedge b \sqsubseteq d$
⟨*proof*⟩

Pair (-,-) is monotone in both arguments

lemma *monofun-pair1*: *monofun* $(\lambda x.\ (x, y))$
⟨*proof*⟩

lemma *monofun-pair2*: *monofun* $(\lambda y.\ (x, y))$
⟨*proof*⟩

lemma *monofun-pair*:
$[\![x1 \sqsubseteq x2;\ y1 \sqsubseteq y2]\!] \Longrightarrow (x1, y1) \sqsubseteq (x2, y2)$
⟨*proof*⟩

fst and *snd* are monotone

lemma *monofun-fst*: *monofun fst*
⟨*proof*⟩

lemma *monofun-snd*: *monofun snd*
⟨*proof*⟩

7.4 Product type is a cpo

lemma *is-lub-Pair*:
$[\![range\ X <<|\ x;\ range\ Y <<|\ y]\!] \Longrightarrow range\ (\lambda i.\ (X\ i,\ Y\ i)) <<|\ (x, y)$
⟨*proof*⟩

lemma *lub-cprod*:
fixes S :: *nat* ⇒ ('*a*::*cpo* × '*b*::*cpo*)
assumes S: *chain S*
shows *range S* $<<|$ $(\bigsqcup i.\ fst\ (S\ i), \bigsqcup i.\ snd\ (S\ i))$
⟨*proof*⟩

lemma *thelub-cprod*:
chain (S::*nat* ⇒ '*a*::*cpo* × '*b*::*cpo*)
$\Longrightarrow (\bigsqcup i.\ S\ i) = (\bigsqcup i.\ fst\ (S\ i), \bigsqcup i.\ snd\ (S\ i))$

⟨*proof*⟩

instance $*$ :: (*cpo*, *cpo*) *cpo*
⟨*proof*⟩

instance $*$ :: (*finite-po*, *finite-po*) *finite-po* ⟨*proof*⟩

instance $*$ :: (*discrete-cpo*, *discrete-cpo*) *discrete-cpo*
⟨*proof*⟩

7.5 Product type is pointed

lemma *minimal-cprod*: $(\bot, \bot) \sqsubseteq p$
⟨*proof*⟩

instance $*$ :: (*pcpo*, *pcpo*) *pcpo*
⟨*proof*⟩

lemma *inst-cprod-pcpo*: $\bot = (\bot, \bot)$
⟨*proof*⟩

lemma *Pair-defined-iff* [*simp*]: $(x, y) = \bot \longleftrightarrow x = \bot \wedge y = \bot$
⟨*proof*⟩

lemma *fst-strict* [*simp*]: *fst* $\bot = \bot$
⟨*proof*⟩

lemma *csnd-strict* [*simp*]: *snd* $\bot = \bot$
⟨*proof*⟩

lemma *Pair-strict* [*simp*]: $(\bot, \bot) = \bot$
⟨*proof*⟩

lemma *split-strict* [*simp*]: *split* $f\ \bot = f\ \bot\ \bot$
⟨*proof*⟩

7.6 Continuity of (-,-), *fst*, *snd*

lemma *cont-pair1*: *cont* $(\lambda x.\ (x, y))$
⟨*proof*⟩

lemma *cont-pair2*: *cont* $(\lambda y.\ (x, y))$
⟨*proof*⟩

lemma *contlub-fst*: *contlub fst*
⟨*proof*⟩

lemma *contlub-snd*: *contlub snd*
⟨*proof*⟩

lemma *cont-fst*: *cont fst*
⟨*proof*⟩

lemma *cont-snd*: *cont snd*
⟨*proof*⟩

lemma *cont2cont-Pair* [*cont2cont*]:
assumes *f*: *cont* (λ*x*. *f x*)
assumes *g*: *cont* (λ*x*. *g x*)
shows *cont* (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

lemmas *cont2cont-fst* [*cont2cont*] = *cont2cont-compose* [*OF cont-fst*]

lemmas *cont2cont-snd* [*cont2cont*] = *cont2cont-compose* [*OF cont-snd*]

7.7 Compactness and chain-finiteness

lemma *fst-less-iff*: *fst* (*x*::'*a* × '*b*) ⊑ *y* ⟷ *x* ⊑ (*y*, *snd x*)
⟨*proof*⟩

lemma *snd-less-iff*: *snd* (*x*::'*a* × '*b*) ⊑ *y* = *x* ⊑ (*fst x*, *y*)
⟨*proof*⟩

lemma *compact-fst*: *compact x* ⟹ *compact* (*fst x*)
⟨*proof*⟩

lemma *compact-snd*: *compact x* ⟹ *compact* (*snd x*)
⟨*proof*⟩

lemma *compact-Pair*: ⟦*compact x*; *compact y*⟧ ⟹ *compact* (*x*, *y*)
⟨*proof*⟩

lemma *compact-Pair-iff* [*simp*]: *compact* (*x*, *y*) ⟷ *compact x* ∧ *compact y*
⟨*proof*⟩

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

end

8 Cfun: The type of continuous functions

theory *Cfun*
imports *Pcpodef Ffun Product-Cpo*
begin

defaultsort *cpo*

8.1 Definition of continuous function type

lemma *Ex-cont*: $\exists f.\ cont\ f$
⟨*proof*⟩

lemma *adm-cont*: *adm cont*
⟨*proof*⟩

cpodef (*CFun*) $('a, 'b) -> (\textbf{infixr}\ -> 0) = \{f::'a => 'b.\ cont\ f\}$
⟨*proof*⟩

syntax (*xsymbols*)
$->$ :: [*type*, *type*] => *type* ((- →/ -) [*1*,*0*]*0*)

notation
Rep-CFun ((-$/-) [*999*,*1000*] *999*)

notation (*xsymbols*)
Rep-CFun ((-·/-) [*999*,*1000*] *999*)

notation (*HTML* **output**)
Rep-CFun ((-·/-) [*999*,*1000*] *999*)

8.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: $'a$

⟨*ML*⟩

To avoid eta-contraction of body:

⟨*ML*⟩
Syntax for nested abstractions

syntax
-Lambda :: [*cargs*, $'a$] ⇒ *logic* ((*3LAM* -./ -) [*1000*, *10*] *10*)

syntax (*xsymbols*)
-Lambda :: [*cargs*, $'a$] ⇒ *logic* ((*3*Λ -./ -) [*1000*, *10*] *10*)

⟨*ML*⟩

Dummy patterns for continuous abstraction

translations
Λ -. *t* => *CONST Abs-CFun* (λ -. *t*)

8.3 Continuous function space is pointed

lemma *UU-CFun*: $\bot \in CFun$
⟨*proof*⟩

instance $->$ :: (*finite-po*, *finite-po*) *finite-po*
⟨*proof*⟩

instance $->$ :: (*finite-po*, *chfin*) *chfin*
⟨*proof*⟩

instance $->$ :: (*cpo*, *discrete-cpo*) *discrete-cpo*
⟨*proof*⟩

instance $->$ :: (*cpo*, *pcpo*) *pcpo*
⟨*proof*⟩

lemmas *Rep-CFun-strict* =
typedef-Rep-strict [*OF type-definition-CFun less-CFun-def UU-CFun*]

lemmas *Abs-CFun-strict* =
typedef-Abs-strict [*OF type-definition-CFun less-CFun-def UU-CFun*]

function application is strict in its first argument

lemma *Rep-CFun-strict1* [*simp*]: $\bot \cdot x = \bot$
⟨*proof*⟩

for compatibility with old HOLCF-Version

lemma *inst-cfun-pcpo*: $\bot = (\Lambda\ x.\ \bot)$
⟨*proof*⟩

8.4 Basic properties of continuous functions

Beta-equality for continuous functions

lemma *Abs-CFun-inverse2*: *cont* $f \Longrightarrow$ *Rep-CFun* (*Abs-CFun* f) $= f$
⟨*proof*⟩

lemma *beta-cfun* [*simp*]: *cont* $f \Longrightarrow (\Lambda\ x.\ f\ x) \cdot u = f\ u$
⟨*proof*⟩

Eta-equality for continuous functions

lemma *eta-cfun*: $(\Lambda\ x.\ f \cdot x) = f$
⟨*proof*⟩

Extensionality for continuous functions

lemma *expand-cfun-eq*: $(f = g) = (\forall x.\ f \cdot x = g \cdot x)$
⟨*proof*⟩

lemma *ext-cfun*: $(\bigwedge x.\ f \cdot x = g \cdot x) \Longrightarrow f = g$
⟨*proof*⟩

Extensionality wrt. ordering for continuous functions

lemma *expand-cfun-less*: $f \sqsubseteq g = (\forall x.\ f \cdot x \sqsubseteq g \cdot x)$

⟨*proof*⟩

lemma *less-cfun-ext*: $(\bigwedge x.\ f{\cdot}x \sqsubseteq g{\cdot}x) \Longrightarrow f \sqsubseteq g$
⟨*proof*⟩

Congruence for continuous function application

lemma *cfun-cong*: $[\![f = g;\ x = y]\!] \Longrightarrow f{\cdot}x = g{\cdot}y$
⟨*proof*⟩

lemma *cfun-fun-cong*: $f = g \Longrightarrow f{\cdot}x = g{\cdot}x$
⟨*proof*⟩

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

8.5 Continuity of application

lemma *cont-Rep-CFun1*: *cont* $(\lambda f.\ f{\cdot}x)$
⟨*proof*⟩

lemma *cont-Rep-CFun2*: *cont* $(\lambda x.\ f{\cdot}x)$
⟨*proof*⟩

lemmas *monofun-Rep-CFun* = *cont-Rep-CFun* [*THEN cont2mono*]
lemmas *contlub-Rep-CFun* = *cont-Rep-CFun* [*THEN cont2contlub*]

lemmas *monofun-Rep-CFun1* = *cont-Rep-CFun1* [*THEN cont2mono*, *standard*]
lemmas *contlub-Rep-CFun1* = *cont-Rep-CFun1* [*THEN cont2contlub*, *standard*]
lemmas *monofun-Rep-CFun2* = *cont-Rep-CFun2* [*THEN cont2mono*, *standard*]
lemmas *contlub-Rep-CFun2* = *cont-Rep-CFun2* [*THEN cont2contlub*, *standard*]

contlub, cont properties of *Rep-CFun* in each argument

lemma *contlub-cfun-arg*: *chain* $Y \Longrightarrow f{\cdot}(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f{\cdot}(Y\ i))$
⟨*proof*⟩

lemma *cont-cfun-arg*: *chain* $Y \Longrightarrow$ *range* $(\lambda i.\ f{\cdot}(Y\ i)) <<|\ f{\cdot}(\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *contlub-cfun-fun*: *chain* $F \Longrightarrow (\bigsqcup i.\ F\ i){\cdot}x = (\bigsqcup i.\ F\ i{\cdot}x)$
⟨*proof*⟩

lemma *cont-cfun-fun*: *chain* $F \Longrightarrow$ *range* $(\lambda i.\ F\ i{\cdot}x) <<|\ (\bigsqcup i.\ F\ i){\cdot}x$
⟨*proof*⟩

monotonicity of application

lemma *monofun-cfun-fun*: $f \sqsubseteq g \Longrightarrow f{\cdot}x \sqsubseteq g{\cdot}x$
⟨*proof*⟩

lemma *monofun-cfun-arg*: $x \sqsubseteq y \Longrightarrow f{\cdot}x \sqsubseteq f{\cdot}y$

⟨*proof*⟩

lemma *monofun-cfun*: ⟦$f \sqsubseteq g$; $x \sqsubseteq y$⟧ ⟹ $f{\cdot}x \sqsubseteq g{\cdot}y$
⟨*proof*⟩

ch2ch - rules for the type $'a \to 'b$

lemma *chain-monofun*: *chain* Y ⟹ *chain* $(\lambda i.\ f{\cdot}(Y\ i))$
⟨*proof*⟩

lemma *ch2ch-Rep-CFunR*: *chain* Y ⟹ *chain* $(\lambda i.\ f{\cdot}(Y\ i))$
⟨*proof*⟩

lemma *ch2ch-Rep-CFunL*: *chain* F ⟹ *chain* $(\lambda i.\ (F\ i){\cdot}x)$
⟨*proof*⟩

lemma *ch2ch-Rep-CFun* [*simp*]:
⟦*chain* F; *chain* Y⟧ ⟹ *chain* $(\lambda i.\ (F\ i){\cdot}(Y\ i))$
⟨*proof*⟩

lemma *ch2ch-LAM* [*simp*]:
⟦$\bigwedge x.$ *chain* $(\lambda i.\ S\ i\ x)$; $\bigwedge i.$ *cont* $(\lambda x.\ S\ i\ x)$⟧ ⟹ *chain* $(\lambda i.\ \Lambda\ x.\ S\ i\ x)$
⟨*proof*⟩

contlub, cont properties of *Rep-CFun* in both arguments

lemma *contlub-cfun*:
⟦*chain* F; *chain* Y⟧ ⟹ $(\bigsqcup i.\ F\ i){\cdot}(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ F\ i{\cdot}(Y\ i))$
⟨*proof*⟩

lemma *cont-cfun*:
⟦*chain* F; *chain* Y⟧ ⟹ *range* $(\lambda i.\ F\ i{\cdot}(Y\ i)) <<|\ (\bigsqcup i.\ F\ i){\cdot}(\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *contlub-LAM*:
⟦$\bigwedge x.$ *chain* $(\lambda i.\ F\ i\ x)$; $\bigwedge i.$ *cont* $(\lambda x.\ F\ i\ x)$⟧
⟹ $(\Lambda\ x.\ \bigsqcup i.\ F\ i\ x) = (\bigsqcup i.\ \Lambda\ x.\ F\ i\ x)$
⟨*proof*⟩

lemmas *lub-distribs* =
contlub-cfun [*symmetric*]
contlub-LAM [*symmetric*]

strictness

lemma *strictI*: $f{\cdot}x = \bot \Longrightarrow f{\cdot}\bot = \bot$
⟨*proof*⟩

the lub of a chain of continous functions is monotone

lemma *lub-cfun-mono*: *chain* F ⟹ *monofun* $(\lambda x.\ \bigsqcup i.\ F\ i{\cdot}x)$
⟨*proof*⟩

a lemma about the exchange of lubs for type $'a \to 'b$

lemma *ex-lub-cfun*:
$[\![chain\ F;\ chain\ Y]\!] \Longrightarrow (\bigsqcup j.\ \bigsqcup i.\ F\ j\cdot(Y\ i)) = (\bigsqcup i.\ \bigsqcup j.\ F\ j\cdot(Y\ i))$
⟨*proof*⟩

the lub of a chain of cont. functions is continuous

lemma *cont-lub-cfun*: $chain\ F \Longrightarrow cont\ (\lambda x.\ \bigsqcup i.\ F\ i\cdot x)$
⟨*proof*⟩

type $'a \to 'b$ is chain complete

lemma *lub-cfun*: $chain\ F \Longrightarrow range\ F <<|\ (\Lambda\ x.\ \bigsqcup i.\ F\ i\cdot x)$
⟨*proof*⟩

lemma *thelub-cfun*: $chain\ F \Longrightarrow (\bigsqcup i.\ F\ i) = (\Lambda\ x.\ \bigsqcup i.\ F\ i\cdot x)$
⟨*proof*⟩

8.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

lemma *cont2cont-Rep-CFun* [*cont2cont*]:
assumes *f*: $cont\ (\lambda x.\ f\ x)$
assumes *t*: $cont\ (\lambda x.\ t\ x)$
shows $cont\ (\lambda x.\ (f\ x)\cdot(t\ x))$
⟨*proof*⟩

cont2mono Lemma for $\lambda x.\ \Lambda\ y.\ c1\ x\ y$

lemma *cont2mono-LAM*:
$[\![\bigwedge x.\ cont\ (\lambda y.\ f\ x\ y);\ \bigwedge y.\ monofun\ (\lambda x.\ f\ x\ y)]\!]$
$\Longrightarrow monofun\ (\lambda x.\ \Lambda\ y.\ f\ x\ y)$
⟨*proof*⟩

cont2cont Lemma for $\lambda x.\ \Lambda\ y.\ f\ x\ y$

Not suitable as a cont2cont rule, because on nested lambdas it causes exponential blow-up in the number of subgoals.

lemma *cont2cont-LAM*:
assumes *f1*: $\bigwedge x.\ cont\ (\lambda y.\ f\ x\ y)$
assumes *f2*: $\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)$
shows $cont\ (\lambda x.\ \Lambda\ y.\ f\ x\ y)$
⟨*proof*⟩

This version does work as a cont2cont rule, since it has only a single subgoal.

lemma *cont2cont-LAM'* [*cont2cont*]:
fixes $f :: 'a::cpo \Rightarrow 'b::cpo \Rightarrow 'c::cpo$
assumes *f*: $cont\ (\lambda p.\ f\ (fst\ p)\ (snd\ p))$
shows $cont\ (\lambda x.\ \Lambda\ y.\ f\ x\ y)$
⟨*proof*⟩

lemma *cont2cont-LAM-discrete* [*cont2cont*]:
$(\bigwedge y::'a::\textit{discrete-cpo}.\ \textit{cont}\ (\lambda x.\ f\ x\ y)) \Longrightarrow \textit{cont}\ (\lambda x.\ \Lambda\ y.\ f\ x\ y)$
⟨*proof*⟩

lemmas *cont-lemmas1* =
cont-const cont-id cont-Rep-CFun2 cont2cont-Rep-CFun cont2cont-LAM

8.7 Miscellaneous

Monotonicity of *Abs-CFun*

lemma *semi-monofun-Abs-CFun*:
$[\![\textit{cont}\ f;\ \textit{cont}\ g;\ f \sqsubseteq g]\!] \Longrightarrow \textit{Abs-CFun}\ f \sqsubseteq \textit{Abs-CFun}\ g$
⟨*proof*⟩

some lemmata for functions with flat/chfin domain/range types

lemma *chfin-Rep-CFunR*: *chain* (*Y*::*nat* => *'a*::*cpo*−>*'b*::*chfin*)
==> !*s*. ? *n*. (*LUB i*. *Y i*)$*s* = *Y n*$*s*
⟨*proof*⟩

lemma *adm-chfindom*: *adm* $(\lambda(u::'a::\textit{cpo} \to 'b::\textit{chfin}).\ P(u{\cdot}s))$
⟨*proof*⟩

8.8 Continuous injection-retraction pairs

Continuous retractions are strict.

lemma *retraction-strict*:
$\forall x.\ f{\cdot}(g{\cdot}x) = x \Longrightarrow f{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *injection-eq*:
$\forall x.\ f{\cdot}(g{\cdot}x) = x \Longrightarrow (g{\cdot}x = g{\cdot}y) = (x = y)$
⟨*proof*⟩

lemma *injection-less*:
$\forall x.\ f{\cdot}(g{\cdot}x) = x \Longrightarrow (g{\cdot}x \sqsubseteq g{\cdot}y) = (x \sqsubseteq y)$
⟨*proof*⟩

lemma *injection-defined-rev*:
$[\![\forall x.\ f{\cdot}(g{\cdot}x) = x;\ g{\cdot}z = \bot]\!] \Longrightarrow z = \bot$
⟨*proof*⟩

lemma *injection-defined*:
$[\![\forall x.\ f{\cdot}(g{\cdot}x) = x;\ z \neq \bot]\!] \Longrightarrow g{\cdot}z \neq \bot$
⟨*proof*⟩

propagation of flatness and chain-finiteness by retractions

lemma *chfin2chfin*:

$\forall y.\ (f::'a::chfin \to 'b)\cdot(g\cdot y) = y$
$\quad \Longrightarrow \forall Y::nat \Rightarrow 'b.\ chain\ Y \longrightarrow (\exists n.\ max\text{-}in\text{-}chain\ n\ Y)$
⟨*proof*⟩

lemma *flat2flat*:
$\forall y.\ (f::'a::flat \to 'b::pcpo)\cdot(g\cdot y) = y$
$\quad \Longrightarrow \forall x\ y::'b.\ x \sqsubseteq y \longrightarrow x = \bot \vee x = y$
⟨*proof*⟩

a result about functions with flat codomain

lemma *flat-eqI*: $[\![(x::'a::flat) \sqsubseteq y;\ x \neq \bot]\!] \Longrightarrow x = y$
⟨*proof*⟩

lemma *flat-codom*:
$f\cdot x = (c::'b::flat) \Longrightarrow f\cdot\bot = \bot \vee (\forall z.\ f\cdot z = c)$
⟨*proof*⟩

8.9 Identity and composition

definition
$ID :: 'a \to 'a$ **where**
$ID = (\Lambda\ x.\ x)$

definition
$cfcomp :: ('b \to 'c) \to ('a \to 'b) \to 'a \to 'c$ **where**
oo-def: $cfcomp = (\Lambda\ f\ g\ x.\ f\cdot(g\cdot x))$

abbreviation
cfcomp-syn :: $['b \to 'c,\ 'a \to 'b] \Rightarrow 'a \to 'c$ (**infixr** *oo 100*) **where**
$f\ oo\ g == cfcomp\cdot f\cdot g$

lemma *ID1* [*simp*]: $ID\cdot x = x$
⟨*proof*⟩

lemma *cfcomp1*: $(f\ oo\ g) = (\Lambda\ x.\ f\cdot(g\cdot x))$
⟨*proof*⟩

lemma *cfcomp2* [*simp*]: $(f\ oo\ g)\cdot x = f\cdot(g\cdot x)$
⟨*proof*⟩

lemma *cfcomp-LAM*: $cont\ g \Longrightarrow f\ oo\ (\Lambda\ x.\ g\ x) = (\Lambda\ x.\ f\cdot(g\ x))$
⟨*proof*⟩

lemma *cfcomp-strict* [*simp*]: $\bot\ oo\ f = \bot$
⟨*proof*⟩

Show that interpretation of (pcpo,-−>-) is a category. The class of objects is interpretation of syntactical class pcpo. The class of arrows between objects $'a$ and $'b$ is interpret. of $'a \to 'b$. The identity arrow is interpretation of *ID*.

The composition of f and g is interpretation of *oo*.

lemma *ID2* [*simp*]: $f \mathit{oo} \mathit{ID} = f$
⟨*proof*⟩

lemma *ID3* [*simp*]: $\mathit{ID} \mathit{oo} f = f$
⟨*proof*⟩

lemma *assoc-oo*: $f \mathit{oo} (g \mathit{oo} h) = (f \mathit{oo} g) \mathit{oo} h$
⟨*proof*⟩

8.10 Strictified functions

defaultsort *pcpo*

definition
$\mathit{strictify} :: ('a \to 'b) \to 'a \to 'b$ **where**
$\mathit{strictify} = (\Lambda\ f\ x.\ \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$

results about strictify

lemma *cont-strictify1*: $\mathit{cont}\ (\lambda f.\ \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
⟨*proof*⟩

lemma *monofun-strictify2*: $\mathit{monofun}\ (\lambda x.\ \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
⟨*proof*⟩

lemma *contlub-strictify2*: $\mathit{contlub}\ (\lambda x.\ \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
⟨*proof*⟩

lemmas *cont-strictify2* =
monocontlub2cont [*OF monofun-strictify2 contlub-strictify2*, *standard*]

lemma *strictify-conv-if*: $\mathit{strictify}{\cdot}f{\cdot}x = (\mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
⟨*proof*⟩

lemma *strictify1* [*simp*]: $\mathit{strictify}{\cdot}f{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *strictify2* [*simp*]: $x \neq \bot \Longrightarrow \mathit{strictify}{\cdot}f{\cdot}x = f{\cdot}x$
⟨*proof*⟩

8.11 Continuous let-bindings

definition
$\mathit{CLet} :: 'a \to ('a \to 'b) \to 'b$ **where**
$\mathit{CLet} = (\Lambda\ s\ f.\ f{\cdot}s)$

syntax
$\text{-}\mathit{CLet} :: [\mathit{letbinds}, 'a] \Rightarrow 'a\ ((\mathit{Let}\ (\text{-})/\ \mathit{in}\ (\text{-}))\ 10)$

translations
-CLet (-binds b bs) e == -CLet b (-CLet bs e)
Let x = a in e == CONST CLet·a·(Λ x. e)

end

9 Deflation: Continuous Deflations and Embedding-Projection Pairs

theory *Deflation*
imports *Cfun*
begin

defaultsort *cpo*

9.1 Continuous deflations

locale *deflation* =
fixes $d :: 'a \to 'a$
assumes *idem*: $\bigwedge x.\ d{\cdot}(d{\cdot}x) = d{\cdot}x$
assumes *less*: $\bigwedge x.\ d{\cdot}x \sqsubseteq x$
begin

lemma *less-ID*: $d \sqsubseteq ID$
⟨*proof*⟩

The set of fixed points is the same as the range.

lemma *fixes-eq-range*: $\{x.\ d{\cdot}x = x\} = range\ (\lambda x.\ d{\cdot}x)$
⟨*proof*⟩

lemma *range-eq-fixes*: $range\ (\lambda x.\ d{\cdot}x) = \{x.\ d{\cdot}x = x\}$
⟨*proof*⟩

The pointwise ordering on deflation functions coincides with the subset ordering of their sets of fixed-points.

lemma *lessI*:
assumes *f*: $\bigwedge x.\ d{\cdot}x = x \Longrightarrow f{\cdot}x = x$ **shows** $d \sqsubseteq f$
⟨*proof*⟩

lemma *lessD*: $[\![f \sqsubseteq d; f{\cdot}x = x]\!] \Longrightarrow d{\cdot}x = x$
⟨*proof*⟩

end

lemma *adm-deflation*: *adm* $(\lambda d.\ deflation\ d)$
⟨*proof*⟩

lemma *deflation-ID*: *deflation ID*
⟨*proof*⟩

lemma *deflation-UU*: *deflation* ⊥
⟨*proof*⟩

lemma *deflation-less-iff*:
⟦*deflation p*; *deflation q*⟧ ⟹ $p \sqsubseteq q \longleftrightarrow (\forall x.\ p{\cdot}x = x \longrightarrow q{\cdot}x = x)$
⟨*proof*⟩

The composition of two deflations is equal to the lesser of the two (if they are comparable).

lemma *deflation-less-comp1*:
assumes *deflation f*
assumes *deflation g*
shows $f \sqsubseteq g \Longrightarrow f{\cdot}(g{\cdot}x) = f{\cdot}x$
⟨*proof*⟩

lemma *deflation-less-comp2*:
⟦*deflation f*; *deflation g*; $f \sqsubseteq g$⟧ ⟹ $g{\cdot}(f{\cdot}x) = f{\cdot}x$
⟨*proof*⟩

9.2 Deflations with finite range

lemma *finite-range-imp-finite-fixes*:
finite (*range f*) ⟹ *finite* $\{x.\ f\,x = x\}$
⟨*proof*⟩

locale *finite-deflation* = *deflation* +
assumes *finite-fixes*: *finite* $\{x.\ d{\cdot}x = x\}$
begin

lemma *finite-range*: *finite* (*range* $(\lambda x.\ d{\cdot}x)$)
⟨*proof*⟩

lemma *finite-image*: *finite* $((\lambda x.\ d{\cdot}x)\ `\ A)$
⟨*proof*⟩

lemma *compact*: *compact* $(d{\cdot}x)$
⟨*proof*⟩

end

9.3 Continuous embedding-projection pairs

locale *ep-pair* =
fixes $e :: 'a \to 'b$ **and** $p :: 'b \to 'a$
assumes *e-inverse* [*simp*]: $\bigwedge x.\ p{\cdot}(e{\cdot}x) = x$

and *e-p-less*: $\bigwedge y.\ e{\cdot}(p{\cdot}y) \sqsubseteq y$
begin

lemma *e-less-iff* [*simp*]: $e{\cdot}x \sqsubseteq e{\cdot}y \longleftrightarrow x \sqsubseteq y$
⟨*proof*⟩

lemma *e-eq-iff* [*simp*]: $e{\cdot}x = e{\cdot}y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *p-eq-iff*:
$[\![e{\cdot}(p{\cdot}x) = x;\ e{\cdot}(p{\cdot}y) = y]\!] \Longrightarrow p{\cdot}x = p{\cdot}y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *p-inverse*: $(\exists\, x.\ y = e{\cdot}x) = (e{\cdot}(p{\cdot}y) = y)$
⟨*proof*⟩

lemma *e-less-iff-less-p*: $e{\cdot}x \sqsubseteq y \longleftrightarrow x \sqsubseteq p{\cdot}y$
⟨*proof*⟩

lemma *compact-e-rev*: *compact* $(e{\cdot}x) \Longrightarrow$ *compact* x
⟨*proof*⟩

lemma *compact-e*: *compact* $x \Longrightarrow$ *compact* $(e{\cdot}x)$
⟨*proof*⟩

lemma *compact-e-iff*: *compact* $(e{\cdot}x) \longleftrightarrow$ *compact* x
⟨*proof*⟩

Deflations from ep-pairs

lemma *deflation-e-p*: *deflation* $(e\ oo\ p)$
⟨*proof*⟩

lemma *deflation-e-d-p*:
assumes *deflation* d
shows *deflation* $(e\ oo\ d\ oo\ p)$
⟨*proof*⟩

lemma *finite-deflation-e-d-p*:
assumes *finite-deflation* d
shows *finite-deflation* $(e\ oo\ d\ oo\ p)$
⟨*proof*⟩

lemma *deflation-p-d-e*:
assumes *deflation* d
assumes d: $\bigwedge x.\ d{\cdot}x \sqsubseteq e{\cdot}(p{\cdot}x)$
shows *deflation* $(p\ oo\ d\ oo\ e)$
⟨*proof*⟩

lemma *finite-deflation-p-d-e*:

assumes *finite-deflation d*
assumes *d*: $\bigwedge x.\ d{\cdot}x \sqsubseteq e{\cdot}(p{\cdot}x)$
shows *finite-deflation* (*p oo d oo e*)
⟨*proof*⟩

end

9.4 Uniqueness of ep-pairs

lemma *ep-pair-unique-e-lemma*:
assumes *ep-pair e1 p* **and** *ep-pair e2 p*
shows $e1 \sqsubseteq e2$
⟨*proof*⟩

lemma *ep-pair-unique-e*:
$[\![$*ep-pair e1 p*; *ep-pair e2 p*$]\!] \Longrightarrow e1 = e2$
⟨*proof*⟩

lemma *ep-pair-unique-p-lemma*:
assumes *ep-pair e p1* **and** *ep-pair e p2*
shows $p1 \sqsubseteq p2$
⟨*proof*⟩

lemma *ep-pair-unique-p*:
$[\![$*ep-pair e p1*; *ep-pair e p2*$]\!] \Longrightarrow p1 = p2$
⟨*proof*⟩

9.5 Composing ep-pairs

lemma *ep-pair-ID-ID*: *ep-pair ID ID*
⟨*proof*⟩

lemma *ep-pair-comp*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*e2 oo e1*) (*p1 oo p2*)
⟨*proof*⟩

locale *pcpo-ep-pair* = *ep-pair* +
constrains *e* :: $'a$::*pcpo* → $'b$::*pcpo*
constrains *p* :: $'b$::*pcpo* → $'a$::*pcpo*
begin

lemma *e-strict* [*simp*]: $e{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *e-defined-iff* [*simp*]: $e{\cdot}x = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *e-defined*: $x \neq \bot \Longrightarrow e{\cdot}x \neq \bot$
⟨*proof*⟩

lemma *p-strict* [*simp*]: $p{\cdot}\bot = \bot$
⟨*proof*⟩

lemmas *stricts* = *e-strict p-strict*

end

end

10 Bifinite: Bifinite domains and approximation

theory *Bifinite*
imports *Deflation*
begin

10.1 Omega-profinite and bifinite domains

class *profinite* =
  fixes *approx* :: *nat* ⇒ $'a \to 'a$
  assumes *chain-approx* [*simp*]: *chain approx*
  assumes *lub-approx-app* [*simp*]: $(\bigsqcup i.\ approx\ i{\cdot}x) = x$
  assumes *approx-idem*: *approx i*·(*approx i*·*x*) = *approx i*·*x*
  assumes *finite-fixes-approx*: *finite* {*x*. *approx i*·*x* = *x*}

class *bifinite* = *profinite* + *pcpo*

lemma *approx-less*: *approx i*·*x* ⊑ *x*
⟨*proof*⟩

lemma *finite-deflation-approx*: *finite-deflation* (*approx i*)
⟨*proof*⟩

interpretation *approx*: *finite-deflation approx i*
⟨*proof*⟩

lemma (**in** *deflation*) *deflation*: *deflation d* ⟨*proof*⟩

lemma *deflation-approx*: *deflation* (*approx i*)
⟨*proof*⟩

lemma *lub-approx* [*simp*]: $(\bigsqcup i.\ approx\ i) = (\Lambda\ x.\ x)$
⟨*proof*⟩

lemma *approx-strict* [*simp*]: *approx i*·⊥ = ⊥
⟨*proof*⟩

lemma *approx-approx1*:

$i \leq j \Longrightarrow$ *approx* $i{\cdot}($*approx* $j{\cdot}x) =$ *approx* $i{\cdot}x$
⟨*proof*⟩

lemma *approx-approx2*:
$j \leq i \Longrightarrow$ *approx* $i{\cdot}($*approx* $j{\cdot}x) =$ *approx* $j{\cdot}x$
⟨*proof*⟩

lemma *approx-approx* [*simp*]:
approx $i{\cdot}($*approx* $j{\cdot}x) =$ *approx* $(min\ i\ j){\cdot}x$
⟨*proof*⟩

lemma *finite-image-approx*: *finite* $((\lambda x.\ approx\ n{\cdot}x)\ `\ A)$
⟨*proof*⟩

lemma *finite-range-approx*: *finite* $(range\ (\lambda x.\ approx\ i{\cdot}x))$
⟨*proof*⟩

lemma *compact-approx* [*simp*]: *compact* $(approx\ n{\cdot}x)$
⟨*proof*⟩

lemma *profinite-compact-eq-approx*: *compact* $x \Longrightarrow \exists\, i.\ approx\ i{\cdot}x = x$
⟨*proof*⟩

lemma *profinite-compact-iff*: *compact* $x \longleftrightarrow (\exists\, n.\ approx\ n{\cdot}x = x)$
⟨*proof*⟩

lemma *approx-induct*:
assumes *adm*: *adm* P **and** P: $\bigwedge n\ x.\ P\ (approx\ n{\cdot}x)$
shows $P\ x$
⟨*proof*⟩

lemma *profinite-less-ext*: $(\bigwedge i.\ approx\ i{\cdot}x \sqsubseteq approx\ i{\cdot}y) \Longrightarrow x \sqsubseteq y$
⟨*proof*⟩

10.2 Instance for continuous function space

lemma *finite-range-cfun-lemma*:
assumes *a*: *finite* $(range\ (\lambda x.\ a{\cdot}x))$
assumes *b*: *finite* $(range\ (\lambda y.\ b{\cdot}y))$
shows *finite* $(range\ (\lambda f.\ \Lambda\ x.\ b{\cdot}(f{\cdot}(a{\cdot}x))))$ (**is** *finite* $(range\ ?h)$)
⟨*proof*⟩

instantiation $->$ :: (*profinite*, *profinite*) *profinite*
begin

definition
approx-cfun-def:
approx $= (\lambda n.\ \Lambda\ f\ x.\ approx\ n{\cdot}(f{\cdot}(approx\ n{\cdot}x)))$

instance ⟨*proof*⟩

end

instance −> :: (*profinite*, *bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-cfun*: *approx n·f·x* = *approx n·*(*f·*(*approx n·x*))
⟨*proof*⟩

end

11 Cprod: The cpo of cartesian products

theory *Cprod*
imports *Bifinite*
begin

defaultsort *cpo*

11.1 Type *unit* is a pcpo

definition
unit-when :: ′*a* → *unit* → ′*a* **where**
unit-when = (Λ *a* -. *a*)

translations
Λ(). *t* == *CONST unit-when·t*

lemma *unit-when* [*simp*]: *unit-when·a·u* = *a*
⟨*proof*⟩

11.2 Continuous versions of constants

definition
cpair :: ′*a* → ′*b* → (′*a* ∗ ′*b*) — continuous pairing **where**
cpair = (Λ *x y*. (*x*, *y*))

definition
cfst :: (′*a* ∗ ′*b*) → ′*a* **where**
cfst = (Λ *p*. *fst p*)

definition
csnd :: (′*a* ∗ ′*b*) → ′*b* **where**
csnd = (Λ *p*. *snd p*)

definition
csplit :: (′*a* → ′*b* → ′*c*) → (′*a* ∗ ′*b*) → ′*c* **where**
csplit = (Λ *f p*. *f·*(*cfst·p*)·(*csnd·p*))

syntax
-ctuple :: $['a, args] \Rightarrow 'a * 'b$ $((1<\text{-},/\ \text{->}))$

syntax (*xsymbols*)
-ctuple :: $['a, args] \Rightarrow 'a * 'b$ $((1\langle\text{-},/\ \text{-}\rangle))$

translations
$\langle x,\ y,\ z\rangle == \langle x,\ \langle y,\ z\rangle\rangle$
$\langle x,\ y\rangle \quad == CONST\ cpair{\cdot}x{\cdot}y$

translations
$\Lambda(CONST\ cpair{\cdot}x{\cdot}y).\ t == CONST\ csplit{\cdot}(\Lambda\ x\ y.\ t)$

11.3 Convert all lemmas to the continuous versions

lemma *cpair-eq-pair*: $<x,\ y> = (x,\ y)$
⟨*proof*⟩

lemma *pair-eq-cpair*: $(x,\ y) = <x,\ y>$
⟨*proof*⟩

lemma *inject-cpair*: $<a,b> = <aa,ba> \Longrightarrow a = aa \wedge b = ba$
⟨*proof*⟩

lemma *cpair-eq* [*iff*]: $(<a,\ b> = <a',\ b'>) = (a = a' \wedge b = b')$
⟨*proof*⟩

lemma *cpair-less* [*iff*]: $(<a,\ b> \sqsubseteq <a',\ b'>) = (a \sqsubseteq a' \wedge b \sqsubseteq b')$
⟨*proof*⟩

lemma *cpair-defined-iff* [*iff*]: $(<x,\ y> = \bot) = (x = \bot \wedge y = \bot)$
⟨*proof*⟩

lemma *cpair-strict* [*simp*]: $\langle\bot,\ \bot\rangle = \bot$
⟨*proof*⟩

lemma *inst-cprod-pcpo2*: $\bot = <\bot,\ \bot>$
⟨*proof*⟩

lemma *defined-cpair-rev*:
$<a,b> = \bot \Longrightarrow a = \bot \wedge b = \bot$
⟨*proof*⟩

lemma *Exh-Cprod2*: $\exists\, a\ b.\ z = <a,\ b>$
⟨*proof*⟩

lemma *cprodE*: $[\![\bigwedge x\ y.\ p = <x,\ y> \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *cfst-cpair* [*simp*]: $cfst{\cdot}<x,\ y> = x$
⟨*proof*⟩

lemma *csnd-cpair* [*simp*]: $csnd{\cdot}<x,\ y> = y$
⟨*proof*⟩

lemma *cfst-strict* [*simp*]: $cfst{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *csnd-strict* [*simp*]: $csnd{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *cpair-cfst-csnd*: $\langle cfst{\cdot}p,\ csnd{\cdot}p\rangle = p$
⟨*proof*⟩

lemmas *surjective-pairing-Cprod2* = *cpair-cfst-csnd*

lemma *less-cprod*: $x \sqsubseteq y = (cfst{\cdot}x \sqsubseteq cfst{\cdot}y \wedge csnd{\cdot}x \sqsubseteq csnd{\cdot}y)$
⟨*proof*⟩

lemma *eq-cprod*: $(x = y) = (cfst{\cdot}x = cfst{\cdot}y \wedge csnd{\cdot}x = csnd{\cdot}y)$
⟨*proof*⟩

lemma *cfst-less-iff*: $cfst{\cdot}x \sqsubseteq y = x \sqsubseteq <y,\ csnd{\cdot}x>$
⟨*proof*⟩

lemma *csnd-less-iff*: $csnd{\cdot}x \sqsubseteq y = x \sqsubseteq <cfst{\cdot}x,\ y>$
⟨*proof*⟩

lemma *compact-cfst*: $compact\ x \Longrightarrow compact\ (cfst{\cdot}x)$
⟨*proof*⟩

lemma *compact-csnd*: $compact\ x \Longrightarrow compact\ (csnd{\cdot}x)$
⟨*proof*⟩

lemma *compact-cpair*: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact\ <x,\ y>$
⟨*proof*⟩

lemma *compact-cpair-iff* [*simp*]: $compact\ <x,\ y> = (compact\ x \wedge compact\ y)$
⟨*proof*⟩

lemma *lub-cprod2*:
$chain\ S \Longrightarrow range\ S <<|\ <\bigsqcup i.\ cfst{\cdot}(S\ i),\ \bigsqcup i.\ csnd{\cdot}(S\ i)>$
⟨*proof*⟩

lemma *thelub-cprod2*:
$chain\ S \Longrightarrow (\bigsqcup i.\ S\ i) = <\bigsqcup i.\ cfst{\cdot}(S\ i),\ \bigsqcup i.\ csnd{\cdot}(S\ i)>$
⟨*proof*⟩

lemma *csplit1* [*simp*]: $csplit{\cdot}f{\cdot}\bot = f{\cdot}\bot{\cdot}\bot$
⟨*proof*⟩

lemma *csplit2* [*simp*]: $csplit{\cdot}f{\cdot}{<}x,y{>} = f{\cdot}x{\cdot}y$
⟨*proof*⟩

lemma *csplit3* [*simp*]: $csplit{\cdot}cpair{\cdot}z = z$
⟨*proof*⟩

lemmas *Cprod-rews = cfst-cpair csnd-cpair csplit2*

11.4 Product type is a bifinite domain

instantiation $*$ :: (*profinite*, *profinite*) *profinite*
begin

definition
approx-cprod-def:
$approx = (\lambda n.\ \Lambda\langle x,\ y\rangle.\ \langle approx\ n{\cdot}x,\ approx\ n{\cdot}y\rangle)$

instance ⟨*proof*⟩

end

instance $*$ :: (*bifinite*, *bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-cpair* [*simp*]:
$approx\ i{\cdot}\langle x,\ y\rangle = \langle approx\ i{\cdot}x,\ approx\ i{\cdot}y\rangle$
⟨*proof*⟩

lemma *cfst-approx*: $cfst{\cdot}(approx\ i{\cdot}p) = approx\ i{\cdot}(cfst{\cdot}p)$
⟨*proof*⟩

lemma *csnd-approx*: $csnd{\cdot}(approx\ i{\cdot}p) = approx\ i{\cdot}(csnd{\cdot}p)$
⟨*proof*⟩

end

12 Discrete: Discrete cpo types

theory *Discrete*
imports *Cont*
begin

datatype $'a\ discr = Discr\ 'a$ :: *type*

12.1 Type *'a discr* is a discrete cpo

instantiation *discr* :: (*type*) *sq-ord*
begin

definition
less-discr-def:
$(op \sqsubseteq :: 'a\ discr \Rightarrow 'a\ discr \Rightarrow bool) = (op =)$

instance ⟨*proof*⟩
end

instance *discr* :: (*type*) *discrete-cpo*
⟨*proof*⟩

lemma *discr-less-eq* [*iff*]: $((x::('a::type)discr) << y) = (x = y)$
⟨*proof*⟩

12.2 Type *'a discr* is a cpo

lemma *discr-chain0*:
$!!S::nat=>('a::type)discr.\ chain\ S ==> S\ i = S\ 0$
⟨*proof*⟩

lemma *discr-chain-range0* [*simp*]:
$!!S::nat=>('a::type)discr.\ chain(S) ==> range(S) = \{S\ 0\}$
⟨*proof*⟩

instance *discr* :: (*finite*) *finite-po*
⟨*proof*⟩

instance *discr* :: (*type*) *chfin*
⟨*proof*⟩

12.3 *undiscr*

definition
$undiscr :: ('a::type)discr => 'a$ **where**
$undiscr\ x = (case\ x\ of\ Discr\ y => y)$

lemma *undiscr-Discr* [*simp*]: $undiscr\ (Discr\ x) = x$
⟨*proof*⟩

lemma *Discr-undiscr* [*simp*]: $Discr\ (undiscr\ y) = y$
⟨*proof*⟩

lemma *discr-chain-f-range0*:
$!!S::nat=>('a::type)discr.\ chain(S) ==> range(\%i.\ f(S\ i)) = \{f(S\ 0)\}$
⟨*proof*⟩

lemma *cont-discr* [*iff*]: *cont* (%*x*::('*a*::*type*)*discr*. *f x*)
⟨*proof*⟩

end

13 Up: The type of lifted values

theory *Up*
imports *Bifinite*
begin

defaultsort *cpo*

13.1 Definition of new type for lifting

datatype '*a u* = *Ibottom* | *Iup* '*a*

syntax (*xsymbols*)
u :: *type* ⇒ *type* ((-⊥) [*1000*] *999*)

consts
Ifup :: ('*a* → '*b*::*pcpo*) ⇒ '*a u* ⇒ '*b*

primrec
Ifup f Ibottom = ⊥
Ifup f (*Iup x*) = *f*·*x*

13.2 Ordering on lifted cpo

instantiation *u* :: (*cpo*) *sq-ord*
begin

definition
less-up-def:
(*op* ⊑) ≡ (λ*x y*. *case x of Ibottom* ⇒ *True* | *Iup a* ⇒
(*case y of Ibottom* ⇒ *False* | *Iup b* ⇒ *a* ⊑ *b*))

instance ⟨*proof*⟩
end

lemma *minimal-up* [*iff*]: *Ibottom* ⊑ *z*
⟨*proof*⟩

lemma *not-Iup-less* [*iff*]: ¬ *Iup x* ⊑ *Ibottom*
⟨*proof*⟩

lemma *Iup-less* [*iff*]: (*Iup x* ⊑ *Iup y*) = (*x* ⊑ *y*)
⟨*proof*⟩

13.3 Lifted cpo is a partial order

instance $u :: (cpo)\ po$
⟨*proof*⟩

lemma *u-UNIV*: $UNIV = insert\ Ibottom\ (range\ Iup)$
⟨*proof*⟩

instance $u :: (finite\text{-}po)\ finite\text{-}po$
⟨*proof*⟩

13.4 Lifted cpo is a cpo

lemma *is-lub-Iup*:
$range\ S <<|\ x \Longrightarrow range\ (\lambda i.\ Iup\ (S\ i)) <<|\ Iup\ x$
⟨*proof*⟩

Now some lemmas about chains of $'a_\perp$ elements

lemma *up-lemma1*: $z \neq Ibottom \Longrightarrow Iup\ (THE\ a.\ Iup\ a = z) = z$
⟨*proof*⟩

lemma *up-lemma2*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow Y\ (i + j) \neq Ibottom$
⟨*proof*⟩

lemma *up-lemma3*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow Iup\ (THE\ a.\ Iup\ a = Y\ (i + j)) = Y\ (i + j)$
⟨*proof*⟩

lemma *up-lemma4*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow chain\ (\lambda i.\ THE\ a.\ Iup\ a = Y\ (i + j))$
⟨*proof*⟩

lemma *up-lemma5*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow$
$(\lambda i.\ Y\ (i + j)) = (\lambda i.\ Iup\ (THE\ a.\ Iup\ a = Y\ (i + j)))$
⟨*proof*⟩

lemma *up-lemma6*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!]$
$\Longrightarrow range\ Y <<|\ Iup\ (\bigsqcup i.\ THE\ a.\ Iup\ a = Y(i + j))$
⟨*proof*⟩

lemma *up-chain-lemma*:
$chain\ Y \Longrightarrow$
$(\exists A.\ chain\ A \wedge (\bigsqcup i.\ Y\ i) = Iup\ (\bigsqcup i.\ A\ i) \wedge$
$(\exists j.\ \forall i.\ Y\ (i + j) = Iup\ (A\ i))) \vee (Y = (\lambda i.\ Ibottom))$
⟨*proof*⟩

lemma *cpo-up*: $chain\ (Y{::}nat \Rightarrow 'a\ u) \Longrightarrow \exists x.\ range\ Y <<|\ x$

⟨*proof*⟩

instance $u :: (cpo)\ cpo$
⟨*proof*⟩

13.5 Lifted cpo is pointed

lemma *least-up*: $\exists\, x::'a\ u.\ \forall\, y.\ x \sqsubseteq y$
⟨*proof*⟩

instance $u :: (cpo)\ pcpo$
⟨*proof*⟩

for compatibility with old HOLCF-Version

lemma *inst-up-pcpo*: $\bot = Ibottom$
⟨*proof*⟩

13.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

lemma *cont-Iup*: *cont Iup*
⟨*proof*⟩

continuity for *Ifup*

lemma *cont-Ifup1*: *cont* $(\lambda f.\ Ifup\ f\ x)$
⟨*proof*⟩

lemma *monofun-Ifup2*: *monofun* $(\lambda x.\ Ifup\ f\ x)$
⟨*proof*⟩

lemma *cont-Ifup2*: *cont* $(\lambda x.\ Ifup\ f\ x)$
⟨*proof*⟩

13.7 Continuous versions of constants

definition
up :: $'a \to 'a\ u$ **where**
$up = (\Lambda\ x.\ Iup\ x)$

definition
fup :: $('a \to 'b::pcpo) \to 'a\ u \to 'b$ **where**
$fup = (\Lambda\ f\ p.\ Ifup\ f\ p)$

translations
case l of XCONST up·x ⇒ *t* == *CONST fup·*$(\Lambda\ x.\ t)$*·l*
$\Lambda(XCONST\ up{\cdot}x).\ t ==$ *CONST fup·*$(\Lambda\ x.\ t)$

continuous versions of lemmas for $'a_\bot$

lemma *Exh-Up*: $z = \bot \vee (\exists\, x.\ z = up{\cdot}x)$
⟨*proof*⟩

lemma *up-eq* [*simp*]: $(up{\cdot}x = up{\cdot}y) = (x = y)$
⟨*proof*⟩

lemma *up-inject*: $up{\cdot}x = up{\cdot}y \Longrightarrow x = y$
⟨*proof*⟩

lemma *up-defined* [*simp*]: $up{\cdot}x \neq \bot$
⟨*proof*⟩

lemma *not-up-less-UU*: $\neg\ up{\cdot}x \sqsubseteq \bot$
⟨*proof*⟩

lemma *up-less* [*simp*]: $(up{\cdot}x \sqsubseteq up{\cdot}y) = (x \sqsubseteq y)$
⟨*proof*⟩

lemma *upE* [*cases type*: *u*]: $[\![p = \bot \Longrightarrow Q;\ \bigwedge x.\ p = up{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *up-induct* [*induct type*: *u*]: $[\![P\ \bot;\ \bigwedge x.\ P\ (up{\cdot}x)]\!] \Longrightarrow P\ x$
⟨*proof*⟩

lifting preserves chain-finiteness

lemma *up-chain-cases*:
$chain\ Y \Longrightarrow$
$(\exists\, A.\ chain\ A \wedge (\bigsqcup i.\ Y\ i) = up{\cdot}(\bigsqcup i.\ A\ i) \wedge$
$(\exists\, j.\ \forall\, i.\ Y\ (i + j) = up{\cdot}(A\ i))) \vee Y = (\lambda i.\ \bot)$
⟨*proof*⟩

lemma *compact-up*: $compact\ x \Longrightarrow compact\ (up{\cdot}x)$
⟨*proof*⟩

lemma *compact-upD*: $compact\ (up{\cdot}x) \Longrightarrow compact\ x$
⟨*proof*⟩

lemma *compact-up-iff* [*simp*]: $compact\ (up{\cdot}x) = compact\ x$
⟨*proof*⟩

instance $u :: (chfin)\ chfin$
⟨*proof*⟩

properties of fup

lemma *fup1* [*simp*]: $fup{\cdot}f{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *fup2* [*simp*]: $fup{\cdot}f{\cdot}(up{\cdot}x) = f{\cdot}x$
⟨*proof*⟩

lemma *fup3* [*simp*]: *fup*·*up*·*x* = *x*
⟨*proof*⟩

13.8 Lifted cpo is a bifinite domain

instantiation *u* :: (*profinite*) *bifinite*
begin

definition
approx-up-def:
approx = (λ*n*. *fup*·(Λ *x*. *up*·(*approx n*·*x*)))

instance ⟨*proof*⟩

end

lemma *approx-up* [*simp*]: *approx i*·(*up*·*x*) = *up*·(*approx i*·*x*)
⟨*proof*⟩

end

14 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports
~~/*src*/*HOL*/*List*
~~/*src*/*HOL*/*Hilbert-Choice*
~~/*src*/*HOL*/*Nat-Int-Bij*
~~/*src*/*HOL*/*Rational*
Main
begin

14.1 The class of countable types

class *countable* =
assumes *ex-inj*: ∃ *to-nat* :: *'a* ⇒ *nat*. *inj to-nat*

lemma *countable-classI*:
fixes *f* :: *'a* ⇒ *nat*
assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *OFCLASS*(*'a*, *countable-class*)
⟨*proof*⟩

14.2 Conversion functions

definition *to-nat* :: *'a*::*countable* ⇒ *nat* **where**

to-nat = (*SOME f*. *inj f*)

definition *from-nat* :: *nat* ⇒ *'a*::*countable* **where**
from-nat = *inv* (*to-nat* :: *'a* ⇒ *nat*)

lemma *inj-to-nat* [*simp*]: *inj to-nat*
⟨*proof*⟩

lemma *surj-from-nat* [*simp*]: *surj from-nat*
⟨*proof*⟩

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat x*) = *x*
⟨*proof*⟩

14.3 Countable types

instance *nat* :: *countable*
⟨*proof*⟩

subclass (**in** *finite*) *countable*
⟨*proof*⟩

Pairs

primrec *sum* :: *nat* ⇒ *nat*
where
sum 0 = *0*
| *sum* (*Suc n*) = *Suc n* + *sum n*

lemma *sum-arith*: *sum n* = *n* ∗ *Suc n div 2*
⟨*proof*⟩

lemma *sum-mono*: *n* ≥ *m* ⟹ *sum n* ≥ *sum m*
⟨*proof*⟩

definition
pair-encode = (λ(*m*, *n*). *sum* (*m* + *n*) + *m*)

lemma *inj-pair-cencode*: *inj pair-encode*
⟨*proof*⟩

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

Sums

instance +:: (*countable*, *countable*) *countable*

⟨*proof*⟩

Integers

lemma *int-cases*: $(i::int) = 0 \vee i < 0 \vee i > 0$
⟨*proof*⟩

lemma *int-pos-neg-zero*:
obtains (*zero*) $(z::int) = 0$ *sgn* $z = 0$ *abs* $z = 0$
| (*pos*) n **where** $z =$ *of-nat* n *sgn* $z = 1$ *abs* $z =$ *of-nat* n
| (*neg*) n **where** $z = -$ (*of-nat* n) *sgn* $z = -1$ *abs* $z =$ *of-nat* n
⟨*proof*⟩

instance *int* :: *countable*
⟨*proof*⟩

Options

instance *option* :: (*countable*) *countable*
⟨*proof*⟩

Lists

lemma *from-nat-to-nat-map* [*simp*]: *map from-nat* (*map to-nat xs*) = *xs*
⟨*proof*⟩

primrec
list-encode :: $'a::countable\ list \Rightarrow nat$
where
list-encode [] = 0
| *list-encode* (*x*#*xs*) = *Suc* (*to-nat* (*x*, *list-encode xs*))

instance *list* :: (*countable*) *countable*
⟨*proof*⟩

Functions

instance *fun* :: (*finite*, *countable*) *countable*
⟨*proof*⟩

14.4 The Rationals are Countably Infinite

definition *nat-to-rat-surj* :: *nat* ⇒ *rat* **where**
nat-to-rat-surj n = (*let* (*a*,*b*) = *nat-to-nat2 n*
in Fract (*nat-to-int-bij a*) (*nat-to-int-bij b*))

lemma *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*
⟨*proof*⟩

lemma *Rats-eq-range-nat-to-rat-surj*: $\mathbb{Q}$ = *range nat-to-rat-surj*
⟨*proof*⟩

context *field-char-0*

begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
$\mathbb{Q} = range\ (of\text{-}rat\ o\ nat\text{-}to\text{-}rat\text{-}surj)$
⟨*proof*⟩

lemma *surj-of-rat-nat-to-rat-surj*:
$r \in \mathbb{Q} \Longrightarrow \exists\, n.\ r = of\text{-}rat(nat\text{-}to\text{-}rat\text{-}surj\ n)$
⟨*proof*⟩

end

instance *rat* :: *countable*
⟨*proof*⟩

end

15 Lift: Lifting types of class type to flat pcpo's

theory *Lift*
imports *Discrete Up Countable*
begin

defaultsort *type*

pcpodef *'a lift* = *UNIV* :: *'a discr u set*
⟨*proof*⟩

instance *lift* :: (*finite*) *finite-po*
⟨*proof*⟩

lemmas *inst-lift-pcpo* = *Abs-lift-strict* [*symmetric*]

definition
Def :: *'a* ⇒ *'a lift* **where**
Def x = *Abs-lift* (*up*·(*Discr x*))

15.1 Lift as a datatype

lemma *lift-induct*: $[\![P\ \bot;\ \bigwedge x.\ P\ (Def\ x)]\!] \Longrightarrow P\ y$
⟨*proof*⟩

rep-datatype ⊥::*'a lift Def*
⟨*proof*⟩

lemmas *lift-distinct1* = *lift.distinct*(*1*)
lemmas *lift-distinct2* = *lift.distinct*(*2*)
lemmas *Def-not-UU* = *lift.distinct*(*2*)

lemmas *Def-inject* = *lift.inject*

$\bot$ and *Def*

lemma *Lift-exhaust*: $x = \bot \vee (\exists\, y.\ x = Def\ y)$
⟨*proof*⟩

lemma *Lift-cases*: $[\![x = \bot \Longrightarrow P;\ \exists\, a.\ x = Def\ a \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *not-Undef-is-Def*: $(x \neq \bot) = (\exists\, y.\ x = Def\ y)$
⟨*proof*⟩

lemma *lift-definedE*: $[\![x \neq \bot;\ \bigwedge a.\ x = Def\ a \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

For $x \neq \bot$ in assumptions *defined* replaces x by *Def* a in conclusion.

⟨*ML*⟩

lemma *DefE*: $Def\ x = \bot \Longrightarrow R$
⟨*proof*⟩

lemma *DefE2*: $[\![x = Def\ s;\ x = \bot]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *Def-inject-less-eq*: $Def\ x \sqsubseteq Def\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *Def-less-is-eq* [*simp*]: $Def\ x \sqsubseteq y \longleftrightarrow Def\ x = y$
⟨*proof*⟩

15.2 Lift is flat

instance *lift* :: (*type*) *flat*
⟨*proof*⟩

Two specific lemmas for the combination of LCF and HOL terms.

lemma *cont-Rep-CFun-app* [*simp*]: $[\![cont\ g;\ cont\ f]\!] \Longrightarrow cont(\lambda x.\ ((f\ x)\cdot(g\ x))\ s)$
⟨*proof*⟩

lemma *cont-Rep-CFun-app-app* [*simp*]: $[\![cont\ g;\ cont\ f]\!] \Longrightarrow cont(\lambda x.\ ((f\ x)\cdot(g\ x))\ s\ t)$
⟨*proof*⟩

15.3 Further operations

definition
flift1 :: $('a \Rightarrow 'b{::}pcpo) \Rightarrow ('a\ lift \rightarrow 'b)$ (**binder** *FLIFT* 10) **where**
flift1 = $(\lambda f.\ (\Lambda\ x.\ lift\text{-}case\ \bot\ f\ x))$

definition
flift2 :: $('a \Rightarrow 'b) \Rightarrow ('a\ lift \to 'b\ lift)$ **where**
flift2 $f = (FLIFT\ x.\ Def\ (f\ x))$

15.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

lemma *cont-lift-case1*: $cont\ (\lambda f.\ lift\text{-}case\ a\ f\ x)$
⟨*proof*⟩

lemma *cont-lift-case2*: $cont\ (\lambda x.\ lift\text{-}case\ \bot\ f\ x)$
⟨*proof*⟩

lemma *cont-flift1*: *cont flift1*
⟨*proof*⟩

lemma *FLIFT-mono*:
$(\bigwedge x.\ f\ x \sqsubseteq g\ x) \Longrightarrow (FLIFT\ x.\ f\ x) \sqsubseteq (FLIFT\ x.\ g\ x)$
⟨*proof*⟩

lemma *cont2cont-flift1* [*simp*]:
$[\![\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)]\!] \Longrightarrow cont\ (\lambda x.\ FLIFT\ y.\ f\ x\ y)$
⟨*proof*⟩

lemma *cont2cont-lift-case* [*simp*]:
$[\![\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y);\ cont\ g]\!] \Longrightarrow cont\ (\lambda x.\ lift\text{-}case\ UU\ (f\ x)\ (g\ x))$
⟨*proof*⟩

rewrites for *flift1*, *flift2*

lemma *flift1-Def* [*simp*]: *flift1* $f{\cdot}(Def\ x) = (f\ x)$
⟨*proof*⟩

lemma *flift2-Def* [*simp*]: *flift2* $f{\cdot}(Def\ x) = Def\ (f\ x)$
⟨*proof*⟩

lemma *flift1-strict* [*simp*]: *flift1* $f{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *flift2-strict* [*simp*]: *flift2* $f{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *flift2-defined* [*simp*]: $x \neq \bot \Longrightarrow$ (*flift2* $f){\cdot}x \neq \bot$
⟨*proof*⟩

lemma *flift2-defined-iff* [*simp*]: (*flift2* $f{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

Extension of *cont-tac* and installation of simplifier.

lemmas *cont-lemmas-ext* =
cont2cont-flift1 cont2cont-lift-case cont2cont-lambda
cont-Rep-CFun-app cont-Rep-CFun-app-app cont-if

⟨*ML*⟩

15.5 Lifted countable types are bifinite

instantiation *lift* :: (*countable*) *bifinite*
begin

definition
approx-lift-def:
approx = (λ*n*. *FLIFT x*. *if to-nat x* < *n then Def x else* ⊥)

instance ⟨*proof*⟩

end

end

16 Tr: The type of lifted booleans

theory *Tr*
imports *Lift*
begin

16.1 Type definition and constructors

types
tr = *bool lift*

translations
tr <= (*type*) *bool lift*

definition
TT :: *tr* **where**
TT = *Def True*

definition
FF :: *tr* **where**
FF = *Def False*

Exhaustion and Elimination for type *tr*

lemma *Exh-tr*: $t = \bot \vee t = TT \vee t = FF$
⟨*proof*⟩

lemma *trE*: $[\![p = \bot \Longrightarrow Q;\ p = TT \Longrightarrow Q;\ p = FF \Longrightarrow Q]\!] \Longrightarrow Q$

⟨*proof*⟩

lemma *tr-induct*: ⟦*P* ⊥; *P TT*; *P FF*⟧ ⟹ *P x*
⟨*proof*⟩

distinctness for type *tr*

lemma *dist-less-tr* [*simp*]:
¬ *TT* ⊑ ⊥ ¬ *FF* ⊑ ⊥ ¬ *TT* ⊑ *FF* ¬ *FF* ⊑ *TT*
⟨*proof*⟩

lemma *dist-eq-tr* [*simp*]:
TT ≠ ⊥ *FF* ≠ ⊥ *TT* ≠ *FF* ⊥ ≠ *TT* ⊥ ≠ *FF FF* ≠ *TT*
⟨*proof*⟩

lemma *TT-less-iff* [*simp*]: *TT* ⊑ *x* ⟷ *x* = *TT*
⟨*proof*⟩

lemma *FF-less-iff* [*simp*]: *FF* ⊑ *x* ⟷ *x* = *FF*
⟨*proof*⟩

lemma *not-less-TT-iff* [*simp*]: ¬ (*x* ⊑ *TT*) ⟷ *x* = *FF*
⟨*proof*⟩

lemma *not-less-FF-iff* [*simp*]: ¬ (*x* ⊑ *FF*) ⟷ *x* = *TT*
⟨*proof*⟩

16.2 Case analysis

defaultsort *pcpo*

definition
trifte :: *'c* → *'c* → *tr* → *'c* **where**
ifte-def: *trifte* = (Λ *t e*. *FLIFT b*. *if b then t else e*)
abbreviation
cifte-syn :: [*tr*, *'c*, *'c*] ⇒ *'c* (*3If -/ (then -/ else -) fi*) *60*) **where**
If b then e1 else e2 fi == *trifte*·*e1*·*e2*·*b*

translations
Λ (*XCONST TT*). *t* == *CONST trifte*·*t*·⊥
Λ (*XCONST FF*). *t* == *CONST trifte*·⊥·*t*

lemma *ifte-thms* [*simp*]:
If ⊥ *then e1 else e2 fi* = ⊥
If FF then e1 else e2 fi = *e2*
If TT then e1 else e2 fi = *e1*
⟨*proof*⟩

16.3 Boolean connectives

definition

trand :: *tr* → *tr* → *tr* **where**
andalso-def: *trand* = (Λ *x y*. *If x then y else FF fi*)
abbreviation
andalso-syn :: *tr* ⇒ *tr* ⇒ *tr* (- *andalso* - [*36*,*35*] *35*) **where**
x andalso y == *trand*·*x*·*y*

definition
tror :: *tr* → *tr* → *tr* **where**
orelse-def: *tror* = (Λ *x y*. *If x then TT else y fi*)
abbreviation
orelse-syn :: *tr* ⇒ *tr* ⇒ *tr* (- *orelse* - [*31*,*30*] *30*) **where**
x orelse y == *tror*·*x*·*y*

definition
neg :: *tr* → *tr* **where**
neg = *flift2 Not*

definition
If2 :: [*tr*, *'c*, *'c*] ⇒ *'c* **where**
If2 Q x y = (*If Q then x else y fi*)

tactic for tr-thms with case split

lemmas *tr-defs* = *andalso-def orelse-def neg-def ifte-def TT-def FF-def*

lemmas about andalso, orelse, neg and if

lemma *andalso-thms* [*simp*]:
(*TT andalso y*) = *y*
(*FF andalso y*) = *FF*
(⊥ *andalso y*) = ⊥
(*y andalso TT*) = *y*
(*y andalso y*) = *y*
⟨*proof*⟩

lemma *orelse-thms* [*simp*]:
(*TT orelse y*) = *TT*
(*FF orelse y*) = *y*
(⊥ *orelse y*) = ⊥
(*y orelse FF*) = *y*
(*y orelse y*) = *y*
⟨*proof*⟩

lemma *neg-thms* [*simp*]:
neg·*TT* = *FF*
neg·*FF* = *TT*
neg·⊥ = ⊥
⟨*proof*⟩

split-tac for If via If2 because the constant has to be a constant

lemma *split-If2*:

$P\ (If2\ Q\ x\ y) = ((Q = \bot \longrightarrow P\ \bot) \wedge (Q = TT \longrightarrow P\ x) \wedge (Q = FF \longrightarrow P\ y))$
⟨*proof*⟩

⟨*ML*⟩

16.4 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*:
$t \neq \bot \Longrightarrow ((t\ andalso\ s) = FF) = (t = FF \vee s = FF)$
⟨*proof*⟩

lemma *andalso-and*:
$t \neq \bot \Longrightarrow ((t\ andalso\ s) \neq FF) = (t \neq FF \wedge s \neq FF)$
⟨*proof*⟩

lemma *Def-bool1* [*simp*]: $(Def\ x \neq FF) = x$
⟨*proof*⟩

lemma *Def-bool2* [*simp*]: $(Def\ x = FF) = (\neg\ x)$
⟨*proof*⟩

lemma *Def-bool3* [*simp*]: $(Def\ x = TT) = x$
⟨*proof*⟩

lemma *Def-bool4* [*simp*]: $(Def\ x \neq TT) = (\neg\ x)$
⟨*proof*⟩

lemma *If-and-if*:
$(If\ Def\ P\ then\ A\ else\ B\ fi) = (if\ P\ then\ A\ else\ B)$
⟨*proof*⟩

16.5 Compactness

lemma *compact-TT*: *compact TT*
⟨*proof*⟩

lemma *compact-FF*: *compact FF*
⟨*proof*⟩

end

17 Ssum: The type of strict sums

theory *Ssum*
imports *Cprod Tr*
begin

defaultsort *pcpo*

17.1 Definition of strict sum type

pcpodef (*Ssum*) ('*a*, '*b*) ++ (**infixr** ++ *10*) =
  {*p* :: *tr* × ('*a* × '*b*).
    (*cfst*·*p* ⊑ *TT* ⟷ *csnd*·(*csnd*·*p*) = ⊥) ∧
    (*cfst*·*p* ⊑ *FF* ⟷ *cfst*·(*csnd*·*p*) = ⊥)}
⟨*proof*⟩

instance ++ :: ({*finite-po*,*pcpo*}, {*finite-po*,*pcpo*}) *finite-po*
⟨*proof*⟩

instance ++ :: ({*chfin*,*pcpo*}, {*chfin*,*pcpo*}) *chfin*
⟨*proof*⟩

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

17.2 Definitions of constructors

definition
  sinl :: '*a* → ('*a* ++ '*b*) **where**
  sinl = (Λ *a*. *Abs-Ssum* <*strictify*·(Λ -. *TT*)·*a*, *a*, ⊥>)

definition
  sinr :: '*b* → ('*a* ++ '*b*) **where**
  sinr = (Λ *b*. *Abs-Ssum* <*strictify*·(Λ -. *FF*)·*b*, ⊥, *b*>)

lemma *sinl-Ssum*: <*strictify*·(Λ -. *TT*)·*a*, *a*, ⊥> ∈ *Ssum*
⟨*proof*⟩

lemma *sinr-Ssum*: <*strictify*·(Λ -. *FF*)·*b*, ⊥, *b*> ∈ *Ssum*
⟨*proof*⟩

lemma *sinl-Abs-Ssum*: *sinl*·*a* = *Abs-Ssum* <*strictify*·(Λ -. *TT*)·*a*, *a*, ⊥>
⟨*proof*⟩

lemma *sinr-Abs-Ssum*: *sinr*·*b* = *Abs-Ssum* <*strictify*·(Λ -. *FF*)·*b*, ⊥, *b*>
⟨*proof*⟩

lemma *Rep-Ssum-sinl*: *Rep-Ssum* (*sinl*·*a*) = <*strictify*·(Λ -. *TT*)·*a*, *a*, ⊥>
⟨*proof*⟩

lemma *Rep-Ssum-sinr*: *Rep-Ssum* (*sinr*·*b*) = <*strictify*·(Λ -. *FF*)·*b*, ⊥, *b*>
⟨*proof*⟩

17.3 Properties of *sinl* and *sinr*

Ordering

lemma *sinl-less* [*simp*]: $(sinl{\cdot}x \sqsubseteq sinl{\cdot}y) = (x \sqsubseteq y)$
⟨*proof*⟩

lemma *sinr-less* [*simp*]: $(sinr{\cdot}x \sqsubseteq sinr{\cdot}y) = (x \sqsubseteq y)$
⟨*proof*⟩

lemma *sinl-less-sinr* [*simp*]: $(sinl{\cdot}x \sqsubseteq sinr{\cdot}y) = (x = \bot)$
⟨*proof*⟩

lemma *sinr-less-sinl* [*simp*]: $(sinr{\cdot}x \sqsubseteq sinl{\cdot}y) = (x = \bot)$
⟨*proof*⟩

Equality

lemma *sinl-eq* [*simp*]: $(sinl{\cdot}x = sinl{\cdot}y) = (x = y)$
⟨*proof*⟩

lemma *sinr-eq* [*simp*]: $(sinr{\cdot}x = sinr{\cdot}y) = (x = y)$
⟨*proof*⟩

lemma *sinl-eq-sinr* [*simp*]: $(sinl{\cdot}x = sinr{\cdot}y) = (x = \bot \wedge y = \bot)$
⟨*proof*⟩

lemma *sinr-eq-sinl* [*simp*]: $(sinr{\cdot}x = sinl{\cdot}y) = (x = \bot \wedge y = \bot)$
⟨*proof*⟩

lemma *sinl-inject*: $sinl{\cdot}x = sinl{\cdot}y \Longrightarrow x = y$
⟨*proof*⟩

lemma *sinr-inject*: $sinr{\cdot}x = sinr{\cdot}y \Longrightarrow x = y$
⟨*proof*⟩

Strictness

lemma *sinl-strict* [*simp*]: $sinl{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *sinr-strict* [*simp*]: $sinr{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *sinl-defined-iff* [*simp*]: $(sinl{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

lemma *sinr-defined-iff* [*simp*]: $(sinr{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

lemma *sinl-defined* [*intro*!]: $x \neq \bot \Longrightarrow sinl{\cdot}x \neq \bot$
⟨*proof*⟩

lemma *sinr-defined* [*intro*!]: $x \neq \bot \Longrightarrow sinr{\cdot}x \neq \bot$
⟨*proof*⟩

Compactness

lemma *compact-sinl*: $compact\ x \Longrightarrow compact\ (sinl{\cdot}x)$
⟨*proof*⟩

lemma *compact-sinr*: $compact\ x \Longrightarrow compact\ (sinr{\cdot}x)$
⟨*proof*⟩

lemma *compact-sinlD*: $compact\ (sinl{\cdot}x) \Longrightarrow compact\ x$
⟨*proof*⟩

lemma *compact-sinrD*: $compact\ (sinr{\cdot}x) \Longrightarrow compact\ x$
⟨*proof*⟩

lemma *compact-sinl-iff* [*simp*]: $compact\ (sinl{\cdot}x) = compact\ x$
⟨*proof*⟩

lemma *compact-sinr-iff* [*simp*]: $compact\ (sinr{\cdot}x) = compact\ x$
⟨*proof*⟩

17.4 Case analysis

lemma *Exh-Ssum*:
$z = \bot \vee (\exists\, a.\ z = sinl{\cdot}a \wedge a \neq \bot) \vee (\exists\, b.\ z = sinr{\cdot}b \wedge b \neq \bot)$
⟨*proof*⟩

lemma *ssumE* [*cases type*: ++]:
$[\![p = \bot \Longrightarrow Q;$
$\bigwedge x.\ [\![p = sinl{\cdot}x;\ x \neq \bot]\!] \Longrightarrow Q;$
$\bigwedge y.\ [\![p = sinr{\cdot}y;\ y \neq \bot]\!] \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *ssum-induct* [*induct type*: ++]:
$[\![P\ \bot;$
$\bigwedge x.\ x \neq \bot \Longrightarrow P\ (sinl{\cdot}x);$
$\bigwedge y.\ y \neq \bot \Longrightarrow P\ (sinr{\cdot}y)]\!] \Longrightarrow P\ x$
⟨*proof*⟩

lemma *ssumE2*:
$[\![\bigwedge x.\ p = sinl{\cdot}x \Longrightarrow Q;\ \bigwedge y.\ p = sinr{\cdot}y \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *less-sinlD*: $p \sqsubseteq sinl{\cdot}x \Longrightarrow \exists\, y.\ p = sinl{\cdot}y \wedge y \sqsubseteq x$
⟨*proof*⟩

lemma *less-sinrD*: $p \sqsubseteq sinr{\cdot}x \Longrightarrow \exists\, y.\ p = sinr{\cdot}y \wedge y \sqsubseteq x$

⟨*proof*⟩

17.5 Case analysis combinator

definition
sscase :: ('*a* → '*c*) → ('*b* → '*c*) → ('*a* ++ '*b*) → '*c* **where**
sscase = (Λ *f g s*. (Λ<*t*, *x*, *y*>. *If t then f*·*x else g*·*y fi*)·(*Rep-Ssum s*))

translations
case s of XCONST sinl·*x* ⇒ *t1* | *XCONST sinr*·*y* ⇒ *t2* == *CONST sscase*·(Λ *x*. *t1*)·(Λ *y*. *t2*)·*s*

translations
Λ(*XCONST sinl*·*x*). *t* == *CONST sscase*·(Λ *x*. *t*)·⊥
Λ(*XCONST sinr*·*y*). *t* == *CONST sscase*·⊥·(Λ *y*. *t*)

lemma *beta-sscase*:
sscase·*f*·*g*·*s* = (Λ<*t*, *x*, *y*>. *If t then f*·*x else g*·*y fi*)·(*Rep-Ssum s*)
⟨*proof*⟩

lemma *sscase1* [*simp*]: *sscase*·*f*·*g*·⊥ = ⊥
⟨*proof*⟩

lemma *sscase2* [*simp*]: *x* ≠ ⊥ ⟹ *sscase*·*f*·*g*·(*sinl*·*x*) = *f*·*x*
⟨*proof*⟩

lemma *sscase3* [*simp*]: *y* ≠ ⊥ ⟹ *sscase*·*f*·*g*·(*sinr*·*y*) = *g*·*y*
⟨*proof*⟩

lemma *sscase4* [*simp*]: *sscase*·*sinl*·*sinr*·*z* = *z*
⟨*proof*⟩

17.6 Strict sum preserves flatness

instance ++ :: (*flat*, *flat*) *flat*
⟨*proof*⟩

17.7 Strict sum is a bifinite domain

instantiation ++ :: (*bifinite*, *bifinite*) *bifinite*
begin

definition
approx-ssum-def:
approx = (λ*n*. *sscase*·(Λ *x*. *sinl*·(*approx n*·*x*))·(Λ *y*. *sinr*·(*approx n*·*y*)))

lemma *approx-sinl* [*simp*]: *approx i*·(*sinl*·*x*) = *sinl*·(*approx i*·*x*)
⟨*proof*⟩

lemma *approx-sinr* [*simp*]: *approx i*·(*sinr*·*x*) = *sinr*·(*approx i*·*x*)

⟨*proof*⟩

instance ⟨*proof*⟩

end

end

18 Sprod: The type of strict products

theory *Sprod*
imports *Cprod*
begin

defaultsort *pcpo*

18.1 Definition of strict product type

pcpodef (*Sprod*) (*'a*, *'b*) ** (**infixr** ** *20*) =
{*p*::*'a* × *'b*. *p* = ⊥ ∨ (*cfst*·*p* ≠ ⊥ ∧ *csnd*·*p* ≠ ⊥)}
⟨*proof*⟩

instance ** :: ({*finite-po*,*pcpo*}, {*finite-po*,*pcpo*}) *finite-po*
⟨*proof*⟩

instance ** :: ({*chfin*,*pcpo*}, {*chfin*,*pcpo*}) *chfin*
⟨*proof*⟩

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

lemma *spair-lemma*:
<*strictify*·(Λ *b*. *a*)·*b*, *strictify*·(Λ *a*. *b*)·*a*> ∈ *Sprod*
⟨*proof*⟩

18.2 Definitions of constants

definition
sfst :: (*'a* ** *'b*) → *'a* **where**
sfst = (Λ *p*. *cfst*·(*Rep-Sprod p*))

definition
ssnd :: (*'a* ** *'b*) → *'b* **where**
ssnd = (Λ *p*. *csnd*·(*Rep-Sprod p*))

definition

spair :: $'a \to 'b \to ('a \mathbin{**} 'b)$ **where**
spair = $(\Lambda\ a\ b.\ \mathit{Abs\text{-}Sprod}$
$<\mathit{strictify}\cdot(\Lambda\ b.\ a)\cdot b,\ \mathit{strictify}\cdot(\Lambda\ a.\ b)\cdot a>)$

definition
ssplit :: $('a \to 'b \to 'c) \to ('a \mathbin{**} 'b) \to 'c$ **where**
ssplit = $(\Lambda\ f.\ \mathit{strictify}\cdot(\Lambda\ p.\ f\cdot(\mathit{sfst}\cdot p)\cdot(\mathit{ssnd}\cdot p)))$

syntax
@*stuple* :: $['a,\ \mathit{args}] => 'a \mathbin{**} 'b$ $((1'(:\text{-},/\ \text{-}:')))$
translations
$(:x,\ y,\ z:) == (:x,\ (:y,\ z:):)$
$(:x,\ y:) == \mathit{CONST\ spair}\cdot x\cdot y$

translations
$\Lambda(\mathit{CONST\ spair}\cdot x\cdot y).\ t == \mathit{CONST\ ssplit}\cdot(\Lambda\ x\ y.\ t)$

18.3 Case analysis

lemma *Rep-Sprod-spair*:
Rep-Sprod $(:a,\ b:) = <\mathit{strictify}\cdot(\Lambda\ b.\ a)\cdot b,\ \mathit{strictify}\cdot(\Lambda\ a.\ b)\cdot a>$
⟨*proof*⟩

lemmas *Rep-Sprod-simps* =
Rep-Sprod-inject [*symmetric*] *less-Sprod-def*
Rep-Sprod-strict Rep-Sprod-spair

lemma *Exh-Sprod*:
$z = \bot \vee (\exists\ a\ b.\ z = (:a,\ b:) \wedge a \neq \bot \wedge b \neq \bot)$
⟨*proof*⟩

lemma *sprodE* [*cases type*: $**$]:
$[\![p = \bot \Longrightarrow Q;\ \bigwedge x\ y.\ [\![p = (:x,\ y:);\ x \neq \bot;\ y \neq \bot]\!] \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *sprod-induct* [*induct type*: $**$]:
$[\![P\ \bot;\ \bigwedge x\ y.\ [\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow P\ (:x,\ y:)]\!] \Longrightarrow P\ x$
⟨*proof*⟩

18.4 Properties of *spair*

lemma *spair-strict1* [*simp*]: $(:\bot,\ y:) = \bot$
⟨*proof*⟩

lemma *spair-strict2* [*simp*]: $(:x,\ \bot:) = \bot$
⟨*proof*⟩

lemma *spair-strict-iff* [*simp*]: $((:x,\ y:) = \bot) = (x = \bot \vee y = \bot)$
⟨*proof*⟩

lemma *spair-less-iff*:
$((:a,\ b:) \sqsubseteq (:c,\ d:)) = (a = \bot \vee b = \bot \vee (a \sqsubseteq c \wedge b \sqsubseteq d))$
⟨*proof*⟩

lemma *spair-eq-iff*:
$((:a,\ b:) = (:c,\ d:)) =$
$(a = c \wedge b = d \vee (a = \bot \vee b = \bot) \wedge (c = \bot \vee d = \bot))$
⟨*proof*⟩

lemma *spair-strict*: $x = \bot \vee y = \bot \Longrightarrow (:x,\ y:) = \bot$
⟨*proof*⟩

lemma *spair-strict-rev*: $(:x,\ y:) \neq \bot \Longrightarrow x \neq \bot \wedge y \neq \bot$
⟨*proof*⟩

lemma *spair-defined*: $[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \neq \bot$
⟨*proof*⟩

lemma *spair-defined-rev*: $(:x,\ y:) = \bot \Longrightarrow x = \bot \vee y = \bot$
⟨*proof*⟩

lemma *spair-eq*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow ((:x,\ y:) = (:a,\ b:)) = (x = a \wedge y = b)$
⟨*proof*⟩

lemma *spair-inject*:
$[\![x \neq \bot;\ y \neq \bot;\ (:x,\ y:) = (:a,\ b:)]\!] \Longrightarrow x = a \wedge y = b$
⟨*proof*⟩

lemma *inst-sprod-pcpo2*: $UU = (:UU, UU:)$
⟨*proof*⟩

18.5 Properties of *sfst* and *ssnd*

lemma *sfst-strict* [*simp*]: $sfst{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *ssnd-strict* [*simp*]: $ssnd{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *sfst-spair* [*simp*]: $y \neq \bot \Longrightarrow sfst{\cdot}(:x,\ y:) = x$
⟨*proof*⟩

lemma *ssnd-spair* [*simp*]: $x \neq \bot \Longrightarrow ssnd{\cdot}(:x,\ y:) = y$
⟨*proof*⟩

lemma *sfst-defined-iff* [*simp*]: $(sfst{\cdot}p = \bot) = (p = \bot)$
⟨*proof*⟩

lemma *ssnd-defined-iff* [*simp*]: $(ssnd{\cdot}p = \bot) = (p = \bot)$
⟨*proof*⟩

lemma *sfst-defined*: $p \neq \bot \Longrightarrow sfst{\cdot}p \neq \bot$
⟨*proof*⟩

lemma *ssnd-defined*: $p \neq \bot \Longrightarrow ssnd{\cdot}p \neq \bot$
⟨*proof*⟩

lemma *surjective-pairing-Sprod2*: $(:sfst{\cdot}p,\ ssnd{\cdot}p:) = p$
⟨*proof*⟩

lemma *less-sprod*: $x \sqsubseteq y = (sfst{\cdot}x \sqsubseteq sfst{\cdot}y \wedge ssnd{\cdot}x \sqsubseteq ssnd{\cdot}y)$
⟨*proof*⟩

lemma *eq-sprod*: $(x = y) = (sfst{\cdot}x = sfst{\cdot}y \wedge ssnd{\cdot}x = ssnd{\cdot}y)$
⟨*proof*⟩

lemma *spair-less*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) = (x \sqsubseteq a \wedge y \sqsubseteq b)$
⟨*proof*⟩

lemma *sfst-less-iff*: $sfst{\cdot}x \sqsubseteq y = x \sqsubseteq (:y,\ ssnd{\cdot}x:)$
⟨*proof*⟩

lemma *ssnd-less-iff*: $ssnd{\cdot}x \sqsubseteq y = x \sqsubseteq (:sfst{\cdot}x,\ y:)$
⟨*proof*⟩

18.6 Compactness

lemma *compact-sfst*: $compact\ x \Longrightarrow compact\ (sfst{\cdot}x)$
⟨*proof*⟩

lemma *compact-ssnd*: $compact\ x \Longrightarrow compact\ (ssnd{\cdot}x)$
⟨*proof*⟩

lemma *compact-spair*: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact\ (:x,\ y:)$
⟨*proof*⟩

lemma *compact-spair-iff*:
$compact\ (:x,\ y:) = (x = \bot \vee y = \bot \vee (compact\ x \wedge compact\ y))$
⟨*proof*⟩

18.7 Properties of *ssplit*

lemma *ssplit1* [*simp*]: $ssplit{\cdot}f{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *ssplit2* [*simp*]: $[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow ssplit{\cdot}f{\cdot}(:x,\ y:) = f{\cdot}x{\cdot}y$
⟨*proof*⟩

lemma *ssplit3* [*simp*]: *ssplit·spair·z* = *z*
⟨*proof*⟩

18.8 Strict product preserves flatness

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

18.9 Strict product is a bifinite domain

instantiation ∗∗ :: (*bifinite*, *bifinite*) *bifinite*
begin

definition
approx-sprod-def:
approx = (λ*n*. Λ(:*x*, *y*:). (:*approx n·x*, *approx n·y*:))

instance ⟨*proof*⟩

end

lemma *approx-spair* [*simp*]:
approx i·(:*x*, *y*:) = (:*approx i·x*, *approx i·y*:)
⟨*proof*⟩

end

19 One: The unit domain

theory *One*
imports *Lift*
begin

types *one* = *unit lift*
translations
one <= (*type*) *unit lift*

definition
ONE :: *one*
where
ONE == *Def* ()

Exhaustion and Elimination for type *one*

lemma *Exh-one*: *t* = ⊥ ∨ *t* = *ONE*
⟨*proof*⟩

lemma *oneE*: ⟦*p* = ⊥ ⟹ *Q*; *p* = *ONE* ⟹ *Q*⟧ ⟹ *Q*

⟨*proof*⟩

lemma *one-induct*: ⟦*P* ⊥; *P ONE*⟧ ⟹ *P x*
⟨*proof*⟩

lemma *dist-less-one* [*simp*]: ¬ *ONE* ⊑ ⊥
⟨*proof*⟩

lemma *less-ONE* [*simp*]: *x* ⊑ *ONE*
⟨*proof*⟩

lemma *ONE-less-iff* [*simp*]: *ONE* ⊑ *x* ⟷ *x* = *ONE*
⟨*proof*⟩

lemma *dist-eq-one* [*simp*]: *ONE* ≠ ⊥ ⊥ ≠ *ONE*
⟨*proof*⟩

lemma *one-neq-iffs* [*simp*]:
x ≠ *ONE* ⟷ *x* = ⊥
ONE ≠ *x* ⟷ *x* = ⊥
x ≠ ⊥ ⟷ *x* = *ONE*
⊥ ≠ *x* ⟷ *x* = *ONE*
⟨*proof*⟩

lemma *compact-ONE*: *compact ONE*
⟨*proof*⟩

Case analysis function for type *one*

definition
one-when :: *'a*::*pcpo* → *one* → *'a* **where**
one-when = (Λ *a*. *strictify*·(Λ -. *a*))

translations
case x of XCONST ONE ⇒ *t* == *CONST one-when*·*t*·*x*
Λ (*XCONST ONE*). *t* == *CONST one-when*·*t*

lemma *one-when1* [*simp*]: (*case* ⊥ *of ONE* ⇒ *t*) = ⊥
⟨*proof*⟩

lemma *one-when2* [*simp*]: (*case ONE of ONE* ⇒ *t*) = *t*
⟨*proof*⟩

lemma *one-when3* [*simp*]: (*case x of ONE* ⇒ *ONE*) = *x*
⟨*proof*⟩

end

20 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun Cprod*
begin

defaultsort *pcpo*

20.1 Iteration

consts
iterate :: *nat* ⇒ ('*a*::*cpo* → '*a*) → ('*a* → '*a*)

primrec
iterate 0 = (Λ *F x*. *x*)
iterate (*Suc n*) = (Λ *F x*. *F*·(*iterate n*·*F*·*x*))

Derive inductive properties of iterate from primitive recursion

lemma *iterate-0* [*simp*]: *iterate 0*·*F*·*x* = *x*
⟨*proof*⟩

lemma *iterate-Suc* [*simp*]: *iterate* (*Suc n*)·*F*·*x* = *F*·(*iterate n*·*F*·*x*)
⟨*proof*⟩

declare *iterate.simps* [*simp del*]

lemma *iterate-Suc2*: *iterate* (*Suc n*)·*F*·*x* = *iterate n*·*F*·(*F*·*x*)
⟨*proof*⟩

lemma *iterate-iterate*:
iterate m·*F*·(*iterate n*·*F*·*x*) = *iterate* (*m* + *n*)·*F*·*x*
⟨*proof*⟩

The sequence of function iterations is a chain. This property is essential since monotonicity of iterate makes no sense.

lemma *chain-iterate2*: *x* ⊑ *F*·*x* ⟹ *chain* (λ*i*. *iterate i*·*F*·*x*)
⟨*proof*⟩

lemma *chain-iterate* [*simp*]: *chain* (λ*i*. *iterate i*·*F*·⊥)
⟨*proof*⟩

20.2 Least fixed point operator

definition
fix :: ('*a* → '*a*) → '*a* **where**
fix = (Λ *F*. ⨆ *i*. *iterate i*·*F*·⊥)

Binder syntax for *fix*

abbreviation

fix-syn :: $('a \Rightarrow 'a) \Rightarrow 'a$ (**binder** *FIX* *10*) **where**
fix-syn $(\lambda x.\ f\ x) \equiv fix{\cdot}(\Lambda\ x.\ f\ x)$

notation (*xsymbols*)
fix-syn (**binder** μ *10*)

Properties of *fix*

direct connection between *fix* and iteration

lemma *fix-def2*: $fix{\cdot}F = (\bigsqcup i.\ iterate\ i{\cdot}F{\cdot}\bot)$
⟨*proof*⟩

Kleene's fixed point theorems for continuous functions in pointed omega cpo's

lemma *fix-eq*: $fix{\cdot}F = F{\cdot}(fix{\cdot}F)$
⟨*proof*⟩

lemma *fix-least-less*: $F{\cdot}x \sqsubseteq x \Longrightarrow fix{\cdot}F \sqsubseteq x$
⟨*proof*⟩

lemma *fix-least*: $F{\cdot}x = x \Longrightarrow fix{\cdot}F \sqsubseteq x$
⟨*proof*⟩

lemma *fix-eqI*:
assumes *fixed*: $F{\cdot}x = x$ **and** *least*: $\bigwedge z.\ F{\cdot}z = z \Longrightarrow x \sqsubseteq z$
shows $fix{\cdot}F = x$
⟨*proof*⟩

lemma *fix-eq2*: $f \equiv fix{\cdot}F \Longrightarrow f = F{\cdot}f$
⟨*proof*⟩

lemma *fix-eq3*: $f \equiv fix{\cdot}F \Longrightarrow f{\cdot}x = F{\cdot}f{\cdot}x$
⟨*proof*⟩

lemma *fix-eq4*: $f = fix{\cdot}F \Longrightarrow f = F{\cdot}f$
⟨*proof*⟩

lemma *fix-eq5*: $f = fix{\cdot}F \Longrightarrow f{\cdot}x = F{\cdot}f{\cdot}x$
⟨*proof*⟩

strictness of *fix*

lemma *fix-defined-iff*: $(fix{\cdot}F = \bot) = (F{\cdot}\bot = \bot)$
⟨*proof*⟩

lemma *fix-strict*: $F{\cdot}\bot = \bot \Longrightarrow fix{\cdot}F = \bot$
⟨*proof*⟩

lemma *fix-defined*: $F{\cdot}\bot \neq \bot \Longrightarrow fix{\cdot}F \neq \bot$
⟨*proof*⟩

fix applied to identity and constant functions

lemma *fix-id*: $(\mu\ x.\ x) = \bot$
⟨*proof*⟩

lemma *fix-const*: $(\mu\ x.\ c) = c$
⟨*proof*⟩

20.3 Fixed point induction

lemma *fix-ind*: $[\![adm\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F{\cdot}x)]\!] \Longrightarrow P\ (fix{\cdot}F)$
⟨*proof*⟩

lemma *def-fix-ind*:
$[\![f \equiv fix{\cdot}F;\ adm\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F{\cdot}x)]\!] \Longrightarrow P\ f$
⟨*proof*⟩

lemma *fix-ind2*:
assumes *adm*: *adm P*
assumes *0*: $P\ \bot$ **and** *1*: $P\ (F{\cdot}\bot)$
assumes *step*: $\bigwedge x.\ [\![P\ x;\ P\ (F{\cdot}x)]\!] \Longrightarrow P\ (F{\cdot}(F{\cdot}x))$
shows $P\ (fix{\cdot}F)$
⟨*proof*⟩

20.4 Recursive let bindings

definition
CLetrec :: $('a \to 'a \times 'b) \to 'b$ **where**
$CLetrec = (\Lambda\ F.\ csnd{\cdot}(F{\cdot}(\mu\ x.\ cfst{\cdot}(F{\cdot}x))))$

nonterminals
recbinds recbindt recbind

syntax
-recbind :: $['a, 'a] \Rightarrow recbind$ ((2- =/ -) 10)
:: $recbind \Rightarrow recbindt$ (-)
-recbindt :: $[recbind, recbindt] \Rightarrow recbindt$ (-,/ -)
:: $recbindt \Rightarrow recbinds$ (-)
-recbinds :: $[recbindt, recbinds] \Rightarrow recbinds$ (-;/ -)
-Letrec :: $[recbinds, 'a] \Rightarrow 'a$ ((*Letrec* (-)/ *in* (-)) 10)

translations
(*recbindt*) $x = a, \langle y,ys\rangle = \langle b,bs\rangle$ == (*recbindt*) $\langle x,y,ys\rangle = \langle a,b,bs\rangle$
(*recbindt*) $x = a,\ y = b$ == (*recbindt*) $\langle x,y\rangle = \langle a,b\rangle$

translations
-Letrec (*-recbinds b bs*) *e* == *-Letrec b* (*-Letrec bs e*)
Letrec xs = a in $\langle e,es\rangle$ == *CONST CLetrec*·(Λ *xs*. $\langle a,e,es\rangle$)
Letrec xs = a in e == *CONST CLetrec*·(Λ *xs*. $\langle a,e\rangle$)

Bekic's Theorem: Simultaneous fixed points over pairs can be written in terms of separate fixed points.

lemma *fix-cprod*:
fix·(*F*::'*a* × '*b* → '*a* × '*b*) =
⟨*μ x*. *cfst*·(*F*·⟨*x*, *μ y*. *csnd*·(*F*·⟨*x*, *y*⟩)⟩),
μ y. *csnd*·(*F*·⟨*μ x*. *cfst*·(*F*·⟨*x*, *μ y*. *csnd*·(*F*·⟨*x*, *y*⟩)⟩), *y*⟩)⟩
(**is** *fix*·*F* = ⟨*?x*, *?y*⟩)
⟨*proof*⟩

20.5 Weak admissibility

definition
admw :: ('*a* ⇒ *bool*) ⇒ *bool* **where**
admw P = (∀ *F*. (∀ *n*. *P* (*iterate n*·*F*·⊥)) ⟶ *P* (⨆ *i*. *iterate i*·*F*·⊥))

an admissible formula is also weak admissible

lemma *adm-impl-admw*: *adm P* ⟹ *admw P*
⟨*proof*⟩

computational induction for weak admissible formulae

lemma *wfix-ind*: ⟦*admw P*; ∀ *n*. *P* (*iterate n*·*F*·⊥)⟧ ⟹ *P* (*fix*·*F*)
⟨*proof*⟩

lemma *def-wfix-ind*:
⟦*f* ≡ *fix*·*F*; *admw P*; ∀ *n*. *P* (*iterate n*·*F*·⊥)⟧ ⟹ *P f*
⟨*proof*⟩

end

21 Fixrec: Package for defining recursive functions in HOLCF

theory *Fixrec*
imports *Sprod Ssum Up One Tr Fix*
uses (*Tools/fixrec-package.ML*)
begin

21.1 Maybe monad type

defaultsort *cpo*

pcpodef (**open**) '*a maybe* = *UNIV*::(*one* ++ '*a u*) *set*
⟨*proof*⟩

definition
fail :: '*a maybe* **where**

fail = *Abs-maybe* (*sinl*·*ONE*)

definition
return :: $'a \to 'a$ *maybe* **where**
return = (Λ *x*. *Abs-maybe* (*sinr*·(*up*·*x*)))

definition
maybe-when :: $'b \to ('a \to 'b) \to 'a$ *maybe* $\to 'b$::*pcpo* **where**
maybe-when = (Λ *f r m*. *sscase*·(Λ *x*. *f*)·(*fup*·*r*)·(*Rep-maybe m*))

lemma *maybeE*:
$[\![p = \bot \Longrightarrow Q; p = fail \Longrightarrow Q; \bigwedge x.\ p = return{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *return-defined* [*simp*]: *return*·*x* ≠ ⊥
⟨*proof*⟩

lemma *fail-defined* [*simp*]: *fail* ≠ ⊥
⟨*proof*⟩

lemma *return-eq* [*simp*]: (*return*·*x* = *return*·*y*) = (*x* = *y*)
⟨*proof*⟩

lemma *return-neq-fail* [*simp*]:
return·*x* ≠ *fail* *fail* ≠ *return*·*x*
⟨*proof*⟩

lemma *maybe-when-rews* [*simp*]:
maybe-when·*f*·*r*·⊥ = ⊥
maybe-when·*f*·*r*·*fail* = *f*
maybe-when·*f*·*r*·(*return*·*x*) = *r*·*x*
⟨*proof*⟩

translations
case m of XCONST fail ⇒ *t1* | *XCONST return*·*x* ⇒ *t2*
== *CONST maybe-when*·*t1*·(Λ *x*. *t2*)·*m*

21.1.1 Monadic bind operator

definition
bind :: *'a maybe* → (*'a* → *'b maybe*) → *'b maybe* **where**
bind = (Λ *m f*. *case m of fail* ⇒ *fail* | *return*·*x* ⇒ *f*·*x*)

monad laws

lemma *bind-strict* [*simp*]: *bind*·⊥·*f* = ⊥
⟨*proof*⟩

lemma *bind-fail* [*simp*]: *bind*·*fail*·*f* = *fail*
⟨*proof*⟩

lemma *left-unit* [*simp*]: *bind*·(*return*·*a*)·*k* = *k*·*a*
⟨*proof*⟩

lemma *right-unit* [*simp*]: *bind*·*m*·*return* = *m*
⟨*proof*⟩

lemma *bind-assoc*:
bind·(*bind*·*m*·*k*)·*h* = *bind*·*m*·(Λ *a*. *bind*·(*k*·*a*)·*h*)
⟨*proof*⟩

21.1.2 Run operator

definition
run :: *'a maybe* → *'a*::*pcpo* **where**
run = *maybe-when*·⊥·*ID*

rewrite rules for run

lemma *run-strict* [*simp*]: *run*·⊥ = ⊥
⟨*proof*⟩

lemma *run-fail* [*simp*]: *run*·*fail* = ⊥
⟨*proof*⟩

lemma *run-return* [*simp*]: *run*·(*return*·*x*) = *x*
⟨*proof*⟩

21.1.3 Monad plus operator

definition
mplus :: *'a maybe* → *'a maybe* → *'a maybe* **where**
mplus = (Λ *m1 m2*. *case m1 of fail* ⇒ *m2* | *return*·*x* ⇒ *m1*)

abbreviation
mplus-syn :: [*'a maybe*, *'a maybe*] ⇒ *'a maybe* (**infixr** +++ *65*) **where**
m1 +++ *m2* == *mplus*·*m1*·*m2*

rewrite rules for mplus

lemma *mplus-strict* [*simp*]: ⊥ +++ *m* = ⊥
⟨*proof*⟩

lemma *mplus-fail* [*simp*]: *fail* +++ *m* = *m*
⟨*proof*⟩

lemma *mplus-return* [*simp*]: *return*·*x* +++ *m* = *return*·*x*
⟨*proof*⟩

lemma *mplus-fail2* [*simp*]: *m* +++ *fail* = *m*
⟨*proof*⟩

lemma *mplus-assoc*: $(x +\!\!+\!\!+ y) +\!\!+\!\!+ z = x +\!\!+\!\!+ (y +\!\!+\!\!+ z)$
⟨*proof*⟩

21.1.4 Fatbar combinator

definition
$fatbar :: ('a \to 'b\ maybe) \to ('a \to 'b\ maybe) \to ('a \to 'b\ maybe)$ **where**
$fatbar = (\Lambda\ a\ b\ x.\ a{\cdot}x +\!\!+\!\!+ b{\cdot}x)$

abbreviation
$fatbar\text{-}syn :: ['a \to 'b\ maybe,\ 'a \to 'b\ maybe] \Rightarrow 'a \to 'b\ maybe$ (**infixr** $\parallel$ *60*)
where
$m1 \parallel m2 == fatbar{\cdot}m1{\cdot}m2$

lemma *fatbar1*: $m{\cdot}x = \bot \Longrightarrow (m \parallel ms){\cdot}x = \bot$
⟨*proof*⟩

lemma *fatbar2*: $m{\cdot}x = fail \Longrightarrow (m \parallel ms){\cdot}x = ms{\cdot}x$
⟨*proof*⟩

lemma *fatbar3*: $m{\cdot}x = return{\cdot}y \Longrightarrow (m \parallel ms){\cdot}x = return{\cdot}y$
⟨*proof*⟩

lemmas *fatbar-simps = fatbar1 fatbar2 fatbar3*

lemma *run-fatbar1*: $m{\cdot}x = \bot \Longrightarrow run{\cdot}((m \parallel ms){\cdot}x) = \bot$
⟨*proof*⟩

lemma *run-fatbar2*: $m{\cdot}x = fail \Longrightarrow run{\cdot}((m \parallel ms){\cdot}x) = run{\cdot}(ms{\cdot}x)$
⟨*proof*⟩

lemma *run-fatbar3*: $m{\cdot}x = return{\cdot}y \Longrightarrow run{\cdot}((m \parallel ms){\cdot}x) = y$
⟨*proof*⟩

lemmas *run-fatbar-simps* [*simp*] = *run-fatbar1 run-fatbar2 run-fatbar3*

21.2 Case branch combinator

definition
$branch :: ('a \to 'b\ maybe) \Rightarrow ('b \to 'c) \to ('a \to 'c\ maybe)$ **where**
$branch\ p \equiv \Lambda\ r\ x.\ bind{\cdot}(p{\cdot}x){\cdot}(\Lambda\ y.\ return{\cdot}(r{\cdot}y))$

lemma *branch-rews*:
$p{\cdot}x = \bot \Longrightarrow branch\ p{\cdot}r{\cdot}x = \bot$
$p{\cdot}x = fail \Longrightarrow branch\ p{\cdot}r{\cdot}x = fail$
$p{\cdot}x = return{\cdot}y \Longrightarrow branch\ p{\cdot}r{\cdot}x = return{\cdot}(r{\cdot}y)$
⟨*proof*⟩

lemma *branch-return* [*simp*]: $branch\ return{\cdot}r{\cdot}x = return{\cdot}(r{\cdot}x)$

⟨*proof*⟩

21.2.1 Cases operator

definition
cases :: *'a maybe* → *'a*::*pcpo* **where**
cases = *maybe-when*·⊥·*ID*

rewrite rules for cases

lemma *cases-strict* [*simp*]: *cases*·⊥ = ⊥
⟨*proof*⟩

lemma *cases-fail* [*simp*]: *cases*·*fail* = ⊥
⟨*proof*⟩

lemma *cases-return* [*simp*]: *cases*·(*return*·*x*) = *x*
⟨*proof*⟩

21.3 Case syntax

nonterminals
Case-syn *Cases-syn*

syntax
-Case-syntax:: [*'a*, *Cases-syn*] => *'b* (("*Case* - *of*/ -) 10)
-Case1 :: [*'a*, *'b*] => *Case-syn* (("2- =>/ -) 10)
:: *Case-syn* => *Cases-syn* (-)
-Case2 :: [*Case-syn*, *Cases-syn*] => *Cases-syn* (-/ | -)

syntax (*xsymbols*)
-Case1 :: [*'a*, *'b*] => *Case-syn* (("2- ⇒/ -) 10)

translations
-Case-syntax x ms == *CONST Fixrec.cases*·(*ms*·*x*)
-Case2 m ms == *m* ‖ *ms*

Parsing Case expressions

syntax
-pat :: *'a*
-variable :: *'a*
-noargs :: *'a*

translations
-Case1 p r => *CONST branch* (*-pat p*)·(*-variable p r*)
-variable (*-args x y*) *r* => *CONST csplit*·(*-variable x* (*-variable y r*))
-variable -noargs r => *CONST unit-when*·*r*

⟨*ML*⟩

Printing Case expressions

syntax
-match :: *'a*

⟨*ML*⟩

translations
x <= *-match Fixrec.return* (*-variable x*)

21.4 Pattern combinators for data constructors

types (*'a*, *'b*) *pat* = *'a* → *'b maybe*

definition
cpair-pat :: (*'a*, *'c*) *pat* ⇒ (*'b*, *'d*) *pat* ⇒ (*'a* × *'b*, *'c* × *'d*) *pat* **where**
cpair-pat p1 p2 = (Λ⟨*x*, *y*⟩.
bind·(*p1*·*x*)·(Λ *a*. *bind*·(*p2*·*y*)·(Λ *b*. *return*·⟨*a*, *b*⟩)))

definition
spair-pat ::
(*'a*, *'c*) *pat* ⇒ (*'b*, *'d*) *pat* ⇒ (*'a*::*pcpo* ⊗ *'b*::*pcpo*, *'c* × *'d*) *pat* **where**
spair-pat p1 p2 = (Λ(:*x*, *y*:). *cpair-pat p1 p2*·⟨*x*, *y*⟩)

definition
sinl-pat :: (*'a*, *'c*) *pat* ⇒ (*'a*::*pcpo* ⊕ *'b*::*pcpo*, *'c*) *pat* **where**
sinl-pat p = *sscase*·*p*·(Λ *x*. *fail*)

definition
sinr-pat :: (*'b*, *'c*) *pat* ⇒ (*'a*::*pcpo* ⊕ *'b*::*pcpo*, *'c*) *pat* **where**
sinr-pat p = *sscase*·(Λ *x*. *fail*)·*p*

definition
up-pat :: (*'a*, *'b*) *pat* ⇒ (*'a u*, *'b*) *pat* **where**
up-pat p = *fup*·*p*

definition
TT-pat :: (*tr*, *unit*) *pat* **where**
TT-pat = (Λ *b*. *If b then return*·() *else fail fi*)

definition
FF-pat :: (*tr*, *unit*) *pat* **where**
FF-pat = (Λ *b*. *If b then fail else return*·() *fi*)

definition
ONE-pat :: (*one*, *unit*) *pat* **where**
ONE-pat = (Λ *ONE*. *return*·())

Parse translations (patterns)

translations
-pat (*XCONST cpair*·*x*·*y*) => *CONST cpair-pat* (*-pat x*) (*-pat y*)

-pat (XCONST spair·x·y) => CONST spair-pat (-pat x) (-pat y)
-pat (XCONST sinl·x) => CONST sinl-pat (-pat x)
-pat (XCONST sinr·x) => CONST sinr-pat (-pat x)
-pat (XCONST up·x) => CONST up-pat (-pat x)
-pat (XCONST TT) => CONST TT-pat
-pat (XCONST FF) => CONST FF-pat
-pat (XCONST ONE) => CONST ONE-pat

CONST version is also needed for constructors with special syntax

translations
-pat (CONST cpair·x·y) => CONST cpair-pat (-pat x) (-pat y)
-pat (CONST spair·x·y) => CONST spair-pat (-pat x) (-pat y)

Parse translations (variables)

translations
-variable (XCONST cpair·x·y) r => -variable (-args x y) r
-variable (XCONST spair·x·y) r => -variable (-args x y) r
-variable (XCONST sinl·x) r => -variable x r
-variable (XCONST sinr·x) r => -variable x r
-variable (XCONST up·x) r => -variable x r
-variable (XCONST TT) r => -variable -noargs r
-variable (XCONST FF) r => -variable -noargs r
-variable (XCONST ONE) r => -variable -noargs r

translations
-variable (CONST cpair·x·y) r => -variable (-args x y) r
-variable (CONST spair·x·y) r => -variable (-args x y) r

Print translations

translations
CONST cpair·(-match p1 v1)·(-match p2 v2)
<= -match (CONST cpair-pat p1 p2) (-args v1 v2)
CONST spair·(-match p1 v1)·(-match p2 v2)
<= -match (CONST spair-pat p1 p2) (-args v1 v2)
CONST sinl·(-match p1 v1) <= -match (CONST sinl-pat p1) v1
CONST sinr·(-match p1 v1) <= -match (CONST sinr-pat p1) v1
CONST up·(-match p1 v1) <= -match (CONST up-pat p1) v1
CONST TT <= -match (CONST TT-pat) -noargs
CONST FF <= -match (CONST FF-pat) -noargs
CONST ONE <= -match (CONST ONE-pat) -noargs

lemma *cpair-pat1*:
branch p·r·x = ⊥ ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = ⊥
⟨*proof*⟩

lemma *cpair-pat2*:
branch p·r·x = fail ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = fail
⟨*proof*⟩

lemma *cpair-pat3*:
branch p·r·x = return·s ⟹
branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = branch q·s·y
⟨*proof*⟩

lemmas *cpair-pat* [*simp*] =
cpair-pat1 cpair-pat2 cpair-pat3

lemma *spair-pat* [*simp*]:
branch (spair-pat p1 p2)·r·⊥ = ⊥
⟦x ≠ ⊥; y ≠ ⊥⟧
⟹ branch (spair-pat p1 p2)·r·(:x, y:) =
branch (cpair-pat p1 p2)·r·⟨x, y⟩
⟨*proof*⟩

lemma *sinl-pat* [*simp*]:
branch (sinl-pat p)·r·⊥ = ⊥
x ≠ ⊥ ⟹ branch (sinl-pat p)·r·(sinl·x) = branch p·r·x
y ≠ ⊥ ⟹ branch (sinl-pat p)·r·(sinr·y) = fail
⟨*proof*⟩

lemma *sinr-pat* [*simp*]:
branch (sinr-pat p)·r·⊥ = ⊥
x ≠ ⊥ ⟹ branch (sinr-pat p)·r·(sinl·x) = fail
y ≠ ⊥ ⟹ branch (sinr-pat p)·r·(sinr·y) = branch p·r·y
⟨*proof*⟩

lemma *up-pat* [*simp*]:
branch (up-pat p)·r·⊥ = ⊥
branch (up-pat p)·r·(up·x) = branch p·r·x
⟨*proof*⟩

lemma *TT-pat* [*simp*]:
branch TT-pat·(unit-when·r)·⊥ = ⊥
branch TT-pat·(unit-when·r)·TT = return·r
branch TT-pat·(unit-when·r)·FF = fail
⟨*proof*⟩

lemma *FF-pat* [*simp*]:
branch FF-pat·(unit-when·r)·⊥ = ⊥
branch FF-pat·(unit-when·r)·TT = fail
branch FF-pat·(unit-when·r)·FF = return·r
⟨*proof*⟩

lemma *ONE-pat* [*simp*]:
branch ONE-pat·(unit-when·r)·⊥ = ⊥
branch ONE-pat·(unit-when·r)·ONE = return·r
⟨*proof*⟩

21.5 Wildcards, as-patterns, and lazy patterns

syntax
-as-pat :: [*idt*, *'a*] ⇒ *'a* (**infixr** *as* *10*)
-lazy-pat :: *'a* ⇒ *'a* (∼ - [*1000*] *1000*)

definition
wild-pat :: *'a* → *unit maybe* **where**
wild-pat = (Λ *x*. *return*·())

definition
as-pat :: (*'a* → *'b maybe*) ⇒ *'a* → (*'a* × *'b*) *maybe* **where**
as-pat p = (Λ *x*. *bind*·(*p*·*x*)·(Λ *a*. *return*·⟨*x*, *a*⟩))

definition
lazy-pat :: (*'a* → *'b*::*pcpo maybe*) ⇒ (*'a* → *'b maybe*) **where**
lazy-pat p = (Λ *x*. *return*·(*cases*·(*p*·*x*)))

Parse translations (patterns)

translations
-pat - => *CONST wild-pat*
-pat (*-as-pat x y*) => *CONST as-pat* (*-pat y*)
-pat (*-lazy-pat x*) => *CONST lazy-pat* (*-pat x*)

Parse translations (variables)

translations
-variable - *r* => *-variable -noargs r*
-variable (*-as-pat x y*) *r* => *-variable* (*-args x y*) *r*
-variable (*-lazy-pat x*) *r* => *-variable x r*

Print translations

translations
- <= *-match* (*CONST wild-pat*) *-noargs*
-as-pat x (*-match p v*) <= *-match* (*CONST as-pat p*) (*-args* (*-variable x*) *v*)
-lazy-pat (*-match p v*) <= *-match* (*CONST lazy-pat p*) *v*

Lazy patterns in lambda abstractions

translations
-cabs (*-lazy-pat p*) *r* == *CONST Fixrec.cases oo* (*-Case1* (*-lazy-pat p*) *r*)

lemma *wild-pat* [*simp*]: *branch wild-pat*·(*unit-when*·*r*)·*x* = *return*·*r*
⟨*proof*⟩

lemma *as-pat* [*simp*]:
branch (*as-pat p*)·(*csplit*·*r*)·*x* = *branch p*·(*r*·*x*)·*x*
⟨*proof*⟩

lemma *lazy-pat* [*simp*]:
branch p·*r*·*x* = ⊥ ⟹ *branch* (*lazy-pat p*)·*r*·*x* = *return*·(*r*·⊥)

branch p·r·x = fail ⟹ branch (lazy-pat p)·r·x = return·(r·⊥)
branch p·r·x = return·s ⟹ branch (lazy-pat p)·r·x = return·s
⟨*proof*⟩

21.6 Match functions for built-in types

defaultsort *pcpo*

definition
match-UU :: *'a → unit maybe* **where**
match-UU = (Λ *x*. *fail*)

definition
match-cpair :: *'a*::*cpo* × *'b*::*cpo* → (*'a* × *'b*) *maybe* **where**
match-cpair = *csplit*·(Λ *x y*. *return*·<*x*,*y*>)

definition
match-spair :: *'a* ⊗ *'b* → (*'a* × *'b*) *maybe* **where**
match-spair = *ssplit*·(Λ *x y*. *return*·<*x*,*y*>)

definition
match-sinl :: *'a* ⊕ *'b* → *'a maybe* **where**
match-sinl = *sscase*·*return*·(Λ *y*. *fail*)

definition
match-sinr :: *'a* ⊕ *'b* → *'b maybe* **where**
match-sinr = *sscase*·(Λ *x*. *fail*)·*return*

definition
match-up :: *'a*::*cpo u* → *'a maybe* **where**
match-up = *fup*·*return*

definition
match-ONE :: *one* → *unit maybe* **where**
match-ONE = (Λ *ONE*. *return*·())

definition
match-TT :: *tr* → *unit maybe* **where**
match-TT = (Λ *b*. *If b then return*·() *else fail fi*)

definition
match-FF :: *tr* → *unit maybe* **where**
match-FF = (Λ *b*. *If b then fail else return*·() *fi*)

lemma *match-UU-simps* [*simp*]:
match-UU·*x* = *fail*
⟨*proof*⟩

lemma *match-cpair-simps* [*simp*]:

match-cpair·<*x*,*y*> = *return*·<*x*,*y*>
⟨*proof*⟩

lemma *match-spair-simps* [*simp*]:
⟦*x* ≠ ⊥; *y* ≠ ⊥⟧ ⟹ *match-spair*·(:*x*,*y*:) = *return*·<*x*,*y*>
match-spair·⊥ = ⊥
⟨*proof*⟩

lemma *match-sinl-simps* [*simp*]:
x ≠ ⊥ ⟹ *match-sinl*·(*sinl*·*x*) = *return*·*x*
x ≠ ⊥ ⟹ *match-sinl*·(*sinr*·*x*) = *fail*
match-sinl·⊥ = ⊥
⟨*proof*⟩

lemma *match-sinr-simps* [*simp*]:
x ≠ ⊥ ⟹ *match-sinr*·(*sinr*·*x*) = *return*·*x*
x ≠ ⊥ ⟹ *match-sinr*·(*sinl*·*x*) = *fail*
match-sinr·⊥ = ⊥
⟨*proof*⟩

lemma *match-up-simps* [*simp*]:
match-up·(*up*·*x*) = *return*·*x*
match-up·⊥ = ⊥
⟨*proof*⟩

lemma *match-ONE-simps* [*simp*]:
match-ONE·*ONE* = *return*·()
match-ONE·⊥ = ⊥
⟨*proof*⟩

lemma *match-TT-simps* [*simp*]:
match-TT·*TT* = *return*·()
match-TT·*FF* = *fail*
match-TT·⊥ = ⊥
⟨*proof*⟩

lemma *match-FF-simps* [*simp*]:
match-FF·*FF* = *return*·()
match-FF·*TT* = *fail*
match-FF·⊥ = ⊥
⟨*proof*⟩

21.7 Mutual recursion

The following rules are used to prove unfolding theorems from fixed-point definitions of mutually recursive functions.

lemma *cpair-equalI*: ⟦*x* ≡ *cfst*·*p*; *y* ≡ *csnd*·*p*⟧ ⟹ <*x*,*y*> ≡ *p*
⟨*proof*⟩

lemma *cpair-eqD1*: $<x,y> = <x',y'> \Longrightarrow x = x'$
⟨*proof*⟩

lemma *cpair-eqD2*: $<x,y> = <x',y'> \Longrightarrow y = y'$
⟨*proof*⟩

lemma for proving rewrite rules

lemma *ssubst-lhs*: $[\![t = s;\ P\ s = Q]\!] \Longrightarrow P\ t = Q$
⟨*proof*⟩

21.8 Initializing the fixrec package

⟨*ML*⟩

hide (**open**) *const return bind fail run cases*

end

22 Domain: Domain package

theory *Domain*
imports *Ssum Sprod Up One Tr Fixrec*
begin

defaultsort *pcpo*

22.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
  fixes *abs* :: $'a \to 'b$
  fixes *rep* :: $'b \to 'a$
  assumes *abs-iso* [*simp*]: $rep{\cdot}(abs{\cdot}x) = x$
  assumes *rep-iso* [*simp*]: $abs{\cdot}(rep{\cdot}y) = y$
begin

lemma *swap*: *iso rep abs*
  ⟨*proof*⟩

lemma *abs-less*: $(abs{\cdot}x \sqsubseteq abs{\cdot}y) = (x \sqsubseteq y)$
⟨*proof*⟩

lemma *rep-less*: $(rep{\cdot}x \sqsubseteq rep{\cdot}y) = (x \sqsubseteq y)$
  ⟨*proof*⟩

lemma *abs-eq*: $(abs{\cdot}x = abs{\cdot}y) = (x = y)$
  ⟨*proof*⟩

lemma *rep-eq*: $(rep{\cdot}x = rep{\cdot}y) = (x = y)$
⟨*proof*⟩

lemma *abs-strict*: $abs{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *rep-strict*: $rep{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *abs-defin'*: $abs{\cdot}x = \bot \Longrightarrow x = \bot$
⟨*proof*⟩

lemma *rep-defin'*: $rep{\cdot}z = \bot \Longrightarrow z = \bot$
⟨*proof*⟩

lemma *abs-defined*: $z \neq \bot \Longrightarrow abs{\cdot}z \neq \bot$
⟨*proof*⟩

lemma *rep-defined*: $z \neq \bot \Longrightarrow rep{\cdot}z \neq \bot$
⟨*proof*⟩

lemma *abs-defined-iff*: $(abs{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

lemma *rep-defined-iff*: $(rep{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

lemma (**in** *iso*) *compact-abs-rev*: $compact\ (abs{\cdot}x) \Longrightarrow compact\ x$
⟨*proof*⟩

lemma *compact-rep-rev*: $compact\ (rep{\cdot}x) \Longrightarrow compact\ x$
⟨*proof*⟩

lemma *compact-abs*: $compact\ x \Longrightarrow compact\ (abs{\cdot}x)$
⟨*proof*⟩

lemma *compact-rep*: $compact\ x \Longrightarrow compact\ (rep{\cdot}x)$
⟨*proof*⟩

lemma *iso-swap*: $(x = abs{\cdot}y) = (rep{\cdot}x = y)$
⟨*proof*⟩

end

22.2 Casedist

lemma *ex-one-defined-iff*:
$(\exists x.\ P\ x \wedge x \neq \bot) = P\ ONE$

⟨*proof*⟩

lemma *ex-up-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) = (\exists\, x.\ P\ (up{\cdot}x))$
⟨*proof*⟩

lemma *ex-sprod-defined-iff*:
$(\exists\, y.\ P\ y \wedge y \neq \bot) =$
$(\exists\, x\ y.\ (P\ (:x,\ y:) \wedge x \neq \bot) \wedge y \neq \bot)$
⟨*proof*⟩

lemma *ex-sprod-up-defined-iff*:
$(\exists\, y.\ P\ y \wedge y \neq \bot) =$
$(\exists\, x\ y.\ P\ (:up{\cdot}x,\ y:) \wedge y \neq \bot)$
⟨*proof*⟩

lemma *ex-ssum-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) =$
$((\exists\, x.\ P\ (sinl{\cdot}x) \wedge x \neq \bot) \vee$
$(\exists\, x.\ P\ (sinr{\cdot}x) \wedge x \neq \bot))$
⟨*proof*⟩

lemma *exh-start*: $p = \bot \vee (\exists\, x.\ p = x \wedge x \neq \bot)$
⟨*proof*⟩

lemmas *ex-defined-iffs* =
ex-ssum-defined-iff
ex-sprod-up-defined-iff
ex-sprod-defined-iff
ex-up-defined-iff
ex-one-defined-iff

Rules for turning exh into casedist

lemma *exh-casedist0*: $[\![R;\ R \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *exh-casedist1*: $((P \vee Q \Longrightarrow R) \Longrightarrow S) \equiv ([\![P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow S)$
⟨*proof*⟩

lemma *exh-casedist2*: $(\exists\, x.\ P\ x \Longrightarrow Q) \equiv (\bigwedge x.\ P\ x \Longrightarrow Q)$
⟨*proof*⟩

lemma *exh-casedist3*: $(P \wedge Q \Longrightarrow R) \equiv (P \Longrightarrow Q \Longrightarrow R)$
⟨*proof*⟩

lemmas *exh-casedists* = *exh-casedist1 exh-casedist2 exh-casedist3*

end

23 Completion: Defining bifinite domains by ideal completion

theory *Completion*
imports *Bifinite*
begin

23.1 Ideals over a preorder

locale *preorder* =
  fixes $r :: 'a::type \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *50*)
  assumes *r-refl*: $x \preceq x$
  assumes *r-trans*: $\llbracket x \preceq y; y \preceq z \rrbracket \Longrightarrow x \preceq z$
begin

definition
  ideal :: $'a\ set \Rightarrow bool$ **where**
  $ideal\ A = ((\exists x.\ x \in A) \wedge (\forall x{\in}A.\ \forall y{\in}A.\ \exists z{\in}A.\ x \preceq z \wedge y \preceq z) \wedge$
    $(\forall x\ y.\ x \preceq y \longrightarrow y \in A \longrightarrow x \in A))$

lemma *idealI*:
  assumes $\exists x.\ x \in A$
  assumes $\bigwedge x\ y.\ \llbracket x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
  assumes $\bigwedge x\ y.\ \llbracket x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
  shows *ideal A*
⟨*proof*⟩

lemma *idealD1*:
  $ideal\ A \Longrightarrow \exists x.\ x \in A$
⟨*proof*⟩

lemma *idealD2*:
  $\llbracket ideal\ A; x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
⟨*proof*⟩

lemma *idealD3*:
  $\llbracket ideal\ A; x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
⟨*proof*⟩

lemma *ideal-directed-finite*:
  assumes *A*: *ideal A*
  shows $\llbracket finite\ U;\ U \subseteq A \rrbracket \Longrightarrow \exists z{\in}A.\ \forall x{\in}U.\ x \preceq z$
⟨*proof*⟩

lemma *ideal-principal*: *ideal* $\{x.\ x \preceq z\}$
⟨*proof*⟩

lemma *ex-ideal*: $\exists A.\ ideal\ A$
⟨*proof*⟩

lemma *directed-image-ideal*:
assumes *A*: *ideal A*
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows *directed* (*f* ' *A*)
⟨*proof*⟩

lemma *lub-image-principal*:
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows $(\bigsqcup x{\in}\{x.\ x \preceq y\}.\ f\ x) = f\ y$
⟨*proof*⟩

The set of ideals is a cpo

lemma *ideal-UN*:
fixes *A* :: *nat* ⇒ *'a set*
assumes *ideal-A*: $\bigwedge i.$ *ideal* (*A i*)
assumes *chain-A*: $\bigwedge i\ j.\ i \leq j \Longrightarrow A\ i \subseteq A\ j$
shows *ideal* $(\bigcup i.\ A\ i)$
⟨*proof*⟩

lemma *typedef-ideal-po*:
fixes *Abs* :: *'a set* ⇒ *'b*::*sq-ord*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *less*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow$ *Rep x* ⊆ *Rep y*
shows *OFCLASS*(*'b*, *po-class*)
⟨*proof*⟩

lemma
fixes *Abs* :: *'a set* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *less*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow$ *Rep x* ⊆ *Rep y*
assumes *S*: *chain S*
shows *typedef-ideal-lub*: *range S* <<| *Abs* $(\bigcup i.$ *Rep* (*S i*))
and *typedef-ideal-rep-contlub*: *Rep* $(\bigsqcup i.\ S\ i) = (\bigcup i.$ *Rep* (*S i*))
⟨*proof*⟩

lemma *typedef-ideal-cpo*:
fixes *Abs* :: *'a set* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *less*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow$ *Rep x* ⊆ *Rep y*
shows *OFCLASS*(*'b*, *cpo-class*)
⟨*proof*⟩

end

interpretation *sq-le*: *preorder sq-le* :: *'a*::*po* ⇒ *'a* ⇒ *bool*
⟨*proof*⟩

23.2 Lemmas about least upper bounds

lemma *finite-directed-contains-lub*:
$[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \exists\, u{\in}S.\ S <<|\ u$
⟨*proof*⟩

lemma *lub-finite-directed-in-self*:
$[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \mathit{lub}\ S \in S$
⟨*proof*⟩

lemma *finite-directed-has-lub*: $[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \exists\, u.\ S <<|\ u$
⟨*proof*⟩

lemma *is-ub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ x \in S]\!] \Longrightarrow x \sqsubseteq \mathit{lub}\ S$
⟨*proof*⟩

lemma *is-lub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ S <|\ x]\!] \Longrightarrow \mathit{lub}\ S \sqsubseteq x$
⟨*proof*⟩

23.3 Locale for ideal completion

locale *basis-take* = *preorder* +
  fixes *take* :: *nat* ⇒ *'a*::*type* ⇒ *'a*
  assumes *take-less*: *take n a* ≼ *a*
  assumes *take-take*: *take n* (*take n a*) = *take n a*
  assumes *take-mono*: *a* ≼ *b* ⟹ *take n a* ≼ *take n b*
  assumes *take-chain*: *take n a* ≼ *take* (*Suc n*) *a*
  assumes *finite-range-take*: *finite* (*range* (*take n*))
  assumes *take-covers*: ∃ *n*. *take n a* = *a*
begin

lemma *take-chain-less*: *m* < *n* ⟹ *take m a* ≼ *take n a*
⟨*proof*⟩

lemma *take-chain-le*: *m* ≤ *n* ⟹ *take m a* ≼ *take n a*
⟨*proof*⟩

end

locale *ideal-completion* = *basis-take* +
  fixes *principal* :: *'a*::*type* ⇒ *'b*::*cpo*
  fixes *rep* :: *'b*::*cpo* ⇒ *'a*::*type set*
  assumes *ideal-rep*: ⋀*x*. *preorder.ideal r* (*rep x*)
  assumes *rep-contlub*: ⋀ *Y*. *chain Y* ⟹ *rep* (⨆ *i*. *Y i*) = (⋃ *i*. *rep* (*Y i*))
  assumes *rep-principal*: ⋀*a*. *rep* (*principal a*) = {*b*. *b* ≼ *a*}
  assumes *subset-repD*: ⋀*x y*. *rep x* ⊆ *rep y* ⟹ *x* ⊑ *y*
begin

lemma *finite-take-rep*: *finite* (*take n* ‘ *rep x*)
⟨*proof*⟩

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

lemma *less-def*: $x \sqsubseteq y \longleftrightarrow rep\ x \subseteq rep\ y$
⟨*proof*⟩

lemma *rep-eq*: $rep\ x = \{a.\ principal\ a \sqsubseteq x\}$
⟨*proof*⟩

lemma *mem-rep-iff-principal-less*: $a \in rep\ x \longleftrightarrow principal\ a \sqsubseteq x$
⟨*proof*⟩

lemma *principal-less-iff-mem-rep*: $principal\ a \sqsubseteq x \longleftrightarrow a \in rep\ x$
⟨*proof*⟩

lemma *principal-less-iff* [*simp*]: $principal\ a \sqsubseteq principal\ b \longleftrightarrow a \preceq b$
⟨*proof*⟩

lemma *principal-eq-iff*: $principal\ a = principal\ b \longleftrightarrow a \preceq b \wedge b \preceq a$
⟨*proof*⟩

lemma *repD*: $a \in rep\ x \Longrightarrow principal\ a \sqsubseteq x$
⟨*proof*⟩

lemma *principal-mono*: $a \preceq b \Longrightarrow principal\ a \sqsubseteq principal\ b$
⟨*proof*⟩

lemma *lessI*: $(\bigwedge a.\ principal\ a \sqsubseteq x \Longrightarrow principal\ a \sqsubseteq u) \Longrightarrow x \sqsubseteq u$
⟨*proof*⟩

lemma *lub-principal-rep*: $principal\ `\ rep\ x <<|\ x$
⟨*proof*⟩

23.4 Defining functions in terms of basis elements

definition
basis-fun :: $('a::type \Rightarrow 'c::cpo) \Rightarrow 'b \to 'c$ **where**
basis-fun = $(\lambda f.\ (\Lambda\ x.\ lub\ (f\ `\ rep\ x)))$

lemma *basis-fun-lemma0*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $\exists u.\ f\ `\ take\ i\ `\ rep\ x <<|\ u$
⟨*proof*⟩

lemma *basis-fun-lemma1*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$

shows *chain* (λ*i*. *lub* (*f* ‘ *take i* ‘ *rep x*))
⟨*proof*⟩

lemma *basis-fun-lemma2*:
fixes *f* :: ′*a*::*type* ⇒ ′*c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows *f* ‘ *rep x* <<| (⨆ *i*. *lub* (*f* ‘ *take i* ‘ *rep x*))
⟨*proof*⟩

lemma *basis-fun-lemma*:
fixes *f* :: ′*a*::*type* ⇒ ′*c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows ∃ *u*. *f* ‘ *rep x* <<| *u*
⟨*proof*⟩

lemma *basis-fun-beta*:
fixes *f* :: ′*a*::*type* ⇒ ′*c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows *basis-fun f*·*x* = *lub* (*f* ‘ *rep x*)
⟨*proof*⟩

lemma *basis-fun-principal*:
fixes *f* :: ′*a*::*type* ⇒ ′*c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows *basis-fun f*·(*principal a*) = *f a*
⟨*proof*⟩

lemma *basis-fun-mono*:
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
assumes *g-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *g a* ⊑ *g b*
assumes *less*: ⋀*a*. *f a* ⊑ *g a*
shows *basis-fun f* ⊑ *basis-fun g*
⟨*proof*⟩

lemma *compact-principal* [*simp*]: *compact* (*principal a*)
⟨*proof*⟩

23.5 Bifiniteness of ideal completions

definition
completion-approx :: *nat* ⇒ ′*b* → ′*b* **where**
completion-approx = (λ*i*. *basis-fun* (λ*a*. *principal* (*take i a*)))

lemma *completion-approx-beta*:
completion-approx i·*x* = (⨆ *a*∈*rep x*. *principal* (*take i a*))
⟨*proof*⟩

lemma *completion-approx-principal*:
completion-approx i·(*principal a*) = *principal* (*take i a*)

⟨*proof*⟩

lemma *chain-completion-approx*: *chain completion-approx*
⟨*proof*⟩

lemma *lub-completion-approx*: $(\bigsqcup i.\ \textit{completion-approx}\ i{\cdot}x) = x$
⟨*proof*⟩

lemma *completion-approx-eq-principal*:
$\exists\, a{\in}\textit{rep}\ x.\ \textit{completion-approx}\ i{\cdot}x = \textit{principal}\ (\textit{take}\ i\ a)$
⟨*proof*⟩

lemma *completion-approx-idem*:
$\textit{completion-approx}\ i{\cdot}(\textit{completion-approx}\ i{\cdot}x) = \textit{completion-approx}\ i{\cdot}x$
⟨*proof*⟩

lemma *finite-fixes-completion-approx*:
finite $\{x.\ \textit{completion-approx}\ i{\cdot}x = x\}$ (**is** *finite ?S*)
⟨*proof*⟩

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes *P*: $\bigwedge a.\ P\ (\textit{principal}\ a)$
shows *P x*
⟨*proof*⟩

lemma *principal-induct2*:
$[\![\bigwedge y.\ \textit{adm}\ (\lambda x.\ P\ x\ y);\ \bigwedge x.\ \textit{adm}\ (\lambda y.\ P\ x\ y);$
$\bigwedge a\ b.\ P\ (\textit{principal}\ a)\ (\textit{principal}\ b)]\!] \Longrightarrow P\ x\ y$
⟨*proof*⟩

lemma *compact-imp-principal*: $\textit{compact}\ x \Longrightarrow \exists\, a.\ x = \textit{principal}\ a$
⟨*proof*⟩

end

end

24 CompactBasis: Compact bases of domains

theory *CompactBasis*
imports *Completion*
begin

24.1 Compact bases of bifinite domains

defaultsort *profinite*

typedef (**open**) *'a compact-basis* = $\{x::'a::profinite.\ compact\ x\}$
⟨*proof*⟩

lemma *compact-Rep-compact-basis*: *compact* (*Rep-compact-basis a*)
⟨*proof*⟩

instantiation *compact-basis* :: (*profinite*) *sq-ord*
begin

definition
compact-le-def:
(*op* $\sqsubseteq$) ≡ ($\lambda x\ y.$ *Rep-compact-basis x* $\sqsubseteq$ *Rep-compact-basis y*)

instance ⟨*proof*⟩

end

instance *compact-basis* :: (*profinite*) *po*
⟨*proof*⟩

Take function for compact basis

definition
compact-take :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis* **where**
compact-take = ($\lambda n\ a.$ *Abs-compact-basis* (*approx n*·(*Rep-compact-basis a*)))

lemma *Rep-compact-take*:
Rep-compact-basis (*compact-take n a*) = *approx n*·(*Rep-compact-basis a*)
⟨*proof*⟩

lemmas *approx-Rep-compact-basis* = *Rep-compact-take* [*symmetric*]

interpretation *compact-basis*:
basis-take sq-le compact-take
⟨*proof*⟩

Ideal completion

definition
approximants :: *'a* ⇒ *'a compact-basis set* **where**
approximants = ($\lambda x.$ $\{a.$ *Rep-compact-basis a* $\sqsubseteq$ *x*$\}$)

interpretation *compact-basis*:
ideal-completion sq-le compact-take Rep-compact-basis approximants
⟨*proof*⟩

minimal compact element

definition
compact-bot :: *'a*::*bifinite compact-basis* **where**
compact-bot = *Abs-compact-basis* $\bot$

lemma *Rep-compact-bot*: *Rep-compact-basis compact-bot* = ⊥
⟨*proof*⟩

lemma *compact-bot-minimal* [*simp*]: *compact-bot* ⊑ *a*
⟨*proof*⟩

24.2 A compact basis for powerdomains

typedef *'a pd-basis* =
{*S*::*'a*::*profinite compact-basis set*. *finite S* ∧ *S* ≠ {}}
⟨*proof*⟩

lemma *finite-Rep-pd-basis* [*simp*]: *finite* (*Rep-pd-basis u*)
⟨*proof*⟩

lemma *Rep-pd-basis-nonempty* [*simp*]: *Rep-pd-basis u* ≠ {}
⟨*proof*⟩

unit and plus

definition
PDUnit :: *'a compact-basis* ⇒ *'a pd-basis* **where**
PDUnit = (λ*x*. *Abs-pd-basis* {*x*})

definition
PDPlus :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* ∪ *Rep-pd-basis u*)

lemma *Rep-PDUnit*:
Rep-pd-basis (*PDUnit x*) = {*x*}
⟨*proof*⟩

lemma *Rep-PDPlus*:
Rep-pd-basis (*PDPlus u v*) = *Rep-pd-basis u* ∪ *Rep-pd-basis v*
⟨*proof*⟩

lemma *PDUnit-inject* [*simp*]: (*PDUnit a* = *PDUnit b*) = (*a* = *b*)
⟨*proof*⟩

lemma *PDPlus-assoc*: *PDPlus* (*PDPlus t u*) *v* = *PDPlus t* (*PDPlus u v*)
⟨*proof*⟩

lemma *PDPlus-commute*: *PDPlus t u* = *PDPlus u t*
⟨*proof*⟩

lemma *PDPlus-absorb*: *PDPlus t t* = *t*
⟨*proof*⟩

lemma *pd-basis-induct1*:
assumes *PDUnit*: ⋀*a*. *P* (*PDUnit a*)

assumes *PDPlus*: $\bigwedge a\ t.\ P\ t \Longrightarrow P\ (PDPlus\ (PDUnit\ a)\ t)$
shows $P\ x$
⟨*proof*⟩

lemma *pd-basis-induct*:
assumes *PDUnit*: $\bigwedge a.\ P\ (PDUnit\ a)$
assumes *PDPlus*: $\bigwedge t\ u.\ [\![P\ t;\ P\ u]\!] \Longrightarrow P\ (PDPlus\ t\ u)$
shows $P\ x$
⟨*proof*⟩

fold-pd

definition
fold-pd ::
$('a\ compact\text{-}basis \Rightarrow 'b::type) \Rightarrow ('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ pd\text{-}basis \Rightarrow 'b$
where *fold-pd g f t* = *fold1 f* (*g* ‘ *Rep-pd-basis t*)

lemma *fold-pd-PDUnit*:
assumes *ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDUnit x*) = *g x*
⟨*proof*⟩

lemma *fold-pd-PDPlus*:
assumes *ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDPlus t u*) = *f* (*fold-pd g f t*) (*fold-pd g f u*)
⟨*proof*⟩

Take function for powerdomain basis

definition
pd-take :: *nat* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
pd-take n = (λ*t*. *Abs-pd-basis* (*compact-take n* ‘ *Rep-pd-basis t*))

lemma *Rep-pd-take*:
Rep-pd-basis (*pd-take n t*) = *compact-take n* ‘ *Rep-pd-basis t*
⟨*proof*⟩

lemma *pd-take-simps* [*simp*]:
pd-take n (*PDUnit a*) = *PDUnit* (*compact-take n a*)
pd-take n (*PDPlus t u*) = *PDPlus* (*pd-take n t*) (*pd-take n u*)
⟨*proof*⟩

lemma *pd-take-idem*: *pd-take n* (*pd-take n t*) = *pd-take n t*
⟨*proof*⟩

lemma *finite-range-pd-take*: *finite* (*range* (*pd-take n*))
⟨*proof*⟩

lemma *pd-take-covers*: ∃ *n*. *pd-take n t* = *t*
⟨*proof*⟩

end

25 UpperPD: Upper powerdomain

theory *UpperPD*
imports *CompactBasis*
begin

25.1 Basis preorder

definition
upper-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♯ *50*) **where**
upper-le = (λ*u v*. ∀ *y*∈*Rep-pd-basis v*. ∃ *x*∈*Rep-pd-basis u*. *x* ⊑ *y*)

lemma *upper-le-refl* [*simp*]: *t* ≤♯ *t*
⟨*proof*⟩

lemma *upper-le-trans*: ⟦*t* ≤♯ *u*; *u* ≤♯ *v*⟧ ⟹ *t* ≤♯ *v*
⟨*proof*⟩

interpretation *upper-le*: *preorder upper-le*
⟨*proof*⟩

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* ≤♯ *t*
⟨*proof*⟩

lemma *PDUnit-upper-mono*: *x* ⊑ *y* ⟹ *PDUnit x* ≤♯ *PDUnit y*
⟨*proof*⟩

lemma *PDPlus-upper-mono*: ⟦*s* ≤♯ *t*; *u* ≤♯ *v*⟧ ⟹ *PDPlus s u* ≤♯ *PDPlus t v*
⟨*proof*⟩

lemma *PDPlus-upper-less*: *PDPlus t u* ≤♯ *t*
⟨*proof*⟩

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* ≤♯ *PDUnit b*) = *a* ⊑ *b*
⟨*proof*⟩

lemma *upper-le-PDPlus-PDUnit-iff*:
(*PDPlus t u* ≤♯ *PDUnit a*) = (*t* ≤♯ *PDUnit a* ∨ *u* ≤♯ *PDUnit a*)
⟨*proof*⟩

lemma *upper-le-PDPlus-iff*: (*t* ≤♯ *PDPlus u v*) = (*t* ≤♯ *u* ∧ *t* ≤♯ *v*)
⟨*proof*⟩

lemma *upper-le-induct* [*induct set*: *upper-le*]:
assumes *le*: *t* ≤♯ *u*

assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *2*: $\bigwedge t\ u\ a.\ P\ t\ (PDUnit\ a) \Longrightarrow P\ (PDPlus\ t\ u)\ (PDUnit\ a)$
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ t\ v]\!] \Longrightarrow P\ t\ (PDPlus\ u\ v)$
shows $P\ t\ u$
⟨*proof*⟩

lemma *pd-take-upper-chain*:
$pd\text{-}take\ n\ t \leq\sharp\ pd\text{-}take\ (Suc\ n)\ t$
⟨*proof*⟩

lemma *pd-take-upper-le*: $pd\text{-}take\ i\ t \leq\sharp\ t$
⟨*proof*⟩

lemma *pd-take-upper-mono*:
$t \leq\sharp\ u \Longrightarrow pd\text{-}take\ n\ t \leq\sharp\ pd\text{-}take\ n\ u$
⟨*proof*⟩

25.2 Type definition

typedef (**open**) *'a upper-pd* =
$\{S::'a\ pd\text{-}basis\ set.\ upper\text{-}le.ideal\ S\}$
⟨*proof*⟩

instantiation *upper-pd* :: (*profinite*) *sq-ord*
begin

definition
$x \sqsubseteq y \longleftrightarrow Rep\text{-}upper\text{-}pd\ x \subseteq Rep\text{-}upper\text{-}pd\ y$

instance ⟨*proof*⟩
end

instance *upper-pd* :: (*profinite*) *po*
⟨*proof*⟩

instance *upper-pd* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-upper-pd-lub*:
$chain\ Y \Longrightarrow Rep\text{-}upper\text{-}pd\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ Rep\text{-}upper\text{-}pd\ (Y\ i))$
⟨*proof*⟩

lemma *ideal-Rep-upper-pd*: $upper\text{-}le.ideal\ (Rep\text{-}upper\text{-}pd\ xs)$
⟨*proof*⟩

definition
$upper\text{-}principal :: 'a\ pd\text{-}basis \Rightarrow 'a\ upper\text{-}pd$ **where**
$upper\text{-}principal\ t = Abs\text{-}upper\text{-}pd\ \{u.\ u \leq\sharp\ t\}$

lemma *Rep-upper-principal*:
Rep-upper-pd (*upper-principal* t) = $\{u.\ u \leq\sharp\ t\}$
⟨*proof*⟩

interpretation *upper-pd*:
ideal-completion upper-le pd-take upper-principal Rep-upper-pd
⟨*proof*⟩

Upper powerdomain is pointed

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*
⟨*proof*⟩

instance *upper-pd* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-upper-pd-pcpo*: $\bot$ = *upper-principal* (*PDUnit compact-bot*)
⟨*proof*⟩

Upper powerdomain is profinite

instantiation *upper-pd* :: (*profinite*) *profinite*
begin

definition
approx-upper-pd-def: *approx* = *upper-pd.completion-approx*

instance
⟨*proof*⟩

end

instance *upper-pd* :: (*bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-upper-principal* [*simp*]:
approx n·(*upper-principal* t) = *upper-principal* (*pd-take* n t)
⟨*proof*⟩

lemma *approx-eq-upper-principal*:
$\exists\, t{\in}$*Rep-upper-pd xs*. *approx* n·*xs* = *upper-principal* (*pd-take* n t)
⟨*proof*⟩

25.3 Monadic unit and plus

definition
upper-unit :: $'a \to 'a$ *upper-pd* **where**
upper-unit = *compact-basis.basis-fun* ($\lambda a.$ *upper-principal* (*PDUnit* a))

definition
upper-plus :: $'a$ *upper-pd* $\to$ $'a$ *upper-pd* $\to$ $'a$ *upper-pd* **where**
upper-plus = *upper-pd.basis-fun* ($\lambda t.$ *upper-pd.basis-fun* ($\lambda u.$

upper-principal (*PDPlus t u*)))

abbreviation
upper-add :: *'a upper-pd* ⇒ *'a upper-pd* ⇒ *'a upper-pd*
(**infixl** +♯ *65*) **where**
xs +♯ *ys* == *upper-plus·xs·ys*

syntax
-upper-pd :: *args* ⇒ *'a upper-pd* ({-}♯)

translations
{*x*,*xs*}♯ == {*x*}♯ +♯ {*xs*}♯
{*x*}♯ == *CONST upper-unit·x*

lemma *upper-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♯ = *upper-principal* (*PDUnit a*)
⟨*proof*⟩

lemma *upper-plus-principal* [*simp*]:
upper-principal t +♯ *upper-principal u* = *upper-principal* (*PDPlus t u*)
⟨*proof*⟩

lemma *approx-upper-unit* [*simp*]:
approx n·{*x*}♯ = {*approx n·x*}♯
⟨*proof*⟩

lemma *approx-upper-plus* [*simp*]:
approx n·(*xs* +♯ *ys*) = (*approx n·xs*) +♯ (*approx n·ys*)
⟨*proof*⟩

lemma *upper-plus-assoc*: (*xs* +♯ *ys*) +♯ *zs* = *xs* +♯ (*ys* +♯ *zs*)
⟨*proof*⟩

lemma *upper-plus-commute*: *xs* +♯ *ys* = *ys* +♯ *xs*
⟨*proof*⟩

lemma *upper-plus-absorb* [*simp*]: *xs* +♯ *xs* = *xs*
⟨*proof*⟩

lemma *upper-plus-left-commute*: *xs* +♯ (*ys* +♯ *zs*) = *ys* +♯ (*xs* +♯ *zs*)
⟨*proof*⟩

lemma *upper-plus-left-absorb* [*simp*]: *xs* +♯ (*xs* +♯ *ys*) = *xs* +♯ *ys*
⟨*proof*⟩

Useful for *simp add*: *upper-plus-ac*

lemmas *upper-plus-ac* =
upper-plus-assoc upper-plus-commute upper-plus-left-commute

Useful for *simp only*: *upper-plus-aci*

lemmas *upper-plus-aci* =
upper-plus-ac upper-plus-absorb upper-plus-left-absorb

lemma *upper-plus-less1*: $xs +\sharp ys \sqsubseteq xs$
⟨*proof*⟩

lemma *upper-plus-less2*: $xs +\sharp ys \sqsubseteq ys$
⟨*proof*⟩

lemma *upper-plus-greatest*: $[\![xs \sqsubseteq ys; xs \sqsubseteq zs]\!] \Longrightarrow xs \sqsubseteq ys +\sharp zs$
⟨*proof*⟩

lemma *upper-less-plus-iff*:
$xs \sqsubseteq ys +\sharp zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
⟨*proof*⟩

lemma *upper-plus-less-unit-iff*:
$xs +\sharp ys \sqsubseteq \{z\}\sharp \longleftrightarrow xs \sqsubseteq \{z\}\sharp \vee ys \sqsubseteq \{z\}\sharp$
⟨*proof*⟩

lemma *upper-unit-less-iff* [*simp*]: $\{x\}\sharp \sqsubseteq \{y\}\sharp \longleftrightarrow x \sqsubseteq y$
⟨*proof*⟩

lemmas *upper-pd-less-simps* =
upper-unit-less-iff
upper-less-plus-iff
upper-plus-less-unit-iff

lemma *upper-unit-eq-iff* [*simp*]: $\{x\}\sharp = \{y\}\sharp \longleftrightarrow x = y$
⟨*proof*⟩

lemma *upper-unit-strict* [*simp*]: $\{\bot\}\sharp = \bot$
⟨*proof*⟩

lemma *upper-plus-strict1* [*simp*]: $\bot +\sharp ys = \bot$
⟨*proof*⟩

lemma *upper-plus-strict2* [*simp*]: $xs +\sharp \bot = \bot$
⟨*proof*⟩

lemma *upper-unit-strict-iff* [*simp*]: $\{x\}\sharp = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *upper-plus-strict-iff* [*simp*]:
$xs +\sharp ys = \bot \longleftrightarrow xs = \bot \vee ys = \bot$
⟨*proof*⟩

lemma *compact-upper-unit-iff* [*simp*]: *compact* $\{x\}\sharp \longleftrightarrow$ *compact* x
⟨*proof*⟩

lemma *compact-upper-plus* [*simp*]:
⟦*compact xs*; *compact ys*⟧ ⟹ *compact* (*xs* +♯ *ys*)
⟨*proof*⟩

25.4 Induction rules

lemma *upper-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: ⋀*x*. *P* {*x*}♯
assumes *insert*: ⋀*x ys*. ⟦*P* {*x*}♯; *P ys*⟧ ⟹ *P* ({*x*}♯ +♯ *ys*)
shows *P* (*xs*::′*a upper-pd*)
⟨*proof*⟩

lemma *upper-pd-induct*:
assumes *P*: *adm P*
assumes *unit*: ⋀*x*. *P* {*x*}♯
assumes *plus*: ⋀*xs ys*. ⟦*P xs*; *P ys*⟧ ⟹ *P* (*xs* +♯ *ys*)
shows *P* (*xs*::′*a upper-pd*)
⟨*proof*⟩

25.5 Monadic bind

definition
upper-bind-basis ::
′*a pd-basis* ⇒ (′*a* → ′*b upper-pd*) → ′*b upper-pd* **where**
upper-bind-basis = *fold-pd*
(λ*a*. Λ *f*. *f*·(*Rep-compact-basis a*))
(λ*x y*. Λ *f*. *x*·*f* +♯ *y*·*f*)

lemma *ACI-upper-bind*:
ab-semigroup-idem-mult (λ*x y*. Λ *f*. *x*·*f* +♯ *y*·*f*)
⟨*proof*⟩

lemma *upper-bind-basis-simps* [*simp*]:
upper-bind-basis (*PDUnit a*) =
(Λ *f*. *f*·(*Rep-compact-basis a*))
upper-bind-basis (*PDPlus t u*) =
(Λ *f*. *upper-bind-basis t*·*f* +♯ *upper-bind-basis u*·*f*)
⟨*proof*⟩

lemma *upper-bind-basis-mono*:
t ≤♯ *u* ⟹ *upper-bind-basis t* ⊑ *upper-bind-basis u*
⟨*proof*⟩

definition
upper-bind :: ′*a upper-pd* → (′*a* → ′*b upper-pd*) → ′*b upper-pd* **where**
upper-bind = *upper-pd.basis-fun upper-bind-basis*

lemma *upper-bind-principal* [*simp*]:

upper-bind·(*upper-principal t*) = *upper-bind-basis t*
⟨*proof*⟩

lemma *upper-bind-unit* [*simp*]:
upper-bind·{*x*}♯·*f* = *f*·*x*
⟨*proof*⟩

lemma *upper-bind-plus* [*simp*]:
upper-bind·(*xs* +♯ *ys*)·*f* = *upper-bind*·*xs*·*f* +♯ *upper-bind*·*ys*·*f*
⟨*proof*⟩

lemma *upper-bind-strict* [*simp*]: *upper-bind*·⊥·*f* = *f*·⊥
⟨*proof*⟩

25.6 Map and join

definition
upper-map :: ('*a* → '*b*) → '*a upper-pd* → '*b upper-pd* **where**
upper-map = (Λ *f xs*. *upper-bind*·*xs*·(Λ *x*. {*f*·*x*}♯))

definition
upper-join :: '*a upper-pd upper-pd* → '*a upper-pd* **where**
upper-join = (Λ *xss*. *upper-bind*·*xss*·(Λ *xs*. *xs*))

lemma *upper-map-unit* [*simp*]:
upper-map·*f*·{*x*}♯ = {*f*·*x*}♯
⟨*proof*⟩

lemma *upper-map-plus* [*simp*]:
upper-map·*f*·(*xs* +♯ *ys*) = *upper-map*·*f*·*xs* +♯ *upper-map*·*f*·*ys*
⟨*proof*⟩

lemma *upper-join-unit* [*simp*]:
upper-join·{*xs*}♯ = *xs*
⟨*proof*⟩

lemma *upper-join-plus* [*simp*]:
upper-join·(*xss* +♯ *yss*) = *upper-join*·*xss* +♯ *upper-join*·*yss*
⟨*proof*⟩

lemma *upper-map-ident*: *upper-map*·(Λ *x*. *x*)·*xs* = *xs*
⟨*proof*⟩

lemma *upper-map-map*:
upper-map·*f*·(*upper-map*·*g*·*xs*) = *upper-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
⟨*proof*⟩

lemma *upper-join-map-unit*:
upper-join·(*upper-map*·*upper-unit*·*xs*) = *xs*

⟨*proof*⟩

lemma *upper-join-map-join*:
upper-join·(*upper-map*·*upper-join*·*xsss*) = *upper-join*·(*upper-join*·*xsss*)
⟨*proof*⟩

lemma *upper-join-map-map*:
upper-join·(*upper-map*·(*upper-map*·*f*)·*xss*) =
upper-map·*f*·(*upper-join*·*xss*)
⟨*proof*⟩

lemma *upper-map-approx*: *upper-map*·(*approx n*)·*xs* = *approx n*·*xs*
⟨*proof*⟩

end

26 LowerPD: Lower powerdomain

theory *LowerPD*
imports *CompactBasis*
begin

26.1 Basis preorder

definition
lower-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♭ *50*) **where**
lower-le = (λ*u v*. ∀ *x*∈*Rep-pd-basis u*. ∃ *y*∈*Rep-pd-basis v*. *x* ⊑ *y*)

lemma *lower-le-refl* [*simp*]: *t* ≤♭ *t*
⟨*proof*⟩

lemma *lower-le-trans*: ⟦*t* ≤♭ *u*; *u* ≤♭ *v*⟧ ⟹ *t* ≤♭ *v*
⟨*proof*⟩

interpretation *lower-le*: *preorder lower-le*
⟨*proof*⟩

lemma *lower-le-minimal* [*simp*]: *PDUnit compact-bot* ≤♭ *t*
⟨*proof*⟩

lemma *PDUnit-lower-mono*: *x* ⊑ *y* ⟹ *PDUnit x* ≤♭ *PDUnit y*
⟨*proof*⟩

lemma *PDPlus-lower-mono*: ⟦*s* ≤♭ *t*; *u* ≤♭ *v*⟧ ⟹ *PDPlus s u* ≤♭ *PDPlus t v*
⟨*proof*⟩

lemma *PDPlus-lower-less*: *t* ≤♭ *PDPlus t u*
⟨*proof*⟩

lemma *lower-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* $\leq\flat$ *PDUnit b*) = $a \sqsubseteq b$
⟨*proof*⟩

lemma *lower-le-PDUnit-PDPlus-iff*:
(*PDUnit a* $\leq\flat$ *PDPlus t u*) = (*PDUnit a* $\leq\flat$ *t* $\vee$ *PDUnit a* $\leq\flat$ *u*)
⟨*proof*⟩

lemma *lower-le-PDPlus-iff*: (*PDPlus t u* $\leq\flat$ *v*) = (*t* $\leq\flat$ *v* $\wedge$ *u* $\leq\flat$ *v*)
⟨*proof*⟩

lemma *lower-le-induct* [*induct set*: *lower-le*]:
assumes *le*: *t* $\leq\flat$ *u*
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P$ (*PDUnit a*) (*PDUnit b*)
assumes *2*: $\bigwedge t\ u\ a.\ P$ (*PDUnit a*) *t* $\Longrightarrow P$ (*PDUnit a*) (*PDPlus t u*)
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ v;\ P\ u\ v]\!] \Longrightarrow P$ (*PDPlus t u*) *v*
shows *P t u*
⟨*proof*⟩

lemma *pd-take-lower-chain*:
pd-take n t $\leq\flat$ *pd-take* (*Suc n*) *t*
⟨*proof*⟩

lemma *pd-take-lower-le*: *pd-take i t* $\leq\flat$ *t*
⟨*proof*⟩

lemma *pd-take-lower-mono*:
t $\leq\flat$ *u* $\Longrightarrow$ *pd-take n t* $\leq\flat$ *pd-take n u*
⟨*proof*⟩

26.2 Type definition

typedef (**open**) *'a lower-pd* =
{*S*::*'a pd-basis set*. *lower-le.ideal S*}
⟨*proof*⟩

instantiation *lower-pd* :: (*profinite*) *sq-ord*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-lower-pd x* $\subseteq$ *Rep-lower-pd y*

instance ⟨*proof*⟩
end

instance *lower-pd* :: (*profinite*) *po*
⟨*proof*⟩

instance *lower-pd* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-lower-pd-lub*:
chain Y ⟹ *Rep-lower-pd* (⨆ *i*. *Y i*) = (⋃ *i*. *Rep-lower-pd* (*Y i*))
⟨*proof*⟩

lemma *ideal-Rep-lower-pd*: *lower-le.ideal* (*Rep-lower-pd xs*)
⟨*proof*⟩

definition
lower-principal :: *'a pd-basis* ⇒ *'a lower-pd* **where**
lower-principal t = *Abs-lower-pd* {*u*. *u* ≤♭ *t*}

lemma *Rep-lower-principal*:
Rep-lower-pd (*lower-principal t*) = {*u*. *u* ≤♭ *t*}
⟨*proof*⟩

interpretation *lower-pd*:
ideal-completion lower-le pd-take lower-principal Rep-lower-pd
⟨*proof*⟩

Lower powerdomain is pointed

lemma *lower-pd-minimal*: *lower-principal* (*PDUnit compact-bot*) ⊑ *ys*
⟨*proof*⟩

instance *lower-pd* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-lower-pd-pcpo*: ⊥ = *lower-principal* (*PDUnit compact-bot*)
⟨*proof*⟩

Lower powerdomain is profinite

instantiation *lower-pd* :: (*profinite*) *profinite*
begin

definition
approx-lower-pd-def: *approx* = *lower-pd.completion-approx*

instance
⟨*proof*⟩

end

instance *lower-pd* :: (*bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-lower-principal* [*simp*]:
approx n·(*lower-principal t*) = *lower-principal* (*pd-take n t*)
⟨*proof*⟩

lemma *approx-eq-lower-principal*:
$\exists\, t{\in}$*Rep-lower-pd xs*. *approx n·xs* = *lower-principal* (*pd-take n t*)
⟨*proof*⟩

26.3 Monadic unit and plus

definition
lower-unit :: *'a* → *'a lower-pd* **where**
lower-unit = *compact-basis.basis-fun* (λ*a*. *lower-principal* (*PDUnit a*))

definition
lower-plus :: *'a lower-pd* → *'a lower-pd* → *'a lower-pd* **where**
lower-plus = *lower-pd.basis-fun* (λ*t*. *lower-pd.basis-fun* (λ*u*.
lower-principal (*PDPlus t u*)))

abbreviation
lower-add :: *'a lower-pd* ⇒ *'a lower-pd* ⇒ *'a lower-pd*
(**infixl** +♭ *65*) **where**
xs +♭ *ys* == *lower-plus·xs·ys*

syntax
-lower-pd :: *args* ⇒ *'a lower-pd* ({-}♭)

translations
{*x*,*xs*}♭ == {*x*}♭ +♭ {*xs*}♭
{*x*}♭ == *CONST lower-unit·x*

lemma *lower-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♭ = *lower-principal* (*PDUnit a*)
⟨*proof*⟩

lemma *lower-plus-principal* [*simp*]:
lower-principal t +♭ *lower-principal u* = *lower-principal* (*PDPlus t u*)
⟨*proof*⟩

lemma *approx-lower-unit* [*simp*]:
approx n·{*x*}♭ = {*approx n·x*}♭
⟨*proof*⟩

lemma *approx-lower-plus* [*simp*]:
approx n·(*xs* +♭ *ys*) = (*approx n·xs*) +♭ (*approx n·ys*)
⟨*proof*⟩

lemma *lower-plus-assoc*: (*xs* +♭ *ys*) +♭ *zs* = *xs* +♭ (*ys* +♭ *zs*)
⟨*proof*⟩

lemma *lower-plus-commute*: *xs* +♭ *ys* = *ys* +♭ *xs*
⟨*proof*⟩

lemma *lower-plus-absorb* [*simp*]: $xs +\flat xs = xs$
⟨*proof*⟩

lemma *lower-plus-left-commute*: $xs +\flat (ys +\flat zs) = ys +\flat (xs +\flat zs)$
⟨*proof*⟩

lemma *lower-plus-left-absorb* [*simp*]: $xs +\flat (xs +\flat ys) = xs +\flat ys$
⟨*proof*⟩

Useful for *simp add*: *lower-plus-ac*

lemmas *lower-plus-ac* =
lower-plus-assoc lower-plus-commute lower-plus-left-commute

Useful for *simp only*: *lower-plus-aci*

lemmas *lower-plus-aci* =
lower-plus-ac lower-plus-absorb lower-plus-left-absorb

lemma *lower-plus-less1*: $xs \sqsubseteq xs +\flat ys$
⟨*proof*⟩

lemma *lower-plus-less2*: $ys \sqsubseteq xs +\flat ys$
⟨*proof*⟩

lemma *lower-plus-least*: $[\![xs \sqsubseteq zs;\ ys \sqsubseteq zs]\!] \Longrightarrow xs +\flat ys \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-plus-less-iff*:
$xs +\flat ys \sqsubseteq zs \longleftrightarrow xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-unit-less-plus-iff*:
$\{x\}\flat \sqsubseteq ys +\flat zs \longleftrightarrow \{x\}\flat \sqsubseteq ys \vee \{x\}\flat \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-unit-less-iff* [*simp*]: $\{x\}\flat \sqsubseteq \{y\}\flat \longleftrightarrow x \sqsubseteq y$
⟨*proof*⟩

lemmas *lower-pd-less-simps* =
lower-unit-less-iff
lower-plus-less-iff
lower-unit-less-plus-iff

lemma *lower-unit-eq-iff* [*simp*]: $\{x\}\flat = \{y\}\flat \longleftrightarrow x = y$
⟨*proof*⟩

lemma *lower-unit-strict* [*simp*]: $\{\bot\}\flat = \bot$
⟨*proof*⟩

lemma *lower-unit-strict-iff* [*simp*]: $\{x\}\flat = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *lower-plus-strict-iff* [*simp*]:
$xs +\flat ys = \bot \longleftrightarrow xs = \bot \wedge ys = \bot$
⟨*proof*⟩

lemma *lower-plus-strict1* [*simp*]: $\bot +\flat ys = ys$
⟨*proof*⟩

lemma *lower-plus-strict2* [*simp*]: $xs +\flat \bot = xs$
⟨*proof*⟩

lemma *compact-lower-unit-iff* [*simp*]: *compact* $\{x\}\flat \longleftrightarrow$ *compact* x
⟨*proof*⟩

lemma *compact-lower-plus* [*simp*]:
$[\![$*compact* xs; *compact* $ys]\!] \Longrightarrow$ *compact* $(xs +\flat ys)$
⟨*proof*⟩

26.4 Induction rules

lemma *lower-pd-induct1*:
assumes P: *adm* P
assumes *unit*: $\bigwedge x.\ P\ \{x\}\flat$
assumes *insert*:
$\bigwedge x\ ys.\ [\![P\ \{x\}\flat; P\ ys]\!] \Longrightarrow P\ (\{x\}\flat +\flat ys)$
shows P (xs::'*a lower-pd*)
⟨*proof*⟩

lemma *lower-pd-induct*:
assumes P: *adm* P
assumes *unit*: $\bigwedge x.\ P\ \{x\}\flat$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs; P\ ys]\!] \Longrightarrow P\ (xs +\flat ys)$
shows P (xs::'*a lower-pd*)
⟨*proof*⟩

26.5 Monadic bind

definition
lower-bind-basis ::
'*a pd-basis* ⇒ ('*a* → '*b lower-pd*) → '*b lower-pd* **where**
lower-bind-basis = *fold-pd*
$(\lambda a.\ \Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\flat y{\cdot}f)$

lemma *ACI-lower-bind*:
ab-semigroup-idem-mult $(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\flat y{\cdot}f)$
⟨*proof*⟩

lemma *lower-bind-basis-simps* [*simp*]:
lower-bind-basis (*PDUnit a*) =
(Λ *f*. *f*·(*Rep-compact-basis a*))
lower-bind-basis (*PDPlus t u*) =
(Λ *f*. *lower-bind-basis t*·*f* +♭ *lower-bind-basis u*·*f*)
⟨*proof*⟩

lemma *lower-bind-basis-mono*:
t ≤♭ *u* ⟹ *lower-bind-basis t* ⊑ *lower-bind-basis u*
⟨*proof*⟩

definition
lower-bind :: *'a lower-pd* → (*'a* → *'b lower-pd*) → *'b lower-pd* **where**
lower-bind = *lower-pd.basis-fun lower-bind-basis*

lemma *lower-bind-principal* [*simp*]:
lower-bind·(*lower-principal t*) = *lower-bind-basis t*
⟨*proof*⟩

lemma *lower-bind-unit* [*simp*]:
lower-bind·{*x*}♭·*f* = *f*·*x*
⟨*proof*⟩

lemma *lower-bind-plus* [*simp*]:
lower-bind·(*xs* +♭ *ys*)·*f* = *lower-bind*·*xs*·*f* +♭ *lower-bind*·*ys*·*f*
⟨*proof*⟩

lemma *lower-bind-strict* [*simp*]: *lower-bind*·⊥·*f* = *f*·⊥
⟨*proof*⟩

26.6 Map and join

definition
lower-map :: (*'a* → *'b*) → *'a lower-pd* → *'b lower-pd* **where**
lower-map = (Λ *f xs*. *lower-bind*·*xs*·(Λ *x*. {*f*·*x*}♭))

definition
lower-join :: *'a lower-pd lower-pd* → *'a lower-pd* **where**
lower-join = (Λ *xss*. *lower-bind*·*xss*·(Λ *xs*. *xs*))

lemma *lower-map-unit* [*simp*]:
lower-map·*f*·{*x*}♭ = {*f*·*x*}♭
⟨*proof*⟩

lemma *lower-map-plus* [*simp*]:
lower-map·*f*·(*xs* +♭ *ys*) = *lower-map*·*f*·*xs* +♭ *lower-map*·*f*·*ys*
⟨*proof*⟩

lemma *lower-join-unit* [*simp*]:

lower-join·{*xs*}♭ = *xs*
⟨*proof*⟩

lemma *lower-join-plus* [*simp*]:
lower-join·(*xss* +♭ *yss*) = *lower-join*·*xss* +♭ *lower-join*·*yss*
⟨*proof*⟩

lemma *lower-map-ident*: *lower-map*·(Λ *x*. *x*)·*xs* = *xs*
⟨*proof*⟩

lemma *lower-map-map*:
lower-map·*f*·(*lower-map*·*g*·*xs*) = *lower-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
⟨*proof*⟩

lemma *lower-join-map-unit*:
lower-join·(*lower-map*·*lower-unit*·*xs*) = *xs*
⟨*proof*⟩

lemma *lower-join-map-join*:
lower-join·(*lower-map*·*lower-join*·*xsss*) = *lower-join*·(*lower-join*·*xsss*)
⟨*proof*⟩

lemma *lower-join-map-map*:
lower-join·(*lower-map*·(*lower-map*·*f*)·*xss*) =
lower-map·*f*·(*lower-join*·*xss*)
⟨*proof*⟩

lemma *lower-map-approx*: *lower-map*·(*approx n*)·*xs* = *approx n*·*xs*
⟨*proof*⟩

end

27 ConvexPD: Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

27.1 Basis preorder

definition
convex-le :: 'a *pd-basis* ⇒ 'a *pd-basis* ⇒ *bool* (**infix** ≤♮ *50*) **where**
convex-le = (λ*u v*. *u* ≤♯ *v* ∧ *u* ≤♭ *v*)

lemma *convex-le-refl* [*simp*]: *t* ≤♮ *t*
⟨*proof*⟩

lemma *convex-le-trans*: ⟦*t* ≤♮ *u*; *u* ≤♮ *v*⟧ ⟹ *t* ≤♮ *v*

⟨*proof*⟩

interpretation *convex-le*: *preorder convex-le*
⟨*proof*⟩

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\natural$ *t*
⟨*proof*⟩

lemma *PDUnit-convex-mono*: $x \sqsubseteq y \Longrightarrow$ *PDUnit x* $\leq\natural$ *PDUnit y*
⟨*proof*⟩

lemma *PDPlus-convex-mono*: $[\![s \leq\natural t;\ u \leq\natural v]\!] \Longrightarrow$ *PDPlus s u* $\leq\natural$ *PDPlus t v*
⟨*proof*⟩

lemma *convex-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* $\leq\natural$ *PDUnit b*) = $a \sqsubseteq b$
⟨*proof*⟩

lemma *convex-le-PDUnit-lemma1*:
(*PDUnit a* $\leq\natural$ *t*) = ($\forall$ *b*∈*Rep-pd-basis t*. $a \sqsubseteq b$)
⟨*proof*⟩

lemma *convex-le-PDUnit-PDPlus-iff* [*simp*]:
(*PDUnit a* $\leq\natural$ *PDPlus t u*) = (*PDUnit a* $\leq\natural$ *t* $\wedge$ *PDUnit a* $\leq\natural$ *u*)
⟨*proof*⟩

lemma *convex-le-PDUnit-lemma2*:
(*t* $\leq\natural$ *PDUnit b*) = ($\forall$ *a*∈*Rep-pd-basis t*. $a \sqsubseteq b$)
⟨*proof*⟩

lemma *convex-le-PDPlus-PDUnit-iff* [*simp*]:
(*PDPlus t u* $\leq\natural$ *PDUnit a*) = (*t* $\leq\natural$ *PDUnit a* $\wedge$ *u* $\leq\natural$ *PDUnit a*)
⟨*proof*⟩

lemma *convex-le-PDPlus-lemma*:
assumes *z*: *PDPlus t u* $\leq\natural$ *z*
shows $\exists$ *v w*. *z* = *PDPlus v w* $\wedge$ *t* $\leq\natural$ *v* $\wedge$ *u* $\leq\natural$ *w*
⟨*proof*⟩

lemma *convex-le-induct* [*induct set*: *convex-le*]:
assumes *le*: *t* $\leq\natural$ *u*
assumes *2*: $\bigwedge$*t u v*. $[\![P\ t\ u;\ P\ u\ v]\!] \Longrightarrow P\ t\ v$
assumes *3*: $\bigwedge$*a b*. $a \sqsubseteq b \Longrightarrow P$ (*PDUnit a*) (*PDUnit b*)
assumes *4*: $\bigwedge$*t u v w*. $[\![P\ t\ v;\ P\ u\ w]\!] \Longrightarrow P$ (*PDPlus t u*) (*PDPlus v w*)
shows *P t u*
⟨*proof*⟩

lemma *pd-take-convex-chain*:
pd-take n t $\leq\natural$ *pd-take* (*Suc n*) *t*

⟨*proof*⟩

lemma *pd-take-convex-le*: *pd-take i t* $\leq\natural$ *t*
⟨*proof*⟩

lemma *pd-take-convex-mono*:
t $\leq\natural$ *u* $\Longrightarrow$ *pd-take n t* $\leq\natural$ *pd-take n u*
⟨*proof*⟩

27.2 Type definition

typedef (**open**) *'a convex-pd* =
{*S*::*'a pd-basis set*. *convex-le.ideal S*}
⟨*proof*⟩

instantiation *convex-pd* :: (*profinite*) *sq-ord*
begin

definition
x $\sqsubseteq$ *y* $\longleftrightarrow$ *Rep-convex-pd x* $\subseteq$ *Rep-convex-pd y*

instance ⟨*proof*⟩
end

instance *convex-pd* :: (*profinite*) *po*
⟨*proof*⟩

instance *convex-pd* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-convex-pd-lub*:
chain Y $\Longrightarrow$ *Rep-convex-pd* ($\bigsqcup$ *i*. *Y i*) = ($\bigcup$ *i*. *Rep-convex-pd* (*Y i*))
⟨*proof*⟩

lemma *ideal-Rep-convex-pd*: *convex-le.ideal* (*Rep-convex-pd xs*)
⟨*proof*⟩

definition
convex-principal :: *'a pd-basis* ⇒ *'a convex-pd* **where**
convex-principal t = *Abs-convex-pd* {*u*. *u* $\leq\natural$ *t*}

lemma *Rep-convex-principal*:
Rep-convex-pd (*convex-principal t*) = {*u*. *u* $\leq\natural$ *t*}
⟨*proof*⟩

interpretation *convex-pd*:
ideal-completion convex-le pd-take convex-principal Rep-convex-pd
⟨*proof*⟩

Convex powerdomain is pointed

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) ⊑ *ys*
⟨*proof*⟩

instance *convex-pd* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-convex-pd-pcpo*: ⊥ = *convex-principal* (*PDUnit compact-bot*)
⟨*proof*⟩

Convex powerdomain is profinite

instantiation *convex-pd* :: (*profinite*) *profinite*
begin

definition
approx-convex-pd-def: *approx* = *convex-pd.completion-approx*

instance
⟨*proof*⟩

end

instance *convex-pd* :: (*bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-convex-principal* [*simp*]:
approx n·(*convex-principal t*) = *convex-principal* (*pd-take n t*)
⟨*proof*⟩

lemma *approx-eq-convex-principal*:
∃ *t*∈*Rep-convex-pd xs*. *approx n*·*xs* = *convex-principal* (*pd-take n t*)
⟨*proof*⟩

27.3 Monadic unit and plus

definition
convex-unit :: *'a* → *'a convex-pd* **where**
convex-unit = *compact-basis.basis-fun* (λ*a*. *convex-principal* (*PDUnit a*))

definition
convex-plus :: *'a convex-pd* → *'a convex-pd* → *'a convex-pd* **where**
convex-plus = *convex-pd.basis-fun* (λ*t*. *convex-pd.basis-fun* (λ*u*.
convex-principal (*PDPlus t u*)))

abbreviation
convex-add :: *'a convex-pd* ⇒ *'a convex-pd* ⇒ *'a convex-pd*
(**infixl** +♮ *65*) **where**
xs +♮ *ys* == *convex-plus*·*xs*·*ys*

syntax
-*convex-pd* :: *args* ⇒ *'a convex-pd* ({-}♮)

translations
$\{x,xs\}\natural == \{x\}\natural +\natural \{xs\}\natural$
$\{x\}\natural == CONST\ convex\text{-}unit{\cdot}x$

lemma *convex-unit-Rep-compact-basis* [*simp*]:
$\{Rep\text{-}compact\text{-}basis\ a\}\natural = convex\text{-}principal\ (PDUnit\ a)$
⟨*proof*⟩

lemma *convex-plus-principal* [*simp*]:
$convex\text{-}principal\ t +\natural convex\text{-}principal\ u = convex\text{-}principal\ (PDPlus\ t\ u)$
⟨*proof*⟩

lemma *approx-convex-unit* [*simp*]:
$approx\ n{\cdot}\{x\}\natural = \{approx\ n{\cdot}x\}\natural$
⟨*proof*⟩

lemma *approx-convex-plus* [*simp*]:
$approx\ n{\cdot}(xs +\natural ys) = approx\ n{\cdot}xs +\natural approx\ n{\cdot}ys$
⟨*proof*⟩

lemma *convex-plus-assoc*:
$(xs +\natural ys) +\natural zs = xs +\natural (ys +\natural zs)$
⟨*proof*⟩

lemma *convex-plus-commute*: $xs +\natural ys = ys +\natural xs$
⟨*proof*⟩

lemma *convex-plus-absorb* [*simp*]: $xs +\natural xs = xs$
⟨*proof*⟩

lemma *convex-plus-left-commute*: $xs +\natural (ys +\natural zs) = ys +\natural (xs +\natural zs)$
⟨*proof*⟩

lemma *convex-plus-left-absorb* [*simp*]: $xs +\natural (xs +\natural ys) = xs +\natural ys$
⟨*proof*⟩

Useful for *simp add*: *convex-plus-ac*

lemmas *convex-plus-ac* =
convex-plus-assoc convex-plus-commute convex-plus-left-commute

Useful for *simp only*: *convex-plus-aci*

lemmas *convex-plus-aci* =
convex-plus-ac convex-plus-absorb convex-plus-left-absorb

lemma *convex-unit-less-plus-iff* [*simp*]:
$\{x\}\natural \sqsubseteq ys +\natural zs \longleftrightarrow \{x\}\natural \sqsubseteq ys \wedge \{x\}\natural \sqsubseteq zs$
⟨*proof*⟩

lemma *convex-plus-less-unit-iff* [*simp*]:
$xs +\natural ys \sqsubseteq \{z\}\natural \longleftrightarrow xs \sqsubseteq \{z\}\natural \wedge ys \sqsubseteq \{z\}\natural$
⟨*proof*⟩

lemma *convex-unit-less-iff* [*simp*]: $\{x\}\natural \sqsubseteq \{y\}\natural \longleftrightarrow x \sqsubseteq y$
⟨*proof*⟩

lemma *convex-unit-eq-iff* [*simp*]: $\{x\}\natural = \{y\}\natural \longleftrightarrow x = y$
⟨*proof*⟩

lemma *convex-unit-strict* [*simp*]: $\{\bot\}\natural = \bot$
⟨*proof*⟩

lemma *convex-unit-strict-iff* [*simp*]: $\{x\}\natural = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *compact-convex-unit-iff* [*simp*]:
$compact\ \{x\}\natural \longleftrightarrow compact\ x$
⟨*proof*⟩

lemma *compact-convex-plus* [*simp*]:
$[\![compact\ xs;\ compact\ ys]\!] \Longrightarrow compact\ (xs +\natural ys)$
⟨*proof*⟩

27.4 Induction rules

lemma *convex-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\natural;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\natural +\natural ys)$
shows *P* (*xs*::'*a convex-pd*)
⟨*proof*⟩

lemma *convex-pd-induct*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\natural ys)$
shows *P* (*xs*::'*a convex-pd*)
⟨*proof*⟩

27.5 Monadic bind

definition
convex-bind-basis ::
'*a pd-basis* ⇒ ('*a* → '*b convex-pd*) → '*b convex-pd* **where**
convex-bind-basis = *fold-pd*
$(\lambda a.\ \Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\natural y{\cdot}f)$

lemma *ACI-convex-bind*:

ab-semigroup-idem-mult $(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f\ +\natural\ y{\cdot}f)$
⟨*proof*⟩

lemma *convex-bind-basis-simps* [*simp*]:
convex-bind-basis $(PDUnit\ a) =$
$(\Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
convex-bind-basis $(PDPlus\ t\ u) =$
$(\Lambda\ f.\ convex\text{-}bind\text{-}basis\ t{\cdot}f\ +\natural\ convex\text{-}bind\text{-}basis\ u{\cdot}f)$
⟨*proof*⟩

lemma *monofun-LAM*:
$[\![cont\ f;\ cont\ g;\ \bigwedge x.\ f\ x \sqsubseteq g\ x]\!] \Longrightarrow (\Lambda\ x.\ f\ x) \sqsubseteq (\Lambda\ x.\ g\ x)$
⟨*proof*⟩

lemma *convex-bind-basis-mono*:
$t \leq\natural\ u \Longrightarrow convex\text{-}bind\text{-}basis\ t \sqsubseteq convex\text{-}bind\text{-}basis\ u$
⟨*proof*⟩

definition
convex-bind :: $'a$ *convex-pd* $\rightarrow$ ($'a \rightarrow 'b$ *convex-pd*) $\rightarrow$ $'b$ *convex-pd* **where**
convex-bind = *convex-pd.basis-fun convex-bind-basis*

lemma *convex-bind-principal* [*simp*]:
convex-bind·(*convex-principal t*) = *convex-bind-basis t*
⟨*proof*⟩

lemma *convex-bind-unit* [*simp*]:
convex-bind·{*x*}♮·*f* = *f*·*x*
⟨*proof*⟩

lemma *convex-bind-plus* [*simp*]:
convex-bind·(*xs* +♮ *ys*)·*f* = *convex-bind*·*xs*·*f* +♮ *convex-bind*·*ys*·*f*
⟨*proof*⟩

lemma *convex-bind-strict* [*simp*]: *convex-bind*·⊥·*f* = *f*·⊥
⟨*proof*⟩

27.6 Map and join

definition
convex-map :: ($'a \rightarrow 'b$) $\rightarrow$ $'a$ *convex-pd* $\rightarrow$ $'b$ *convex-pd* **where**
convex-map = (Λ *f xs*. *convex-bind*·*xs*·(Λ *x*. {*f*·*x*}♮))

definition
convex-join :: $'a$ *convex-pd convex-pd* $\rightarrow$ $'a$ *convex-pd* **where**
convex-join = (Λ *xss*. *convex-bind*·*xss*·(Λ *xs*. *xs*))

lemma *convex-map-unit* [*simp*]:
convex-map·*f*·(*convex-unit*·*x*) = *convex-unit*·(*f*·*x*)

⟨*proof*⟩

lemma *convex-map-plus* [*simp*]:
convex-map·*f*·(*xs* +♮ *ys*) = *convex-map*·*f*·*xs* +♮ *convex-map*·*f*·*ys*
⟨*proof*⟩

lemma *convex-join-unit* [*simp*]:
convex-join·{*xs*}♮ = *xs*
⟨*proof*⟩

lemma *convex-join-plus* [*simp*]:
convex-join·(*xss* +♮ *yss*) = *convex-join*·*xss* +♮ *convex-join*·*yss*
⟨*proof*⟩

lemma *convex-map-ident*: *convex-map*·(Λ *x*. *x*)·*xs* = *xs*
⟨*proof*⟩

lemma *convex-map-map*:
convex-map·*f*·(*convex-map*·*g*·*xs*) = *convex-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
⟨*proof*⟩

lemma *convex-join-map-unit*:
convex-join·(*convex-map*·*convex-unit*·*xs*) = *xs*
⟨*proof*⟩

lemma *convex-join-map-join*:
convex-join·(*convex-map*·*convex-join*·*xsss*) = *convex-join*·(*convex-join*·*xsss*)
⟨*proof*⟩

lemma *convex-join-map-map*:
convex-join·(*convex-map*·(*convex-map*·*f*)·*xss*) =
convex-map·*f*·(*convex-join*·*xss*)
⟨*proof*⟩

lemma *convex-map-approx*: *convex-map*·(*approx n*)·*xs* = *approx n*·*xs*
⟨*proof*⟩

27.7 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: *t* ≤♮ *u* ⟹ *t* ≤♯ *u*
⟨*proof*⟩

definition
convex-to-upper :: *'a convex-pd* → *'a upper-pd* **where**
convex-to-upper = *convex-pd.basis-fun upper-principal*

lemma *convex-to-upper-principal* [*simp*]:
convex-to-upper·(*convex-principal t*) = *upper-principal t*

⟨*proof*⟩

lemma *convex-to-upper-unit* [*simp*]:
convex-to-upper·{*x*}♮ = {*x*}♯
⟨*proof*⟩

lemma *convex-to-upper-plus* [*simp*]:
convex-to-upper·(*xs* +♮ *ys*) = *convex-to-upper*·*xs* +♯ *convex-to-upper*·*ys*
⟨*proof*⟩

lemma *approx-convex-to-upper*:
approx i·(*convex-to-upper*·*xs*) = *convex-to-upper*·(*approx i*·*xs*)
⟨*proof*⟩

lemma *convex-to-upper-bind* [*simp*]:
convex-to-upper·(*convex-bind*·*xs*·*f*) =
upper-bind·(*convex-to-upper*·*xs*)·(*convex-to-upper oo f*)
⟨*proof*⟩

lemma *convex-to-upper-map* [*simp*]:
convex-to-upper·(*convex-map*·*f*·*xs*) = *upper-map*·*f*·(*convex-to-upper*·*xs*)
⟨*proof*⟩

lemma *convex-to-upper-join* [*simp*]:
convex-to-upper·(*convex-join*·*xss*) =
upper-bind·(*convex-to-upper*·*xss*)·*convex-to-upper*
⟨*proof*⟩

Convex to lower

lemma *convex-le-imp-lower-le*: *t* ≤♮ *u* ⟹ *t* ≤♭ *u*
⟨*proof*⟩

definition
convex-to-lower :: *'a convex-pd* → *'a lower-pd* **where**
convex-to-lower = *convex-pd.basis-fun lower-principal*

lemma *convex-to-lower-principal* [*simp*]:
convex-to-lower·(*convex-principal t*) = *lower-principal t*
⟨*proof*⟩

lemma *convex-to-lower-unit* [*simp*]:
convex-to-lower·{*x*}♮ = {*x*}♭
⟨*proof*⟩

lemma *convex-to-lower-plus* [*simp*]:
convex-to-lower·(*xs* +♮ *ys*) = *convex-to-lower*·*xs* +♭ *convex-to-lower*·*ys*
⟨*proof*⟩

lemma *approx-convex-to-lower*:

approx i·(convex-to-lower·xs) = convex-to-lower·(approx i·xs)
⟨*proof*⟩

lemma *convex-to-lower-bind* [*simp*]:
convex-to-lower·(convex-bind·xs·f) =
lower-bind·(convex-to-lower·xs)·(convex-to-lower oo f)
⟨*proof*⟩

lemma *convex-to-lower-map* [*simp*]:
convex-to-lower·(convex-map·f·xs) = lower-map·f·(convex-to-lower·xs)
⟨*proof*⟩

lemma *convex-to-lower-join* [*simp*]:
convex-to-lower·(convex-join·xss) =
lower-bind·(convex-to-lower·xss)·convex-to-lower
⟨*proof*⟩

Ordering property

lemma *convex-pd-less-iff*:
(xs ⊑ ys) =
(convex-to-upper·xs ⊑ convex-to-upper·ys ∧
convex-to-lower·xs ⊑ convex-to-lower·ys)
⟨*proof*⟩

lemmas *convex-plus-less-plus-iff =*
convex-pd-less-iff [**where** *xs=xs +♮ ys* **and** *ys=zs +♮ ws*, *standard*]

lemmas *convex-pd-less-simps =*
convex-unit-less-plus-iff
convex-plus-less-unit-iff
convex-plus-less-plus-iff
convex-unit-less-iff
convex-to-upper-unit
convex-to-upper-plus
convex-to-lower-unit
convex-to-lower-plus
upper-pd-less-simps
lower-pd-less-simps

end

28 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

28.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
infinite :: *'a set* ⇒ *bool* **where**
infinite S == ¬ *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* ==> $S \neq \{\}$
⟨*proof*⟩

lemma *infinite-remove*:
infinite S ⟹ *infinite* $(S - \{a\})$
⟨*proof*⟩

lemma *Diff-infinite-finite*:
assumes *T*: *finite T* **and** *S*: *infinite S*
shows *infinite* $(S - T)$
⟨*proof*⟩

lemma *Un-infinite*: *infinite S* ⟹ *infinite* $(S \cup T)$
⟨*proof*⟩

lemma *infinite-super*:
assumes *T*: $S \subseteq T$ **and** *S*: *infinite S*
shows *infinite T*
⟨*proof*⟩

As a concrete example, we prove that the set of natural numbers is infinite.

lemma *finite-nat-bounded*:
assumes *S*: *finite* (*S*::*nat set*)
shows $\exists k.\ S \subseteq \{..<k\}$ (**is** $\exists k.$ *?bounded S k*)
⟨*proof*⟩

lemma *finite-nat-iff-bounded*:
finite (*S*::*nat set*) = $(\exists k.\ S \subseteq \{..<k\})$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*:
finite (*S*::*nat set*) = $(\exists k.\ S \subseteq \{..k\})$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*:
infinite (*S*::*nat set*) = $(\forall m.\ \exists n.\ m<n \wedge n \in S)$
(**is** *?lhs* = *?rhs*)

⟨*proof*⟩

lemma *infinite-nat-iff-unbounded-le*:
infinite (*S*::*nat set*) = (∀ *m*. ∃ *n*. *m*≤*n* ∧ *n*∈*S*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*:
assumes *k*: ∀ *m*. *k*<*m* ⟶ (∃ *n*. *m*<*n* ∧ *n*∈*S*)
shows *infinite* (*S*::*nat set*)
⟨*proof*⟩

lemma *nat-infinite* [*simp*]: *infinite* (*UNIV* :: *nat set*)
⟨*proof*⟩

lemma *nat-not-finite* [*elim*]: *finite* (*UNIV*::*nat set*) ⟹ *R*
⟨*proof*⟩

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

lemma *range-inj-infinite*:
inj (*f*::*nat* ⇒ *'a*) ⟹ *infinite* (*range f*)
⟨*proof*⟩

lemma *int-infinite* [*simp*]:
shows *infinite* (*UNIV*::*int set*)
⟨*proof*⟩

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *linorder-injI*:
assumes *hyp*: !!*x y*. *x* < (*y*::*'a*::*linorder*) ==> *f x* ≠ *f y*
shows *inj f*
⟨*proof*⟩

lemma *infinite-countable-subset*:
assumes *inf*: *infinite* (*S*::*'a set*)
shows ∃ *f*. *inj* (*f*::*nat* ⇒ *'a*) ∧ *range f* ⊆ *S*
⟨*proof*⟩

lemma *infinite-iff-countable-subset*:
infinite S = (∃ *f*. *inj* (*f*::*nat* ⇒ *'a*) ∧ *range f* ⊆ *S*)

⟨*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 ∃ *y* ∈ *f*‘*A*. *infinite* (*f* −‘ {*y*})
⟨*proof*⟩

lemma *inf-img-fin-domE*:
assumes *finite* (*f*‘*A*) **and** *infinite A*
obtains *y* **where** *y* ∈ *f*‘*A* **and** *infinite* (*f* −‘ {*y*})
⟨*proof*⟩

28.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

definition
Inf-many :: (′*a* ⇒ *bool*) ⇒ *bool* (**binder** *INFM* 10) **where**
Inf-many P = *infinite* {*x*. *P x*}

definition
Alm-all :: (′*a* ⇒ *bool*) ⇒ *bool* (**binder** *MOST* 10) **where**
Alm-all P = (¬ (*INFM x*. ¬ *P x*))

notation (*xsymbols*)
Inf-many (**binder** ∃∞ 10) **and**
Alm-all (**binder** ∀∞ 10)

notation (*HTML* **output**)
Inf-many (**binder** ∃∞ 10) **and**
Alm-all (**binder** ∀∞ 10)

lemma *INFM-EX*:
(∃∞*x*. *P x*) ⟹ (∃ *x*. *P x*)
⟨*proof*⟩

lemma *MOST-iff-finiteNeg*: (∀∞*x*. *P x*) = *finite* {*x*. ¬ *P x*}
⟨*proof*⟩

lemma *ALL-MOST*: ∀ *x*. *P x* ⟹ ∀∞*x*. *P x*
⟨*proof*⟩

lemma *INFM-mono*:

assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
⟨*proof*⟩

lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$
⟨*proof*⟩

lemma *INFM-disj-distrib*:
$(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$
⟨*proof*⟩

lemma *MOST-conj-distrib*:
$(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$
⟨*proof*⟩

lemma *MOST-rev-mp*:
assumes $\forall_\infty x.\ P\ x$ **and** $\forall_\infty x.\ P\ x \longrightarrow Q\ x$
shows $\forall_\infty x.\ Q\ x$
⟨*proof*⟩

lemma *not-INFM* [*simp*]: $\neg\ (INFM\ x.\ P\ x) \longleftrightarrow (MOST\ x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *not-MOST* [*simp*]: $\neg\ (MOST\ x.\ P\ x) \longleftrightarrow (INFM\ x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *INFM-const* [*simp*]: $(INFM\ x{::}'a.\ P) \longleftrightarrow P \wedge infinite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-const* [*simp*]: $(MOST\ x{::}'a.\ P) \longleftrightarrow P \vee finite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *INFM-nat*: $(\exists_\infty n.\ P\ (n{::}nat)) = (\forall\, m.\ \exists\, n.\ m{<}n \wedge P\ n)$
⟨*proof*⟩

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ (n{::}nat)) = (\forall\, m.\ \exists\, n.\ m{\leq}n \wedge P\ n)$
⟨*proof*⟩

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n{::}nat)) = (\exists\, m.\ \forall\, n.\ m{<}n \longrightarrow P\ n)$
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n{::}nat)) = (\exists\, m.\ \forall\, n.\ m{\leq}n \longrightarrow P\ n)$
⟨*proof*⟩

28.3 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

consts
enumerate :: $'a{::}wellorder\ set \Rightarrow (nat \Rightarrow 'a{::}wellorder)$

primrec
enumerate-0: *enumerate S 0* = (*LEAST n. n ∈ S*)
enumerate-Suc: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*LEAST n. n ∈ S*}) *n*

lemma *enumerate-Suc′*:
enumerate S (*Suc n*) = *enumerate* (*S* − {*enumerate S 0*}) *n*
⟨*proof*⟩

lemma *enumerate-in-set*: *infinite S* ⟹ *enumerate S n* : *S*
⟨*proof*⟩

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite S* ⟹ *enumerate S n* < *enumerate S* (*Suc n*)
⟨*proof*⟩

lemma *enumerate-mono*: *m*<*n* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate S n*
⟨*proof*⟩

28.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one :: *′a set* ⇒ *bool* **where**
atmost-one S = (∀ *x y. x*∈*S* ∧ *y*∈*S* ⟶ *x*=*y*)

lemma *atmost-one-empty*: *S* = {} ⟹ *atmost-one S*
⟨*proof*⟩

lemma *atmost-one-singleton*: *S* = {*x*} ⟹ *atmost-one S*
⟨*proof*⟩

lemma *atmost-one-unique* [*elim*]: *atmost-one S* ⟹ *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *y* = *x*
⟨*proof*⟩

end

theory *Eventual*
imports *Infinite-Set*
begin

28.5 Lemmas about MOST

lemma *MOST-INFM*:
assumes *inf*: *infinite* (*UNIV*::*′a set*)
shows *MOST x*::*′a. P x* ⟹ *INFM x*::*′a. P x*
⟨*proof*⟩

lemma *MOST-comp*: $[\![inj\ f;\ MOST\ x.\ P\ x]\!] \Longrightarrow MOST\ x.\ P\ (f\ x)$
⟨*proof*⟩

lemma *INFM-comp*: $[\![inj\ f;\ INFM\ x.\ P\ (f\ x)]\!] \Longrightarrow INFM\ x.\ P\ x$
⟨*proof*⟩

lemma *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
⟨*proof*⟩

lemma *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
⟨*proof*⟩

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
⟨*proof*⟩

lemma *INFM-finite-Bex-distrib*:
$finite\ A \Longrightarrow (INFM\ y.\ \exists\, x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists\, x{\in}A.\ INFM\ y.\ P\ x\ y)$
⟨*proof*⟩

lemma *MOST-finite-Ball-distrib*:
$finite\ A \Longrightarrow (MOST\ y.\ \forall\, x{\in}A.\ P\ x\ y) \longleftrightarrow (\forall\, x{\in}A.\ MOST\ y.\ P\ x\ y)$
⟨*proof*⟩

lemma *MOST-ge-nat*: $MOST\ n{::}nat.\ m \leq n$
⟨*proof*⟩

28.6 Eventually constant sequences

definition
$eventually\text{-}constant :: (nat \Rightarrow 'a) \Rightarrow bool$
where
$eventually\text{-}constant\ S = (\exists\, x.\ MOST\ i.\ S\ i = x)$

lemma *eventually-constant-MOST-MOST*:
$eventually\text{-}constant\ S \longleftrightarrow (MOST\ m.\ MOST\ n.\ S\ n = S\ m)$
⟨*proof*⟩

lemma *eventually-constantI*: $MOST\ i.\ S\ i = x \Longrightarrow eventually\text{-}constant\ S$
⟨*proof*⟩

lemma *eventually-constant-comp*:
$eventually\text{-}constant\ (\lambda i.\ S\ i) \Longrightarrow eventually\text{-}constant\ (\lambda i.\ f\ (S\ i))$
⟨*proof*⟩

lemma *eventually-constant-Suc-iff*:
$eventually\text{-}constant\ (\lambda i.\ S\ (Suc\ i)) \longleftrightarrow eventually\text{-}constant\ (\lambda i.\ S\ i)$
⟨*proof*⟩

lemma *eventually-constant-SucD*:
eventually-constant $(\lambda i.\ S\ (Suc\ i)) \Longrightarrow$ *eventually-constant* $(\lambda i.\ S\ i)$
⟨*proof*⟩

28.7 Limits of eventually constant sequences

definition
eventual :: $(nat \Rightarrow 'a) \Rightarrow 'a$ **where**
eventual $S = (THE\ x.\ MOST\ i.\ S\ i = x)$

lemma *eventual-eqI*: $MOST\ i.\ S\ i = x \Longrightarrow$ *eventual* $S = x$
⟨*proof*⟩

lemma *MOST-eq-eventual*:
eventually-constant $S \Longrightarrow MOST\ i.\ S\ i =$ *eventual* S
⟨*proof*⟩

lemma *eventual-mem-range*:
eventually-constant $S \Longrightarrow$ *eventual* $S \in$ *range* S
⟨*proof*⟩

lemma *eventually-constant-MOST-iff*:
assumes S: *eventually-constant* S
shows $(MOST\ n.\ P\ (S\ n)) \longleftrightarrow P\ (eventual\ S)$
⟨*proof*⟩

lemma *MOST-eventual*:
$[\![$*eventually-constant* S; $MOST\ n.\ P\ (S\ n)]\!] \Longrightarrow P\ (eventual\ S)$
⟨*proof*⟩

lemma *eventually-constant-MOST-Suc-eq*:
eventually-constant $S \Longrightarrow MOST\ n.\ S\ (Suc\ n) = S\ n$
⟨*proof*⟩

lemma *eventual-comp*:
eventually-constant $S \Longrightarrow$ *eventual* $(\lambda i.\ f\ (S\ i)) = f\ (eventual\ (\lambda i.\ S\ i))$
⟨*proof*⟩

end

29 Algebraic: Algebraic deflations

theory *Algebraic*
imports *Completion Fix Eventual*
begin

29.1 Constructing finite deflations by iteration

lemma *finite-deflation-imp-deflation*:
finite-deflation d $\Longrightarrow$ *deflation d*
⟨*proof*⟩

lemma *le-Suc-induct*:
assumes *le*: $i \leq j$
assumes *step*: $\bigwedge i.\ P\ i\ (Suc\ i)$
assumes *refl*: $\bigwedge i.\ P\ i\ i$
assumes *trans*: $\bigwedge i\ j\ k.\ [\![P\ i\ j;\ P\ j\ k]\!] \Longrightarrow P\ i\ k$
shows $P\ i\ j$
⟨*proof*⟩

A pre-deflation is like a deflation, but not idempotent.

locale *pre-deflation* =
fixes $f :: 'a \to 'a$::*cpo*
assumes *less*: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
assumes *finite-range*: *finite* (*range* ($\lambda x.\ f{\cdot}x$))
begin

lemma *iterate-less*: *iterate* $i{\cdot}f{\cdot}x \sqsubseteq x$
⟨*proof*⟩

lemma *iterate-fixed*: $f{\cdot}x = x \Longrightarrow$ *iterate* $i{\cdot}f{\cdot}x = x$
⟨*proof*⟩

lemma *antichain-iterate-app*: $i \leq j \Longrightarrow$ *iterate* $j{\cdot}f{\cdot}x \sqsubseteq$ *iterate* $i{\cdot}f{\cdot}x$
⟨*proof*⟩

lemma *finite-range-iterate-app*: *finite* (*range* ($\lambda i.$ *iterate* $i{\cdot}f{\cdot}x$))
⟨*proof*⟩

lemma *eventually-constant-iterate-app*:
eventually-constant ($\lambda i.$ *iterate* $i{\cdot}f{\cdot}x$)
⟨*proof*⟩

lemma *eventually-constant-iterate*:
eventually-constant ($\lambda n.$ *iterate* $n{\cdot}f$)
⟨*proof*⟩

abbreviation
$d :: 'a \to 'a$
where
$d \equiv$ *eventual* ($\lambda n.$ *iterate* $n{\cdot}f$)

lemma *MOST-d*: *MOST n. P* (*iterate* $n{\cdot}f$) $\Longrightarrow P\ d$
⟨*proof*⟩

lemma *f-d*: $f{\cdot}(d{\cdot}x) = d{\cdot}x$

⟨*proof*⟩

lemma *d-fixed-iff*: $d{\cdot}x = x \longleftrightarrow f{\cdot}x = x$
⟨*proof*⟩

lemma *finite-deflation-d*: *finite-deflation d*
⟨*proof*⟩

end

lemma *pre-deflation-d-f*:
assumes *finite-deflation d*
assumes f: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
shows *pre-deflation* (*d oo f*)
⟨*proof*⟩

lemma *eventual-iterate-oo-fixed-iff*:
assumes *finite-deflation d*
assumes f: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
shows *eventual* $(\lambda n.$ *iterate* $n{\cdot}(d$ *oo* $f)){\cdot}x = x \longleftrightarrow d{\cdot}x = x \wedge f{\cdot}x = x$
⟨*proof*⟩

29.2 Type constructor for finite deflations

defaultsort *profinite*

typedef (**open**) *'a fin-defl* = $\{d{::}'a \to 'a.$ *finite-deflation d*$\}$
⟨*proof*⟩

instantiation *fin-defl* :: (*profinite*) *sq-ord*
begin

definition
sq-le-fin-defl-def:
op $\sqsubseteq \equiv \lambda x\ y.$ *Rep-fin-defl x* $\sqsubseteq$ *Rep-fin-defl y*

instance ⟨*proof*⟩
end

instance *fin-defl* :: (*profinite*) *po*
⟨*proof*⟩

lemma *finite-deflation-Rep-fin-defl*: *finite-deflation* (*Rep-fin-defl d*)
⟨*proof*⟩

interpretation *Rep-fin-defl*: *finite-deflation Rep-fin-defl d*
⟨*proof*⟩

lemma *fin-defl-lessI*:

$(\bigwedge x.\ \mathit{Rep\text{-}fin\text{-}defl}\ a{\cdot}x = x \Longrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ b{\cdot}x = x) \Longrightarrow a \sqsubseteq b$
⟨*proof*⟩

lemma *fin-defl-lessD*:
$[\![a \sqsubseteq b;\ \mathit{Rep\text{-}fin\text{-}defl}\ a{\cdot}x = x]\!] \Longrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ b{\cdot}x = x$
⟨*proof*⟩

lemma *fin-defl-eqI*:
$(\bigwedge x.\ \mathit{Rep\text{-}fin\text{-}defl}\ a{\cdot}x = x \longleftrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ b{\cdot}x = x) \Longrightarrow a = b$
⟨*proof*⟩

lemma *Abs-fin-defl-mono*:
$[\![\mathit{finite\text{-}deflation}\ a;\ \mathit{finite\text{-}deflation}\ b;\ a \sqsubseteq b]\!]$
$\Longrightarrow \mathit{Abs\text{-}fin\text{-}defl}\ a \sqsubseteq \mathit{Abs\text{-}fin\text{-}defl}\ b$
⟨*proof*⟩

29.3 Take function for finite deflations

definition
$\mathit{fd\text{-}take} :: \mathit{nat} \Rightarrow 'a\ \mathit{fin\text{-}defl} \Rightarrow 'a\ \mathit{fin\text{-}defl}$
where
$\mathit{fd\text{-}take}\ i\ d = \mathit{Abs\text{-}fin\text{-}defl}\ (\mathit{eventual}\ (\lambda n.\ \mathit{iterate}\ n{\cdot}(\mathit{approx}\ i\ \mathit{oo}\ \mathit{Rep\text{-}fin\text{-}defl}\ d)))$

lemma *Rep-fin-defl-fd-take*:
$\mathit{Rep\text{-}fin\text{-}defl}\ (\mathit{fd\text{-}take}\ i\ d) =$
$\mathit{eventual}\ (\lambda n.\ \mathit{iterate}\ n{\cdot}(\mathit{approx}\ i\ \mathit{oo}\ \mathit{Rep\text{-}fin\text{-}defl}\ d))$
⟨*proof*⟩

lemma *fd-take-fixed-iff*:
$\mathit{Rep\text{-}fin\text{-}defl}\ (\mathit{fd\text{-}take}\ i\ d){\cdot}x = x \longleftrightarrow$
$\mathit{approx}\ i{\cdot}x = x \wedge \mathit{Rep\text{-}fin\text{-}defl}\ d{\cdot}x = x$
⟨*proof*⟩

lemma *fd-take-less*: $\mathit{fd\text{-}take}\ n\ d \sqsubseteq d$
⟨*proof*⟩

lemma *fd-take-idem*: $\mathit{fd\text{-}take}\ n\ (\mathit{fd\text{-}take}\ n\ d) = \mathit{fd\text{-}take}\ n\ d$
⟨*proof*⟩

lemma *fd-take-mono*: $a \sqsubseteq b \Longrightarrow \mathit{fd\text{-}take}\ n\ a \sqsubseteq \mathit{fd\text{-}take}\ n\ b$
⟨*proof*⟩

lemma *approx-fixed-le-lemma*: $[\![i \leq j;\ \mathit{approx}\ i{\cdot}x = x]\!] \Longrightarrow \mathit{approx}\ j{\cdot}x = x$
⟨*proof*⟩

lemma *fd-take-chain*: $m \leq n \Longrightarrow \mathit{fd\text{-}take}\ m\ a \sqsubseteq \mathit{fd\text{-}take}\ n\ a$
⟨*proof*⟩

lemma *finite-range-fd-take*: $\mathit{finite}\ (\mathit{range}\ (\mathit{fd\text{-}take}\ n))$

⟨*proof*⟩

lemma *fd-take-covers*: $\exists\, n.\ \textit{fd-take}\ n\ a = a$
⟨*proof*⟩

interpretation *fin-defl*: *basis-take sq-le fd-take*
⟨*proof*⟩

29.4 Defining algebraic deflations by ideal completion

typedef (**open**) *'a alg-defl* =
$\{S :: \textit{'a fin-defl set}.\ \textit{sq-le.ideal}\ S\}$
⟨*proof*⟩

instantiation *alg-defl* :: (*profinite*) *sq-ord*
begin

definition
$x \sqsubseteq y \longleftrightarrow \textit{Rep-alg-defl}\ x \subseteq \textit{Rep-alg-defl}\ y$

instance ⟨*proof*⟩
end

instance *alg-defl* :: (*profinite*) *po*
⟨*proof*⟩

instance *alg-defl* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-alg-defl-lub*:
$\textit{chain}\ Y \Longrightarrow \textit{Rep-alg-defl}\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ \textit{Rep-alg-defl}\ (Y\ i))$
⟨*proof*⟩

lemma *ideal-Rep-alg-defl*: *sq-le.ideal* (*Rep-alg-defl xs*)
⟨*proof*⟩

definition
alg-defl-principal :: *'a fin-defl* ⇒ *'a alg-defl* **where**
$\textit{alg-defl-principal}\ t = \textit{Abs-alg-defl}\ \{u.\ u \sqsubseteq t\}$

lemma *Rep-alg-defl-principal*:
$\textit{Rep-alg-defl}\ (\textit{alg-defl-principal}\ t) = \{u.\ u \sqsubseteq t\}$
⟨*proof*⟩

interpretation *alg-defl*:
ideal-completion sq-le fd-take alg-defl-principal Rep-alg-defl
⟨*proof*⟩

Algebraic deflations are pointed

lemma *finite-deflation-UU*: *finite-deflation* $\bot$
⟨*proof*⟩

lemma *alg-defl-minimal*:
alg-defl-principal (*Abs-fin-defl* $\bot$) $\sqsubseteq$ *x*
⟨*proof*⟩

instance *alg-defl* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-alg-defl-pcpo*: $\bot$ = *alg-defl-principal* (*Abs-fin-defl* $\bot$)
⟨*proof*⟩

Algebraic deflations are profinite

instantiation *alg-defl* :: (*profinite*) *profinite*
begin

definition
approx-alg-defl-def: *approx* = *alg-defl.completion-approx*

instance
⟨*proof*⟩

end

instance *alg-defl* :: (*bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-alg-defl-principal* [*simp*]:
approx n·(*alg-defl-principal t*) = *alg-defl-principal* (*fd-take n t*)
⟨*proof*⟩

lemma *approx-eq-alg-defl-principal*:
$\exists$ *t*∈*Rep-alg-defl xs*. *approx n*·*xs* = *alg-defl-principal* (*fd-take n t*)
⟨*proof*⟩

29.5 Applying algebraic deflations

definition
cast :: *'a alg-defl* → *'a* → *'a*
where
cast = *alg-defl.basis-fun Rep-fin-defl*

lemma *cast-alg-defl-principal*:
cast·(*alg-defl-principal a*) = *Rep-fin-defl a*
⟨*proof*⟩

lemma *deflation-cast*: *deflation* (*cast*·*d*)
⟨*proof*⟩

lemma *finite-deflation-cast*:
compact d $\Longrightarrow$ *finite-deflation* (*cast*·*d*)
⟨*proof*⟩

interpretation *cast*: *deflation cast*·*d*
⟨*proof*⟩

lemma *cast*·($\bigsqcup i.$ *alg-defl-principal* (*Abs-fin-defl* (*approx i*)))·*x* = *x*
⟨*proof*⟩

This lemma says that if we have an ep-pair from a bifinite domain into a universal domain, then e oo p is an algebraic deflation.

lemma
assumes *ep-pair e p*
constrains *e* :: 'a::*profinite* → 'b::*profinite*
shows $\exists$ *d*. *cast*·*d* = *e oo p*
⟨*proof*⟩

This lemma says that if we have an ep-pair from a cpo into a bifinite domain, and e oo p is an algebraic deflation, then the cpo is bifinite.

lemma
assumes *ep-pair e p*
constrains *e* :: 'a::*cpo* → 'b::*profinite*
assumes *d*: $\bigwedge x.$ *cast*·*d*·*x* = *e*·(*p*·*x*)
obtains *a* :: *nat* ⇒ 'a → 'a **where**
$\bigwedge i.$ *finite-deflation* (*a i*)
($\bigsqcup i.$ *a i*) = *ID*
⟨*proof*⟩

end

30 NatIso: Isomorphisms of the Natural Numbers

theory *NatIso*
imports *Parity*
begin

30.1 Isomorphism between naturals and sums of naturals

primrec
sum2nat :: *nat* + *nat* ⇒ *nat*
where
sum2nat (*Inl a*) = *a* + *a*
| *sum2nat* (*Inr b*) = *Suc* (*b* + *b*)

primrec
nat2sum :: *nat* ⇒ *nat* + *nat*

where
  nat2sum 0 = Inl 0
| *nat2sum (Suc n) = (case nat2sum n of*
    Inl a ⇒ Inr a | Inr b ⇒ Inl (Suc b))

lemma *nat2sum-sum2nat* [*simp*]: *nat2sum (sum2nat x) = x*
⟨*proof*⟩

lemma *sum2nat-nat2sum* [*simp*]: *sum2nat (nat2sum n) = n*
⟨*proof*⟩

lemma *inj-sum2nat*: *inj sum2nat*
⟨*proof*⟩

lemma *sum2nat-eq-iff* [*simp*]: *sum2nat x = sum2nat y ⟷ x = y*
⟨*proof*⟩

lemma *inj-nat2sum*: *inj nat2sum*
⟨*proof*⟩

lemma *nat2sum-eq-iff* [*simp*]: *nat2sum x = nat2sum y ⟷ x = y*
⟨*proof*⟩

declare *sum2nat.simps* [*simp del*]
declare *nat2sum.simps* [*simp del*]

30.2 Isomorphism between naturals and pairs of naturals

function
  prod2nat :: nat × nat ⇒ nat
where
  prod2nat (0, 0) = 0
| *prod2nat (0, Suc n) = Suc (prod2nat (n, 0))*
| *prod2nat (Suc m, n) = Suc (prod2nat (m, Suc n))*
⟨*proof*⟩

termination ⟨*proof*⟩

primrec
  nat2prod :: nat ⇒ nat × nat
where
  nat2prod 0 = (0, 0)
| *nat2prod (Suc x) =*
    (case nat2prod x of (m, n) ⇒
      (case n of 0 ⇒ (0, Suc m) | Suc n ⇒ (Suc m, n)))

lemma *nat2prod-prod2nat* [*simp*]: *nat2prod (prod2nat x) = x*
⟨*proof*⟩

lemma *prod2nat-nat2prod* [*simp*]: *prod2nat* (*nat2prod n*) = *n*
⟨*proof*⟩

lemma *inj-prod2nat*: *inj prod2nat*
⟨*proof*⟩

lemma *prod2nat-eq-iff* [*simp*]: *prod2nat x* = *prod2nat y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *inj-nat2prod*: *inj nat2prod*
⟨*proof*⟩

lemma *nat2prod-eq-iff* [*simp*]: *nat2prod x* = *nat2prod y* ⟷ *x* = *y*
⟨*proof*⟩

30.2.1 Ordering properties

lemma *fst-snd-le-prod2nat*: *fst x* ≤ *prod2nat x* ∧ *snd x* ≤ *prod2nat x*
⟨*proof*⟩

lemma *fst-le-prod2nat*: *fst x* ≤ *prod2nat x*
⟨*proof*⟩

lemma *snd-le-prod2nat*: *snd x* ≤ *prod2nat x*
⟨*proof*⟩

lemma *le-prod2nat-1*: *a* ≤ *prod2nat* (*a*, *b*)
⟨*proof*⟩

lemma *le-prod2nat-2*: *b* ≤ *prod2nat* (*a*, *b*)
⟨*proof*⟩

declare *prod2nat.simps* [*simp del*]
declare *nat2prod.simps* [*simp del*]

30.3 Isomorphism between naturals and finite sets of naturals

30.3.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite* (*Suc* −‘ *F*) ⟷ *finite F*
⟨*proof*⟩

lemma *vimage-Suc-insert-0*: *Suc* −‘ *insert 0 A* = *Suc* −‘ *A*
⟨*proof*⟩

lemma *vimage-Suc-insert-Suc*:
Suc −‘ *insert* (*Suc n*) *A* = *insert n* (*Suc* −‘ *A*)
⟨*proof*⟩

lemma *even-nat-Suc-div-2*: *even x ⟹ Suc x div 2 = x div 2*
⟨*proof*⟩

lemma *div2-even-ext-nat*:
⟦x div 2 = y div 2; even x = even y⟧ ⟹ (x::nat) = y
⟨*proof*⟩

30.3.2 From sets to naturals

definition
set2nat :: nat set ⇒ nat **where**
set2nat = setsum (op ^ 2)

lemma *set2nat-empty* [*simp*]: *set2nat {} = 0*
⟨*proof*⟩

lemma *set2nat-insert* [*simp*]:
⟦finite A; n ∉ A⟧ ⟹ set2nat (insert n A) = 2^n + set2nat A
⟨*proof*⟩

lemma *even-set2nat-iff*: *finite A ⟹ even (set2nat A) = (0 ∉ A)*
⟨*proof*⟩

lemma *set2nat-vimage-Suc*: *set2nat (Suc −‘ A) = set2nat A div 2*
⟨*proof*⟩

lemmas *set2nat-div-2 = set2nat-vimage-Suc* [*symmetric*]

30.3.3 From naturals to sets

definition
nat2set :: nat ⇒ nat set **where**
nat2set x = {n. odd (x div 2 ^ n)}

lemma *nat2set-0* [*simp*]: *0 ∈ nat2set x ⟷ odd x*
⟨*proof*⟩

lemma *nat2set-Suc* [*simp*]:
Suc n ∈ nat2set x ⟷ n ∈ nat2set (x div 2)
⟨*proof*⟩

lemma *nat2set-zero* [*simp*]: *nat2set 0 = {}*
⟨*proof*⟩

lemma *nat2set-div-2*: *nat2set (x div 2) = Suc −‘ nat2set x*
⟨*proof*⟩

lemma *nat2set-plus-power-2*:
n ∉ nat2set z ⟹ nat2set (2 ^ n + z) = insert n (nat2set z)
⟨*proof*⟩

lemma *finite-nat2set* [*simp*]: *finite* (*nat2set n*)
⟨*proof*⟩

30.3.4 Proof of isomorphism

lemma *set2nat-nat2set* [*simp*]: *set2nat* (*nat2set n*) = *n*
⟨*proof*⟩

lemma *nat2set-set2nat* [*simp*]: *finite A* ⟹ *nat2set* (*set2nat A*) = *A*
⟨*proof*⟩

lemma *inj-nat2set*: *inj nat2set*
⟨*proof*⟩

lemma *nat2set-eq-iff* [*simp*]: *nat2set x* = *nat2set y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *inj-on-set2nat*: *inj-on set2nat* (*Collect finite*)
⟨*proof*⟩

lemma *set2nat-eq-iff* [*simp*]:
⟦*finite A*; *finite B*⟧ ⟹ *set2nat A* = *set2nat B* ⟷ *A* = *B*
⟨*proof*⟩

30.3.5 Ordering properties

lemma *nat-less-power2*: *n* < *2^n*
⟨*proof*⟩

lemma *less-set2nat*: ⟦*finite A*; *x* ∈ *A*⟧ ⟹ *x* < *set2nat A*
⟨*proof*⟩

end

theory *Universal*
imports *CompactBasis NatIso*
begin

30.4 Basis datatype

types *ubasis* = *nat*

definition
node :: *nat* ⇒ *ubasis* ⇒ *ubasis set* ⇒ *ubasis*
where
node i a S = *Suc* (*prod2nat* (*i*, *prod2nat* (*a*, *set2nat S*)))

lemma *node-not-0* [*simp*]: *node i a S* ≠ *0*

⟨*proof*⟩

lemma *node-gt-0* [*simp*]: $0 < node\ i\ a\ S$
⟨*proof*⟩

lemma *node-inject* [*simp*]:
$[\![finite\ S;\ finite\ T]\!]$
$\Longrightarrow node\ i\ a\ S = node\ j\ b\ T \longleftrightarrow i = j \wedge a = b \wedge S = T$
⟨*proof*⟩

lemma *node-gt0*: $i < node\ i\ a\ S$
⟨*proof*⟩

lemma *node-gt1*: $a < node\ i\ a\ S$
⟨*proof*⟩

lemma *nat-less-power2*: $n < 2\hat{}n$
⟨*proof*⟩

lemma *node-gt2*: $[\![finite\ S;\ b \in S]\!] \Longrightarrow b < node\ i\ a\ S$
⟨*proof*⟩

lemma *eq-prod2nat-pairI*:
$[\![fst\ (nat2prod\ x) = a;\ snd\ (nat2prod\ x) = b]\!] \Longrightarrow x = prod2nat\ (a,\ b)$
⟨*proof*⟩

lemma *node-cases*:
assumes *1*: $x = 0 \Longrightarrow P$
assumes *2*: $\bigwedge i\ a\ S.\ [\![finite\ S;\ x = node\ i\ a\ S]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *node-induct*:
assumes *1*: $P\ 0$
assumes *2*: $\bigwedge i\ a\ S.\ [\![P\ a;\ finite\ S;\ \forall b{\in}S.\ P\ b]\!] \Longrightarrow P\ (node\ i\ a\ S)$
shows $P\ x$
⟨*proof*⟩

30.5 Basis ordering

inductive
ubasis-le :: $nat \Rightarrow nat \Rightarrow bool$
where
ubasis-le-refl: *ubasis-le a a*
| *ubasis-le-trans*:
$[\![ubasis\text{-}le\ a\ b;\ ubasis\text{-}le\ b\ c]\!] \Longrightarrow ubasis\text{-}le\ a\ c$
| *ubasis-le-lower*:
$finite\ S \Longrightarrow ubasis\text{-}le\ a\ (node\ i\ a\ S)$
| *ubasis-le-upper*:

$[\![\mathit{finite}\ S;\ b \in S;\ \mathit{ubasis\text{-}le}\ a\ b]\!] \Longrightarrow \mathit{ubasis\text{-}le}\ (\mathit{node}\ i\ a\ S)\ b$

lemma *ubasis-le-minimal*: *ubasis-le 0 x*
⟨*proof*⟩

30.5.1 Generic take function

function
ubasis-until :: $(\mathit{ubasis} \Rightarrow \mathit{bool}) \Rightarrow \mathit{ubasis} \Rightarrow \mathit{ubasis}$
where
ubasis-until P 0 = 0
| *finite S* $\Longrightarrow$ *ubasis-until P (node i a S) =*
(if P (node i a S) then node i a S else ubasis-until P a)
⟨*proof*⟩

termination *ubasis-until*
⟨*proof*⟩

lemma *ubasis-until*: *P 0* $\Longrightarrow$ *P (ubasis-until P x)*
⟨*proof*⟩

lemma *ubasis-until'*: *0 < ubasis-until P x* $\Longrightarrow$ *P (ubasis-until P x)*
⟨*proof*⟩

lemma *ubasis-until-same*: *P x* $\Longrightarrow$ *ubasis-until P x = x*
⟨*proof*⟩

lemma *ubasis-until-idem*:
P 0 $\Longrightarrow$ *ubasis-until P (ubasis-until P x) = ubasis-until P x*
⟨*proof*⟩

lemma *ubasis-until-0*:
$\forall x.\ x \neq 0 \longrightarrow \neg\ P\ x \Longrightarrow$ *ubasis-until P x = 0*
⟨*proof*⟩

lemma *ubasis-until-less*: *ubasis-le (ubasis-until P x) x*
⟨*proof*⟩

lemma *ubasis-until-chain*:
assumes *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows *ubasis-le (ubasis-until P x) (ubasis-until Q x)*
⟨*proof*⟩

lemma *ubasis-until-mono*:
assumes $\bigwedge i\ a\ S\ b.\ [\![\mathit{finite}\ S;\ P\ (\mathit{node}\ i\ a\ S);\ b \in S;\ \mathit{ubasis\text{-}le}\ a\ b]\!] \Longrightarrow P\ b$
shows *ubasis-le a b* $\Longrightarrow$ *ubasis-le (ubasis-until P a) (ubasis-until P b)*
⟨*proof*⟩

lemma *finite-range-ubasis-until*:

finite $\{x.\ P\ x\} \Longrightarrow$ *finite* (*range* (*ubasis-until* P))
⟨*proof*⟩

30.5.2 Take function for *ubasis*

definition
ubasis-take :: *nat* ⇒ *ubasis* ⇒ *ubasis*
where
ubasis-take n = *ubasis-until* $(\lambda x.\ x \leq n)$

lemma *ubasis-take-le*: *ubasis-take* $n\ x \leq n$
⟨*proof*⟩

lemma *ubasis-take-same*: $x \leq n \Longrightarrow$ *ubasis-take* $n\ x = x$
⟨*proof*⟩

lemma *ubasis-take-idem*: *ubasis-take* n (*ubasis-take* $n\ x$) = *ubasis-take* $n\ x$
⟨*proof*⟩

lemma *ubasis-take-0* [*simp*]: *ubasis-take* $0\ x = 0$
⟨*proof*⟩

lemma *ubasis-take-less*: *ubasis-le* (*ubasis-take* $n\ x$) x
⟨*proof*⟩

lemma *ubasis-take-chain*: *ubasis-le* (*ubasis-take* $n\ x$) (*ubasis-take* (*Suc* n) x)
⟨*proof*⟩

lemma *ubasis-take-mono*:
assumes *ubasis-le* $x\ y$
shows *ubasis-le* (*ubasis-take* $n\ x$) (*ubasis-take* $n\ y$)
⟨*proof*⟩

lemma *finite-range-ubasis-take*: *finite* (*range* (*ubasis-take* n))
⟨*proof*⟩

lemma *ubasis-take-covers*: $\exists\, n.$ *ubasis-take* $n\ x = x$
⟨*proof*⟩

interpretation *udom*: *preorder ubasis-le*
⟨*proof*⟩

interpretation *udom*: *basis-take ubasis-le ubasis-take*
⟨*proof*⟩

30.6 Defining the universal domain by ideal completion

typedef (**open**) *udom* = $\{S.$ *udom.ideal* $S\}$
⟨*proof*⟩

instantiation *udom* :: *sq-ord*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-udom* $x \subseteq$ *Rep-udom* y

instance ⟨*proof*⟩
end

instance *udom* :: *po*
⟨*proof*⟩

instance *udom* :: *cpo*
⟨*proof*⟩

lemma *Rep-udom-lub*:
chain $Y \Longrightarrow$ *Rep-udom* $(\bigsqcup i.\ Y\ i) = (\bigcup i.$ *Rep-udom* $(Y\ i))$
⟨*proof*⟩

lemma *ideal-Rep-udom*: *udom.ideal* (*Rep-udom xs*)
⟨*proof*⟩

definition
udom-principal :: *nat* ⇒ *udom* **where**
udom-principal $t =$ *Abs-udom* $\{u.$ *ubasis-le* $u\ t\}$

lemma *Rep-udom-principal*:
Rep-udom (*udom-principal* $t) = \{u.$ *ubasis-le* $u\ t\}$
⟨*proof*⟩

interpretation *udom*:
ideal-completion ubasis-le ubasis-take udom-principal Rep-udom
⟨*proof*⟩

Universal domain is pointed

lemma *udom-minimal*: *udom-principal 0* $\sqsubseteq x$
⟨*proof*⟩

instance *udom* :: *pcpo*
⟨*proof*⟩

lemma *inst-udom-pcpo*: $\bot =$ *udom-principal 0*
⟨*proof*⟩

Universal domain is bifinite

instantiation *udom* :: *bifinite*
begin

definition

approx-udom-def: *approx* = *udom.completion-approx*

instance
⟨*proof*⟩

end

lemma *approx-udom-principal* [*simp*]:
approx *n*·(*udom-principal* *x*) = *udom-principal* (*ubasis-take* *n* *x*)
⟨*proof*⟩

lemma *approx-eq-udom-principal*:
∃ *a*∈*Rep-udom* *x*. *approx* *n*·*x* = *udom-principal* (*ubasis-take* *n* *a*)
⟨*proof*⟩

30.7 Universality of *udom*

defaultsort *bifinite*

30.7.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
fixes *A* :: 'a::*po set*
shows ⟦*finite* *A*; *A* ≠ {}⟧ ⟹ ∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*
⟨*proof*⟩

definition
choose :: 'a *compact-basis set* ⇒ 'a *compact-basis*
where
choose *A* = (*SOME* *x*. *x* ∈ {*x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*})

lemma *choose-lemma*:
⟦*finite* *A*; *A* ≠ {}⟧ ⟹ *choose* *A* ∈ {*x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*}
⟨*proof*⟩

lemma *maximal-choose*:
⟦*finite* *A*; *y* ∈ *A*; *choose* *A* ⊑ *y*⟧ ⟹ *choose* *A* = *y*
⟨*proof*⟩

lemma *choose-in*: ⟦*finite* *A*; *A* ≠ {}⟧ ⟹ *choose* *A* ∈ *A*
⟨*proof*⟩

function
choose-pos :: 'a *compact-basis set* ⇒ 'a *compact-basis* ⇒ *nat*
where
choose-pos *A* *x* =
(*if* *finite* *A* ∧ *x* ∈ *A* ∧ *x* ≠ *choose* *A*
then *Suc* (*choose-pos* (*A* − {*choose* *A*}) *x*) *else* 0)
⟨*proof*⟩

termination *choose-pos*
⟨*proof*⟩

declare *choose-pos.simps* [*simp del*]

lemma *choose-pos-choose*: *finite A* ⟹ *choose-pos A* (*choose A*) = *0*
⟨*proof*⟩

lemma *inj-on-choose-pos* [*OF refl*]:
⟦*card A* = *n*; *finite A*⟧ ⟹ *inj-on* (*choose-pos A*) *A*
⟨*proof*⟩

lemma *choose-pos-bounded* [*OF refl*]:
⟦*card A* = *n*; *finite A*; *x* ∈ *A*⟧ ⟹ *choose-pos A x* < *n*
⟨*proof*⟩

lemma *choose-pos-lessD*:
⟦*choose-pos A x* < *choose-pos A y*; *finite A*; *x* ∈ *A*; *y* ∈ *A*⟧ ⟹ ¬ *x* ⊑ *y*
⟨*proof*⟩

30.7.2 Rank of basis elements

primrec
cb-take :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis*
where
cb-take 0 = (λ*x*. *compact-bot*)
| *cb-take* (*Suc n*) = *compact-take n*

lemma *cb-take-covers*: ∃ *n*. *cb-take n x* = *x*
⟨*proof*⟩

lemma *cb-take-less*: *cb-take n x* ⊑ *x*
⟨*proof*⟩

lemma *cb-take-idem*: *cb-take n* (*cb-take n x*) = *cb-take n x*
⟨*proof*⟩

lemma *cb-take-mono*: *x* ⊑ *y* ⟹ *cb-take n x* ⊑ *cb-take n y*
⟨*proof*⟩

lemma *cb-take-chain-le*: *m* ≤ *n* ⟹ *cb-take m x* ⊑ *cb-take n x*
⟨*proof*⟩

lemma *range-const*: *range* (λ*x*. *c*) = {*c*}
⟨*proof*⟩

lemma *finite-range-cb-take*: *finite* (*range* (*cb-take n*))
⟨*proof*⟩

definition
rank :: *'a compact-basis* ⇒ *nat*
where
rank x = (*LEAST n*. *cb-take n x* = *x*)

lemma *compact-approx-rank*: *cb-take* (*rank x*) *x* = *x*
⟨*proof*⟩

lemma *rank-leD*: *rank x* ≤ *n* ⟹ *cb-take n x* = *x*
⟨*proof*⟩

lemma *rank-leI*: *cb-take n x* = *x* ⟹ *rank x* ≤ *n*
⟨*proof*⟩

lemma *rank-le-iff*: *rank x* ≤ *n* ⟷ *cb-take n x* = *x*
⟨*proof*⟩

lemma *rank-compact-bot* [*simp*]: *rank compact-bot* = *0*
⟨*proof*⟩

lemma *rank-eq-0-iff* [*simp*]: *rank x* = *0* ⟷ *x* = *compact-bot*
⟨*proof*⟩

definition
rank-le :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-le x = {*y*. *rank y* ≤ *rank x*}

definition
rank-lt :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-lt x = {*y*. *rank y* < *rank x*}

definition
rank-eq :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-eq x = {*y*. *rank y* = *rank x*}

lemma *rank-eq-cong*: *rank x* = *rank y* ⟹ *rank-eq x* = *rank-eq y*
⟨*proof*⟩

lemma *rank-lt-cong*: *rank x* = *rank y* ⟹ *rank-lt x* = *rank-lt y*
⟨*proof*⟩

lemma *rank-eq-subset*: *rank-eq x* ⊆ *rank-le x*
⟨*proof*⟩

lemma *rank-lt-subset*: *rank-lt x* ⊆ *rank-le x*
⟨*proof*⟩

lemma *finite-rank-le*: *finite* (*rank-le x*)
⟨*proof*⟩

lemma *finite-rank-eq*: *finite* (*rank-eq x*)
⟨*proof*⟩

lemma *finite-rank-lt*: *finite* (*rank-lt x*)
⟨*proof*⟩

lemma *rank-lt-Int-rank-eq*: *rank-lt x* ∩ *rank-eq x* = {}
⟨*proof*⟩

lemma *rank-lt-Un-rank-eq*: *rank-lt x* ∪ *rank-eq x* = *rank-le x*
⟨*proof*⟩

30.7.3 Sequencing basis elements

definition
place :: *'a compact-basis* ⇒ *nat*
where
place x = *card* (*rank-lt x*) + *choose-pos* (*rank-eq x*) *x*

lemma *place-bounded*: *place x* < *card* (*rank-le x*)
⟨*proof*⟩

lemma *place-ge*: *card* (*rank-lt x*) ≤ *place x*
⟨*proof*⟩

lemma *place-rank-mono*:
fixes *x y* :: *'a compact-basis*
shows *rank x* < *rank y* ⟹ *place x* < *place y*
⟨*proof*⟩

lemma *place-eqD*: *place x* = *place y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *inj-place*: *inj place*
⟨*proof*⟩

30.7.4 Embedding and projection on basis elements

definition
sub :: *'a compact-basis* ⇒ *'a compact-basis*
where
sub x = (*case rank x of 0* ⇒ *compact-bot* | *Suc k* ⇒ *cb-take k x*)

lemma *rank-sub-less*: *x* ≠ *compact-bot* ⟹ *rank* (*sub x*) < *rank x*
⟨*proof*⟩

lemma *place-sub-less*: $x \neq \mathit{compact\text{-}bot} \Longrightarrow \mathit{place}\ (\mathit{sub}\ x) < \mathit{place}\ x$
⟨*proof*⟩

lemma *sub-below*: $\mathit{sub}\ x \sqsubseteq x$
⟨*proof*⟩

lemma *rank-less-imp-below-sub*: $[\![x \sqsubseteq y;\ \mathit{rank}\ x < \mathit{rank}\ y]\!] \Longrightarrow x \sqsubseteq \mathit{sub}\ y$
⟨*proof*⟩

function
basis-emb :: $'a$ *compact-basis* ⇒ *ubasis*
where
basis-emb $x = (\mathit{if}\ x = \mathit{compact\text{-}bot}\ \mathit{then}\ 0\ \mathit{else}$
node (*place* x) (*basis-emb* (*sub* x))
(*basis-emb* ‘ $\{y.\ \mathit{place}\ y < \mathit{place}\ x \wedge x \sqsubseteq y\}$))
⟨*proof*⟩

termination *basis-emb*
⟨*proof*⟩

declare *basis-emb.simps* [*simp del*]

lemma *basis-emb-compact-bot* [*simp*]: *basis-emb compact-bot* = 0
⟨*proof*⟩

lemma *fin1*: *finite* $\{y.\ \mathit{place}\ y < \mathit{place}\ x \wedge x \sqsubseteq y\}$
⟨*proof*⟩

lemma *fin2*: *finite* (*basis-emb* ‘ $\{y.\ \mathit{place}\ y < \mathit{place}\ x \wedge x \sqsubseteq y\}$)
⟨*proof*⟩

lemma *rank-place-mono*:
$[\![\mathit{place}\ x < \mathit{place}\ y;\ x \sqsubseteq y]\!] \Longrightarrow \mathit{rank}\ x < \mathit{rank}\ y$
⟨*proof*⟩

lemma *basis-emb-mono*:
$x \sqsubseteq y \Longrightarrow \mathit{ubasis\text{-}le}\ (\mathit{basis\text{-}emb}\ x)\ (\mathit{basis\text{-}emb}\ y)$
⟨*proof*⟩

lemma *inj-basis-emb*: *inj basis-emb*
⟨*proof*⟩

definition
basis-prj :: *ubasis* ⇒ $'a$ *compact-basis*
where
basis-prj x = *inv basis-emb*
(*ubasis-until* ($\lambda x.\ x \in \mathit{range}$ (*basis-emb* :: $'a$ *compact-basis* ⇒ *ubasis*)) x)

lemma *basis-prj-basis-emb*: $\bigwedge x.\ \mathit{basis\text{-}prj}\ (\mathit{basis\text{-}emb}\ x) = x$

⟨*proof*⟩

lemma *basis-prj-node*:
⟦*finite S*; *node i a S* ∉ *range* (*basis-emb* :: *'a compact-basis* ⇒ *nat*)⟧
⟹ *basis-prj* (*node i a S*) = (*basis-prj a* :: *'a compact-basis*)
⟨*proof*⟩

lemma *basis-prj-0*: *basis-prj 0* = *compact-bot*
⟨*proof*⟩

lemma *node-eq-basis-emb-iff*:
finite S ⟹ *node i a S* = *basis-emb x* ⟷
x ≠ *compact-bot* ∧ *i* = *place x* ∧ *a* = *basis-emb* (*sub x*) ∧
S = *basis-emb* ‘ {*y*. *place y* < *place x* ∧ *x* ⊑ *y*}
⟨*proof*⟩

lemma *basis-prj-mono*: *ubasis-le a b* ⟹ *basis-prj a* ⊑ *basis-prj b*
⟨*proof*⟩

lemma *basis-emb-prj-less*: *ubasis-le* (*basis-emb* (*basis-prj x*)) *x*
⟨*proof*⟩

hide (**open**) *const*
node
choose
choose-pos
place
sub

30.7.5 EP-pair from any bifinite domain into *udom*

definition
udom-emb :: *'a*::*bifinite* → *udom*
where
udom-emb = *compact-basis.basis-fun* (λ*x*. *udom-principal* (*basis-emb x*))

definition
udom-prj :: *udom* → *'a*::*bifinite*
where
udom-prj = *udom.basis-fun* (λ*x*. *Rep-compact-basis* (*basis-prj x*))

lemma *udom-emb-principal*:
udom-emb·(*Rep-compact-basis x*) = *udom-principal* (*basis-emb x*)
⟨*proof*⟩

lemma *udom-prj-principal*:
udom-prj·(*udom-principal x*) = *Rep-compact-basis* (*basis-prj x*)
⟨*proof*⟩

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
⟨*proof*⟩

end

31 Sum-Cpo: The cpo of disjoint sums

theory *Sum-Cpo*
imports *Bifinite*
begin

31.1 Ordering on type $'a + 'b$

instantiation + :: (*sq-ord*, *sq-ord*) *sq-ord*
begin

definition
less-sum-def: $x \sqsubseteq y \equiv$ *case* x *of*
Inl $a \Rightarrow$ (*case* y *of Inl* $b \Rightarrow a \sqsubseteq b \mid$ *Inr* $b \Rightarrow$ *False*) $\mid$
Inr $a \Rightarrow$ (*case* y *of Inl* $b \Rightarrow$ *False* $\mid$ *Inr* $b \Rightarrow a \sqsubseteq b$)

instance ⟨*proof*⟩
end

lemma *Inl-less-iff* [*simp*]: *Inl* $x \sqsubseteq$ *Inl* $y = x \sqsubseteq y$
⟨*proof*⟩

lemma *Inr-less-iff* [*simp*]: *Inr* $x \sqsubseteq$ *Inr* $y = x \sqsubseteq y$
⟨*proof*⟩

lemma *Inl-less-Inr* [*simp*]: $\neg$ *Inl* $x \sqsubseteq$ *Inr* y
⟨*proof*⟩

lemma *Inr-less-Inl* [*simp*]: $\neg$ *Inr* $x \sqsubseteq$ *Inl* y
⟨*proof*⟩

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

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

lemma *Inl-lessE*: $[\![$*Inl* $a \sqsubseteq x$; $\bigwedge b.\ [\![x =$ *Inl* b; $a \sqsubseteq b]\!] \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *Inr-lessE*: $[\![$*Inr* $a \sqsubseteq x$; $\bigwedge b.\ [\![x =$ *Inr* b; $a \sqsubseteq b]\!] \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

lemmas *sum-less-elims = Inl-lessE Inr-lessE*

lemma *sum-less-cases*:
$[\![x \sqsubseteq y;$
$\bigwedge a\ b.\ [\![x = Inl\ a;\ y = Inl\ b;\ a \sqsubseteq b]\!] \Longrightarrow R;$
$\bigwedge a\ b.\ [\![x = Inr\ a;\ y = Inr\ b;\ a \sqsubseteq b]\!] \Longrightarrow R]\!]$
$\Longrightarrow R$
⟨*proof*⟩

31.2 Sum type is a complete partial order

instance $+$ :: (*po*, *po*) *po*
⟨*proof*⟩

lemma *monofun-inv-Inl*: *monofun* $(\lambda p.\ THE\ a.\ p = Inl\ a)$
⟨*proof*⟩

lemma *monofun-inv-Inr*: *monofun* $(\lambda p.\ THE\ b.\ p = Inr\ b)$
⟨*proof*⟩

lemma *sum-chain-cases*:
assumes *Y*: *chain Y*
assumes *A*: $\bigwedge A.\ [\![chain\ A;\ Y = (\lambda i.\ Inl\ (A\ i))]\!] \Longrightarrow R$
assumes *B*: $\bigwedge B.\ [\![chain\ B;\ Y = (\lambda i.\ Inr\ (B\ i))]\!] \Longrightarrow R$
shows *R*
⟨*proof*⟩

lemma *is-lub-Inl*: *range S* $<<|$ *x* $\Longrightarrow$ *range* $(\lambda i.\ Inl\ (S\ i))$ $<<|$ *Inl x*
⟨*proof*⟩

lemma *is-lub-Inr*: *range S* $<<|$ *x* $\Longrightarrow$ *range* $(\lambda i.\ Inr\ (S\ i))$ $<<|$ *Inr x*
⟨*proof*⟩

instance $+$ :: (*cpo*, *cpo*) *cpo*
⟨*proof*⟩

31.3 Continuity of *Inl*, *Inr*, *sum-case*

lemma *cont2cont-Inl* [*simp*]: *cont f* $\Longrightarrow$ *cont* $(\lambda x.\ Inl\ (f\ x))$
⟨*proof*⟩

lemma *cont2cont-Inr* [*simp*]: *cont f* $\Longrightarrow$ *cont* $(\lambda x.\ Inr\ (f\ x))$
⟨*proof*⟩

lemma *cont-Inl*: *cont Inl*
⟨*proof*⟩

lemma *cont-Inr*: *cont Inr*
⟨*proof*⟩

lemmas *ch2ch-Inl* [*simp*] = *ch2ch-cont* [*OF cont-Inl*]
lemmas *ch2ch-Inr* [*simp*] = *ch2ch-cont* [*OF cont-Inr*]

lemmas *lub-Inl* = *cont2contlubE* [*OF cont-Inl*, *symmetric*]
lemmas *lub-Inr* = *cont2contlubE* [*OF cont-Inr*, *symmetric*]

lemma *cont-sum-case1*:
assumes *f*: $\bigwedge a.\ cont\ (\lambda x.\ f\ x\ a)$
assumes *g*: $\bigwedge b.\ cont\ (\lambda x.\ g\ x\ b)$
shows *cont* $(\lambda x.\ case\ y\ of\ Inl\ a \Rightarrow f\ x\ a \mid Inr\ b \Rightarrow g\ x\ b)$
⟨*proof*⟩

lemma *cont-sum-case2*: $[\![cont\ f;\ cont\ g]\!] \Longrightarrow cont\ (sum\text{-}case\ f\ g)$
⟨*proof*⟩

lemma *cont2cont-sum-case* [*simp*]:
assumes *f1*: $\bigwedge a.\ cont\ (\lambda x.\ f\ x\ a)$ **and** *f2*: $\bigwedge x.\ cont\ (\lambda a.\ f\ x\ a)$
assumes *g1*: $\bigwedge b.\ cont\ (\lambda x.\ g\ x\ b)$ **and** *g2*: $\bigwedge x.\ cont\ (\lambda b.\ g\ x\ b)$
assumes *h*: *cont* $(\lambda x.\ h\ x)$
shows *cont* $(\lambda x.\ case\ h\ x\ of\ Inl\ a \Rightarrow f\ x\ a \mid Inr\ b \Rightarrow g\ x\ b)$
⟨*proof*⟩

31.4 Compactness and chain-finiteness

lemma *compact-Inl*: *compact a* $\Longrightarrow$ *compact* (*Inl a*)
⟨*proof*⟩

lemma *compact-Inr*: *compact a* $\Longrightarrow$ *compact* (*Inr a*)
⟨*proof*⟩

lemma *compact-Inl-rev*: *compact* (*Inl a*) $\Longrightarrow$ *compact a*
⟨*proof*⟩

lemma *compact-Inr-rev*: *compact* (*Inr a*) $\Longrightarrow$ *compact a*
⟨*proof*⟩

lemma *compact-Inl-iff* [*simp*]: *compact* (*Inl a*) = *compact a*
⟨*proof*⟩

lemma *compact-Inr-iff* [*simp*]: *compact* (*Inr a*) = *compact a*
⟨*proof*⟩

instance + :: (*chfin*, *chfin*) *chfin*
⟨*proof*⟩

instance + :: (*finite-po*, *finite-po*) *finite-po* ⟨*proof*⟩

instance + :: (*discrete-cpo*, *discrete-cpo*) *discrete-cpo*
⟨*proof*⟩

31.5 Sum type is a bifinite domain

instantiation + :: (*profinite*, *profinite*) *profinite*
begin

definition
approx-sum-def: *approx* =
(λ*n*. Λ *x*. *case x of Inl a* ⇒ *Inl* (*approx n*·*a*) | *Inr b* ⇒ *Inr* (*approx n*·*b*))

lemma *approx-Inl* [*simp*]: *approx n*·(*Inl x*) = *Inl* (*approx n*·*x*)
⟨*proof*⟩

lemma *approx-Inr* [*simp*]: *approx n*·(*Inr x*) = *Inr* (*approx n*·*x*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

end

theory *HOLCF*
imports
Domain ConvexPD Algebraic Universal Sum-Cpo Main
uses
holcf-logic.ML
Tools/cont-consts.ML
Tools/cont-proc.ML
Tools/domain/domain-library.ML
Tools/domain/domain-syntax.ML
Tools/domain/domain-axioms.ML
Tools/domain/domain-theorems.ML
Tools/domain/domain-extender.ML
Tools/adm-tac.ML
begin

defaultsort *pcpo*

⟨*ML*⟩

end