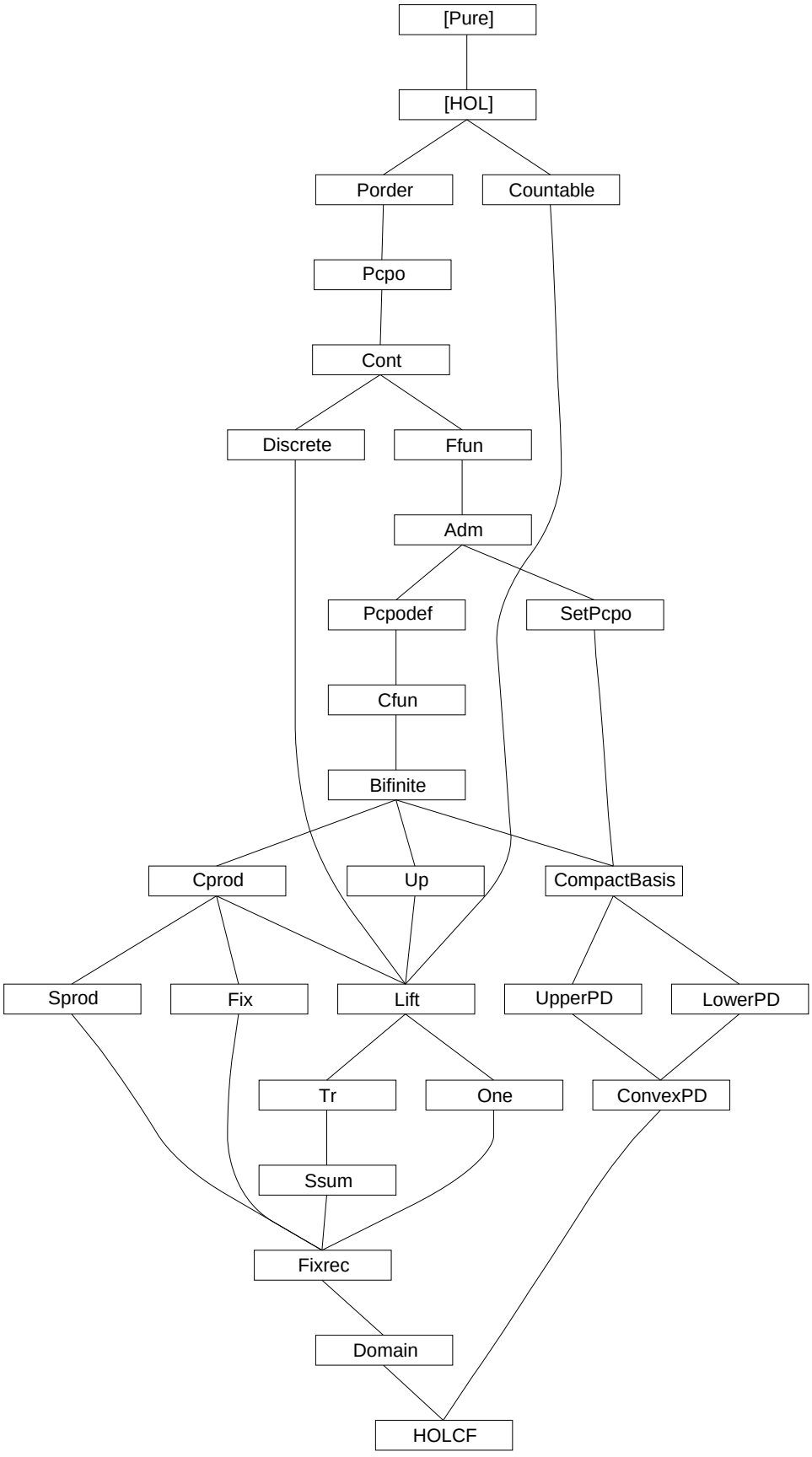
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

June 8, 2008

Contents


[Pure]
[HOL]
Porder
Countable
Pcpo
Cont
Discrete
Ffun
Adm
Pcpodef
SetPcpo
Cfun
Bifinite
Cprod
Up
CompactBasis
Sprod
Fix
Lift
UpperPD
LowerPD
Tr
One
ConvexPD
Ssum
Fixrec
Domain
HOLCF

1 Porder: Partial orders

theory *Porder*
imports *Datatype Finite-Set*
begin

1.1 Type class for partial orders

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

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

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

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

class *po* = *preorder* +
  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*: $S <<| x \Longrightarrow S <| x$
⟨*proof*⟩

lemma *is-lub-lub*: $[\![S <<| x; S <| u]\!] \Longrightarrow x \sqsubseteq u$
⟨*proof*⟩

lemma *is-lubI*: $[\![S <| x; \bigwedge u.\ S <| u \Longrightarrow x \sqsubseteq u]\!] \Longrightarrow S <<| x$
⟨*proof*⟩

lubs are unique

lemma *unique-lub*: $[\![S <<| x; S <<| y]\!] \Longrightarrow x = y$
⟨*proof*⟩

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

lemma *lubI*: $M <<| x \Longrightarrow M <<| lub\ M$
⟨*proof*⟩

lemma *thelubI*: $M <<| l \Longrightarrow lub\ M = l$
⟨*proof*⟩

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

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

lemma *is-lub-bin*: $x \sqsubseteq y \Longrightarrow \{x, y\} <<| y$
⟨*proof*⟩

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

lemma *is-lub-maximal*: $[\![S <| x; x \in S]\!] \Longrightarrow S <<| x$
⟨*proof*⟩

lemma *lub-maximal*: $[\![S <| x; x \in S]\!] \Longrightarrow lub\ S = x$
⟨*proof*⟩

1.4 Countable chains

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

lemma *chainI*: $(\bigwedge i.\ Y\ i \sqsubseteq Y\ (Suc\ i)) \Longrightarrow chain\ Y$
⟨*proof*⟩

lemma *chainE*: $chain\ Y \Longrightarrow Y\ i \sqsubseteq Y\ (Suc\ i)$
⟨*proof*⟩

chains are monotone functions

lemma *chain-mono*:
assumes Y: *chain* Y
shows $i \leq j \Longrightarrow Y\ i \sqsubseteq Y\ j$
⟨*proof*⟩

lemma *chain-mono-less*: $[\![chain\ Y; i < j]\!] \Longrightarrow Y\ i \sqsubseteq Y\ j$
⟨*proof*⟩

lemma *chain-shift*: $chain\ Y \Longrightarrow chain\ (\lambda i.\ Y\ (i + j))$
⟨*proof*⟩

technical lemmas about (least) upper bounds of chains

lemma *is-ub-lub*: $range\ S <<| x \Longrightarrow S\ i \sqsubseteq x$
⟨*proof*⟩

lemma *is-ub-range-shift*:
$chain\ S \Longrightarrow range\ (\lambda i.\ S\ (i + j)) <| x = range\ S <| x$
⟨*proof*⟩

lemma *is-lub-range-shift*:
$chain\ S \Longrightarrow range\ (\lambda i.\ S\ (i + j)) <<| x = range\ S <<| x$
⟨*proof*⟩

the lub of a constant chain is the constant

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

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

lemma *thelub-const* [*simp*]: $(\bigsqcup i.\ c) = c$

⟨*proof*⟩

1.5 Totally ordered sets

definition
— Arbitrary chains are total orders
tord :: *'a::po set* ⇒ *bool* **where**
tord S = (∀ *x y*. *x* ∈ *S* ∧ *y* ∈ *S* ⟶ (*x* ⊑ *y* ∨ *y* ⊑ *x*))

The range of a chain is a totally ordered

lemma *chain-tord*: *chain Y* ⟹ *tord* (*range Y*)
⟨*proof*⟩

lemma *finite-tord-has-max*:
⟦*finite S*; *S* ≠ {}; *tord S*⟧ ⟹ ∃ *y*∈*S*. ∀ *x*∈*S*. *x* ⊑ *y*
⟨*proof*⟩

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

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

lemma *bin-chainmax*:
$x \sqsubseteq y \Longrightarrow$ *max-in-chain* $(Suc\ 0)$ $(\lambda i.$ *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*⟩

1.7 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*]: $\neg$ *directed* $\{\}$
⟨*proof*⟩

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

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

lemma *directed-chain*: *chain* $S \Longrightarrow$ *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

axclass *cpo* < *po*
— class axiom:
cpo: *chain* $S \Longrightarrow \exists x.$ *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* $S <<|$ *lub* (*range* S)
⟨*proof*⟩

lemma *thelubE*: $[\![$*chain* S; ($\bigsqcup i.\ S\ i$) = (l::$'a$::*cpo*)$]\!] \Longrightarrow$ *range* $S <<|\ l$
⟨*proof*⟩

Properties of the lub

lemma *is-ub-thelub*: *chain* (S::*nat* ⇒ $'a$::*cpo*) ⟹ $S\ x \sqsubseteq (\bigsqcup i.\ S\ i)$
⟨*proof*⟩

lemma *is-lub-thelub*:
$[\![$*chain* (S::*nat* ⇒ $'a$::*cpo*); *range* $S <|\ x]\!] \Longrightarrow (\bigsqcup i.\ S\ i) \sqsubseteq x$
⟨*proof*⟩

lemma *lub-range-mono*:
$[\![range\ X \subseteq range\ Y;\ chain\ Y;\ chain\ (X::nat \Rightarrow 'a::cpo)]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *lub-range-shift*:
$chain\ (Y::nat \Rightarrow 'a::cpo) \Longrightarrow (\bigsqcup i.\ Y\ (i + j)) = (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *maxinch-is-thelub*:
$chain\ Y \Longrightarrow max\text{-}in\text{-}chain\ i\ Y = ((\bigsqcup i.\ Y\ i) = ((Y\ i)::'a::cpo))$
⟨*proof*⟩

the $\sqsubseteq$ relation between two chains is preserved by their lubs

lemma *lub-mono*:
$[\![chain\ (X::nat \Rightarrow 'a::cpo);\ chain\ Y;\ \bigwedge i.\ X\ i \sqsubseteq Y\ i]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

the $=$ relation between two chains is preserved by their lubs

lemma *lub-equal*:
$[\![chain\ (X::nat \Rightarrow 'a::cpo);\ chain\ Y;\ \forall k.\ X\ k = Y\ k]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) = (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

more results about mono and $=$ of lubs of chains

lemma *lub-mono2*:
$[\![\exists j.\ \forall i{>}j.\ X\ i = Y\ i;\ chain\ (X::nat \Rightarrow 'a::cpo);\ chain\ Y]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *lub-equal2*:
$[\![\exists j.\ \forall i{>}j.\ X\ i = Y\ i;\ chain\ (X::nat \Rightarrow 'a::cpo);\ chain\ Y]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) = (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *lub-mono3*:
$[\![chain\ (Y::nat \Rightarrow 'a::cpo);\ chain\ X;\ \forall i.\ \exists j.\ Y\ i \sqsubseteq X\ j]\!]$
$\Longrightarrow (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ X\ i)$
⟨*proof*⟩

lemma *ch2ch-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 $chain\ (\lambda i.\ \bigsqcup j.\ Y\ i\ j)$
⟨*proof*⟩

lemma *diag-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 $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) = (\bigsqcup i.\ Y\ i\ i)$
⟨*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

axclass *pcpo* < *cpo*
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

axclass *finite-po* < *finite*, *po*

axclass *chfin* < *po*
chfin: $chain\ Y \Longrightarrow \exists n.\ max\text{-}in\text{-}chain\ n\ Y$

axclass *flat* < *pcpo*
ax-flat: $x \sqsubseteq y \Longrightarrow (x = \bot) \vee (x = y)$

finite partial orders are chain-finite and directed-complete

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

instance *finite-po* < *cpo*
⟨*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-chain* Y
⟨*proof*⟩

Discrete cpos

axclass *discrete-cpo* $<$ *sq-ord*
discrete-cpo [*simp*]: $x \sqsubseteq y \longleftrightarrow x = y$

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

In a discrete cpo, every chain is constant

lemma *discrete-chain-const*:
assumes S: *chain* $(S::nat \Rightarrow 'a::discrete\text{-}cpo)$
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 :: $('a \Rightarrow 'b) \Rightarrow bool$ — 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*⟩

lemma *ub2ub-monofun'*:
$[\![monofun\ f;\ S <|\ u]\!] \Longrightarrow f\ `\ S <|\ f\ u$
⟨*proof*⟩

monotone functions map directed sets to directed sets

lemma *dir2dir-monofun*:
assumes *f*: *monofun f*
assumes *S*: *directed S*
shows *directed* (*f* ` *S*)
⟨*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 *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*: $\bigwedge Y.\ [\![chain\ Y;\ chain\ (\lambda i.\ f\ (Y\ i))]\!]$
$\Longrightarrow f\ (lub\ (range\ Y)) \sqsubseteq (\bigsqcup i.\ f\ (Y\ i))$
shows *cont f*
⟨*proof*⟩

3.3 Continuity of basic functions

The identity function is continuous

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

constant functions are continuous

lemma *cont-const*: *cont* $(\lambda x.\ c)$
⟨*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*⟩

3.4 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 Ffun: Class instances for the full function space

theory *Ffun*

imports *Cont*
begin

4.1 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*⟩

4.2 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* ⇒ *'a*::*type* ⇒ *'b*::*cpo*)
⟹ *range S* <<| (λ*x*. ⨆ *i*. *S i x*)
⟨*proof*⟩

lemma *thelub-fun*:
chain (*S*::*nat* ⇒ *'a*::*type* ⇒ *'b*::*cpo*)
⟹ *lub* (*range S*) = (λ*x*. ⨆ *i*. *S i x*)
⟨*proof*⟩

lemma *cpo-fun*:
chain (*S*::*nat* ⇒ *'a*::*type* ⇒ *'b*::*cpo*) ⟹ ∃ *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*⟩

4.3 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*]: $\bot\ x = \bot$
⟨*proof*⟩

The following results are about application for functions in $'a \Rightarrow 'b$

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

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

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

4.4 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

declare *range-composition* [*simp del*]

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

5 Adm: Admissibility and compactness

theory *Adm*
imports *Ffun*
begin

defaultsort *cpo*

5.1 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 *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*⟩

5.2 Admissibility on chain-finite types

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

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

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

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

declare *range-composition* [*simp del*]

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*⟩

5.4 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*:
(⋀*Y*. ⟦*chain Y*; *x* ⊑ *lub* (*range Y*)⟧ ⟹ ∃ *i*. *x* ⊑ *Y i*) ⟹ *compact x*
⟨*proof*⟩

lemma *compactD2*:
⟦*compact x*; *chain Y*; *x* ⊑ *lub* (*range Y*)⟧ ⟹ ∃ *i*. *x* ⊑ *Y i*
⟨*proof*⟩

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

lemma *compact-imp-max-in-chain*:
⟦*chain Y*; *compact* (⨆ *i*. *Y i*)⟧ ⟹ ∃ *i*. *max-in-chain i Y*
⟨*proof*⟩

admissibility and compactness

lemma *adm-compact-not-less*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. ¬ *k* ⊑ *t x*)
⟨*proof*⟩

lemma *adm-neq-compact*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. *t x* ≠ *k*)
⟨*proof*⟩

lemma *adm-compact-neq*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. *k* ≠ *t x*)
⟨*proof*⟩

lemma *compact-UU* [*simp*, *intro*]: *compact* ⊥
⟨*proof*⟩

lemma *adm-not-UU*: *cont t* ⟹ *adm* (λ*x*. *t x* ≠ ⊥)
⟨*proof*⟩

Any upward-closed predicate is admissible.

lemma *adm-upward*:
assumes *P*: ⋀*x y*. ⟦*P x*; *x* ⊑ *y*⟧ ⟹ *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

6 Pcpodef: Subtypes of pcpos

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

6.1 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.

theorem *typedef-po*:
  fixes *Abs* :: *'a*::*po* ⇒ *'b*::*sq-ord*
  assumes *type*: *type-definition Rep Abs A*
    and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
  shows *OFCLASS*(*'b*, *po-class*)
⟨*proof*⟩

6.2 Proving a subtype is finite

context *type-definition*
begin

lemma *Abs-image*:
  shows *Abs* ‘ *A* = *UNIV*
⟨*proof*⟩

lemma *finite-UNIV*: *finite A* ⟹ *finite* (*UNIV* :: *'b set*)
⟨*proof*⟩

end

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*⟩

6.3 Proving a subtype is chain-finite

lemma *monofun-Rep*:
  assumes *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *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* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
shows *OFCLASS*(*'b*, *chfin-class*)
⟨*proof*⟩

6.4 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* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *chain S* ⟹ *Rep* (*Abs* (⨆ *i*. *Rep* (*S i*))) = (⨆ *i*. *Rep* (*S i*))
⟨*proof*⟩

theorem *typedef-lub*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*po*
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 *chain S* ⟹ *range S* <<| *Abs* (⨆ *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* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *OFCLASS*(*'b*, *cpo-class*)
⟨*proof*⟩

6.4.1 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*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows $range\ (\lambda i.\ Rep\ (S\ i)) <<|\ Rep\ x \Longrightarrow range\ S <<|\ x$
⟨*proof*⟩

theorem *typedef-cont-Abs*:
fixes $Abs :: 'a{::}cpo \Rightarrow 'b{::}cpo$
fixes $f :: 'c{::}cpo \Rightarrow 'a{::}cpo$
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
and *f-in-A*: $\bigwedge x.\ f\ x \in A$
and *cont-f*: $cont\ f$
shows $cont\ (\lambda x.\ Abs\ (f\ x))$
⟨*proof*⟩

6.5 Proving subtype elements are compact

theorem *typedef-compact*:
fixes $Abs :: 'a{::}cpo \Rightarrow 'b{::}cpo$
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 $compact\ (Rep\ k) \Longrightarrow compact\ k$
⟨*proof*⟩

6.6 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 $Abs :: 'a{::}cpo \Rightarrow 'b{::}cpo$
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *z-in-A*: $z \in A$
and *z-least*: $\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x$
shows $OFCLASS('b,\ pcpo\text{-}class)$
⟨*proof*⟩

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

theorem *typedef-pcpo*:
fixes $Abs :: 'a{::}pcpo \Rightarrow 'b{::}cpo$
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*⟩

6.6.1 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*⟩

6.7 Proving a subtype is flat

theorem *typedef-flat*:
fixes *Abs* :: *'a*::*flat* ⇒ *'b*::*pcpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *OFCLASS*(*'b*, *flat-class*)
⟨*proof*⟩

6.8 HOLCF type definition package

⟨*ML*⟩

end

7 Cfun: The type of continuous functions

theory *Cfun*
imports *Pcpodef Ffun*
uses (*Tools/cont-proc.ML*)
begin

defaultsort *cpo*

7.1 Definition of continuous function type

lemma *Ex-cont*: ∃ *f*. *cont f*
⟨*proof*⟩

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

cpodef (*CFun*) (*'a*, *'b*) -> (**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*)

7.2 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*)

7.3 Continuous function space is pointed

lemma *UU-CFun*: ⊥ ∈ *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*]: ⊥·*x* = ⊥
⟨*proof*⟩

for compatibility with old HOLCF-Version

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

7.4 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*⟩

7.5 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* $\Longrightarrow$ $f{\cdot}(lub\ (range\ Y)) = (\bigsqcup i.\ f{\cdot}(Y\ i))$
⟨*proof*⟩

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

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

lemma *cont-cfun-fun*: *chain F* $\Longrightarrow$ *range* $(\lambda i.\ F\ i{\cdot}x)$ <<| $lub\ (range\ F){\cdot}x$
⟨*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*: $[\![f \sqsubseteq g;\ x \sqsubseteq y]\!] \Longrightarrow f{\cdot}x \sqsubseteq g{\cdot}y$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *contlub-LAM*:
$[\![\bigwedge x.\ chain\ (\lambda i.\ F\ i\ x);\ \bigwedge i.\ cont\ (\lambda x.\ F\ i\ x)]\!]$
$\Longrightarrow (\Lambda\ x.\ \bigsqcup i.\ F\ i\ x) = (\bigsqcup i.\ \Lambda\ x.\ F\ i\ x)$
⟨*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*: $chain\ F \Longrightarrow monofun\ (\lambda x.\ \bigsqcup i.\ F\ i\cdot x)$
⟨*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*: $chain\ F \Longrightarrow lub\ (range\ F) = (\Lambda\ x.\ \bigsqcup i.\ F\ i\cdot x)$
⟨*proof*⟩

7.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

lemma *cont2cont-Rep-CFun*:
$[\![cont\ f;\ cont\ t]\!] \Longrightarrow cont\ (\lambda x.\ (f\ x)\cdot(t\ x))$

⟨*proof*⟩

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

lemma *cont2mono-LAM*:
assumes *p1*: $!!x.\ cont(c1\ x)$
assumes *p2*: $!!y.\ monofun(\%x.\ c1\ x\ y)$
shows $monofun(\%x.\ LAM\ y.\ c1\ x\ y)$
⟨*proof*⟩

cont2cont Lemma for $\lambda x.\ \Lambda\ y.\ c1\ x\ y$

lemma *cont2cont-LAM*:
assumes *p1*: $!!x.\ cont(c1\ x)$
assumes *p2*: $!!y.\ cont(\%x.\ c1\ x\ y)$
shows $cont(\%x.\ LAM\ y.\ c1\ x\ y)$
⟨*proof*⟩

continuity simplification procedure

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

⟨*ML*⟩

7.7 Miscellaneous

Monotonicity of *Abs-CFun*

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

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

lemma *chfin-Rep-CFunR*: $chain\ (Y{::}nat \Rightarrow {'a}{::}cpo \rightarrow {'b}{::}chfin)$
$\Longrightarrow\ !s.\ ?\ n.\ lub(range(Y))\$s = Y\ n\$s$
⟨*proof*⟩

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

7.8 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*:
$\forall y.\ (f{::}'a{::}\mathit{chfin} \to 'b){\cdot}(g{\cdot}y) = y$
$\Longrightarrow \forall Y{::}\mathit{nat} \Rightarrow 'b.\ \mathit{chain}\ Y \longrightarrow (\exists n.\ \mathit{max\text{-}in\text{-}chain}\ n\ Y)$
⟨*proof*⟩

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

a result about functions with flat codomain

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

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

7.9 Identity and composition

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

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

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

lemma *ID1* [*simp*]: $\mathit{ID}{\cdot}x = x$
⟨*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-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*]: $f\ oo\ ID = f$
⟨*proof*⟩

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

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

7.10 Strictified functions

defaultsort *pcpo*

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

results about strictify

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

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

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

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

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

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

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

7.11 Continuous let-bindings

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

syntax
-CLet :: [*letbinds*, $'a$] => $'a$ ((*Let* (-)/ *in* (-)) *10*)

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

end

8 Bifinite: Bifinite domains and approximation

theory *Bifinite*
imports *Cfun*
begin

8.1 Omega-profinite and bifinite domains

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

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

lemma *finite-range-imp-finite-fixes*:
finite $\{x.\ \exists\, y.\ x = f\ y\} \Longrightarrow$ *finite* $\{x.\ f\ x = x\}$
⟨*proof*⟩

lemma *chain-approx* [*simp*]:
chain (*approx* :: *nat* ⇒ $'a$::*profinite* $\to$ $'a$)
⟨*proof*⟩

lemma *lub-approx* [*simp*]: ($\bigsqcup i.$ *approx* i) = ($\Lambda(x{::}'a{::}\mathit{profinite}).\ x$)
⟨*proof*⟩

lemma *approx-less*: *approx* $i{\cdot}x \sqsubseteq (x{::}'a{::}\mathit{profinite})$

⟨*proof*⟩

lemma *approx-strict* [*simp*]: *approx i*·(⊥::′*a*::*bifinite*) = ⊥
⟨*proof*⟩

lemma *approx-approx1*:
i ≤ *j* ⟹ *approx i*·(*approx j*·*x*) = *approx i*·(*x*::′*a*::*profinite*)
⟨*proof*⟩

lemma *approx-approx2*:
j ≤ *i* ⟹ *approx i*·(*approx j*·*x*) = *approx j*·(*x*::′*a*::*profinite*)
⟨*proof*⟩

lemma *approx-approx* [*simp*]:
approx i·(*approx j*·*x*) = *approx* (*min i j*)·(*x*::′*a*::*profinite*)
⟨*proof*⟩

lemma *idem-fixes-eq-range*:
∀ *x*. *f* (*f x*) = *f x* ⟹ {*x*. *f x* = *x*} = {*y*. ∃ *x*. *y* = *f x*}
⟨*proof*⟩

lemma *finite-approx*: *finite* {*y*::′*a*::*profinite*. ∃ *x*. *y* = *approx n*·*x*}
⟨*proof*⟩

lemma *finite-range-approx*:
finite (*range* (λ*x*::′*a*::*profinite*. *approx n*·*x*))
⟨*proof*⟩

lemma *compact-approx* [*simp*]:
fixes *x* :: ′*a*::*profinite*
shows *compact* (*approx n*·*x*)
⟨*proof*⟩

lemma *bifinite-compact-eq-approx*:
fixes *x* :: ′*a*::*profinite*
assumes *x*: *compact x*
shows ∃ *i*. *approx i*·*x* = *x*
⟨*proof*⟩

lemma *bifinite-compact-iff*:
compact (*x*::′*a*::*profinite*) = (∃ *n*. *approx n*·*x* = *x*)
⟨*proof*⟩

lemma *approx-induct*:
assumes *adm*: *adm P* **and** *P*: ⋀*n x*. *P* (*approx n*·*x*)
shows *P* (*x*::′*a*::*profinite*)
⟨*proof*⟩

lemma *bifinite-less-ext*:

fixes $x\ y :: 'a{::}profinite$
shows $(\bigwedge i.\ approx\ i{\cdot}x \sqsubseteq approx\ i{\cdot}y) \Longrightarrow x \sqsubseteq y$
⟨*proof*⟩

8.2 Instance for continuous function space

lemma *finite-range-lemma*:
fixes $h :: 'a{::}cpo \to 'b{::}cpo$
fixes $k :: 'c{::}cpo \to 'd{::}cpo$
shows $[\![finite\ \{y.\ \exists x.\ y = h{\cdot}x\};\ finite\ \{y.\ \exists x.\ y = k{\cdot}x\}]\!]$
$\Longrightarrow finite\ \{g.\ \exists f.\ g = (\Lambda\ x.\ k{\cdot}(f{\cdot}(h{\cdot}x)))\}$
⟨*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*: $approx\ n{\cdot}f{\cdot}x = approx\ n{\cdot}(f{\cdot}(approx\ n{\cdot}x))$
⟨*proof*⟩

end

9 Cprod: The cpo of cartesian products

theory *Cprod*
imports *Bifinite*
begin

defaultsort *cpo*

9.1 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*⟩

definition
unit-when :: $'a \to unit \to 'a$ **where**
unit-when = (Λ *a* -. *a*)

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

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

9.2 Product type is a partial order

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

definition
less-cprod-def: (*op* ⊑) ≡ λ*p1 p2*. (*fst p1* ⊑ *fst p2* ∧ *snd p1* ⊑ *snd p2*)

instance ⟨*proof*⟩
end

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

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

lemma *prod-lessI*: ⟦*fst p* ⊑ *fst q*; *snd p* ⊑ *snd q*⟧ ⟹ *p* ⊑ *q*
⟨*proof*⟩

lemma *Pair-less-iff* [*simp*]: (*a*, *b*) ⊑ (*c*, *d*) = (*a* ⊑ *c* ∧ *b* ⊑ *d*)
⟨*proof*⟩

Pair (-,-) is monotone in both arguments

lemma *monofun-pair1*: *monofun* (λ*x*. (*x*, *y*))
⟨*proof*⟩

lemma *monofun-pair2*: *monofun* (λ*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*⟩

9.4 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 $S :: nat \Rightarrow ('a::cpo \times 'b::cpo)$
assumes *S*: *chain S*
shows *range S* $<<|$ $(\bigsqcup i.\ fst\ (S\ i), \bigsqcup i.\ snd\ (S\ i))$
⟨*proof*⟩

lemma *thelub-cprod*:
chain $(S::nat \Rightarrow 'a::cpo \times 'b::cpo)$
$\Longrightarrow lub\ (range\ S) = (\bigsqcup i.\ fst\ (S\ i), \bigsqcup i.\ snd\ (S\ i))$
⟨*proof*⟩

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

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

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

9.5 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*⟩

9.6 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*⟩

9.7 Continuous versions of constants

definition
cpair :: $'a \to 'b \to ('a * 'b)$ — continuous pairing **where**
cpair = $(\Lambda\ x\ y.\ (x,\ y))$

definition
cfst :: $('a * 'b) \to 'a$ **where**
cfst = $(\Lambda\ p.\ fst\ p)$

definition
csnd :: $('a * 'b) \to 'b$ **where**
csnd = $(\Lambda\ p.\ snd\ p)$

definition
csplit :: $('a \to 'b \to 'c) \to ('a * 'b) \to 'c$ **where**
csplit = $(\Lambda\ f\ p.\ f{\cdot}(cfst{\cdot}p){\cdot}(csnd{\cdot}p))$

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

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

translations
$\langle x,\ y,\ z\rangle$ == $\langle x,\ \langle y,\ z\rangle\rangle$
$\langle x,\ y\rangle$ == *CONST cpair*·*x*·*y*

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

9.8 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*⟩

instance $*$ :: $(chfin,\ chfin)\ chfin$
⟨*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*:
$chain\ S \Longrightarrow lub\ (range\ S) = <\bigsqcup i.\ cfst{\cdot}(S\ i),\ \bigsqcup i.\ csnd{\cdot}(S\ i)>$
⟨*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*

9.9 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

10 Sprod: The type of strict products

theory *Sprod*
imports *Cprod*
begin

defaultsort *pcpo*

10.1 Definition of strict product type

pcpodef (*Sprod*) $('a,\ 'b)\ **$ (**infixr** $**$ *20*) =
$\{p{::}'a \times 'b.\ p = \bot \vee (cfst{\cdot}p \neq \bot \wedge csnd{\cdot}p \neq \bot)\}$
⟨*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*⟩

10.2 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 → 'b → ('a ** 'b)* **where**
spair = (Λ a b. Abs-Sprod
<strictify·(Λ b. a)·b, strictify·(Λ a. b)·a>)

definition
*ssplit :: ('a → 'b → 'c) → ('a ** 'b) → 'c* **where**
ssplit = (Λ f. strictify·(Λ p. f·(sfst·p)·(ssnd·p)))

syntax
*@stuple :: ['a, args] => 'a ** 'b ((1'(:-,/ -:')))*
translations
(:x, y, z:) == (:x, (:y, z:):)
(:x, y:) == CONST spair·x·y

translations
Λ(CONST spair·x·y). t == CONST ssplit·(Λ x y. t)

10.3 Case analysis

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

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

lemma *Exh-Sprod2*:
z = ⊥ ∨ (∃ a b. z = (:a, b:) ∧ a ≠ ⊥ ∧ b ≠ ⊥)
⟨*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*⟩

10.4 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*⟩

10.5 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*⟩

10.6 Compactness

lemma *compact-sfst*: $compact\ x \Longrightarrow compact\ (sfst{\cdot}x)$

⟨*proof*⟩

lemma *compact-ssnd*: *compact* $x \Longrightarrow$ *compact* (*ssnd*·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*⟩

10.7 Properties of *ssplit*

lemma *ssplit1* [*simp*]: *ssplit*·f·$\bot = \bot$
⟨*proof*⟩

lemma *ssplit2* [*simp*]: ⟦$x \neq \bot$; $y \neq \bot$⟧ $\Longrightarrow$ *ssplit*·f·(:x, y:) = f·x·y
⟨*proof*⟩

lemma *ssplit3* [*simp*]: *ssplit*·*spair*·$z = z$
⟨*proof*⟩

10.8 Strict product preserves flatness

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

10.9 Strict product is a bifinite domain

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

definition
approx-sprod-def:
approx = (λn. 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

11 Discrete: Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

11.1 Type *'a discr* is a discrete cpo

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

definition
less-discr-def:
(op ⊑ :: 'a discr ⇒ 'a discr ⇒ bool) = (op =)

instance ⟨*proof*⟩
end

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

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

11.2 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*⟩

11.3 *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

12 Up: The type of lifted values

theory *Up*
imports *Bifinite*
begin

defaultsort *cpo*

12.1 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*

12.2 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* $\sqsubseteq$ *z*
⟨*proof*⟩

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

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

12.3 Lifted cpo is a partial order

instance *u* :: (*cpo*) *po*
⟨*proof*⟩

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

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

12.4 Lifted cpo is a cpo

lemma *is-lub-Iup*:
range S <<| *x* ⟹ *range* (λ*i*. *Iup* (*S i*)) <<| *Iup x*
⟨*proof*⟩

lemma *is-lub-Iup'*: ⟦*directed S*; *S* <<| *x*⟧ ⟹ (*Iup* ‘ *S*) <<| *Iup x*
⟨*proof*⟩

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

lemma *up-lemma1*: *z* ≠ *Ibottom* ⟹ *Iup* (*THE a*. *Iup a* = *z*) = *z*
⟨*proof*⟩

lemma *up-lemma2*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹ *Y* (*i* + *j*) ≠ *Ibottom*
⟨*proof*⟩

lemma *up-lemma3*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹ *Iup* (*THE a*. *Iup a* = *Y* (*i* + *j*)) = *Y* (*i* + *j*)
⟨*proof*⟩

lemma *up-lemma4*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹ *chain* (λ*i*. *THE a*. *Iup a* = *Y* (*i* + *j*))
⟨*proof*⟩

lemma *up-lemma5*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹
(λ*i*. *Y* (*i* + *j*)) = (λ*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*:
$chain\ Y \Longrightarrow$
$(\exists A.\ chain\ A \wedge lub\ (range\ Y) = Iup\ (lub\ (range\ A)) \wedge$
$(\exists j.\ \forall i.\ Y\ (i+j) = Iup\ (A\ i))) \vee (Y = (\lambda i.\ Ibottom))$
⟨*proof*⟩

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

instance $u :: (cpo)\ cpo$
⟨*proof*⟩

12.5 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*⟩

12.6 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*⟩

12.7 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·$x \Rightarrow t$ == *CONST fup*·$(\Lambda\ x.\ t)$·l
$\Lambda(XCONST\ up{\cdot}x).\ t$ == *CONST fup*·$(\Lambda\ x.\ t)$

continuous versions of lemmas for $'a_\perp$

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

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

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

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

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

lifting preserves chain-finiteness

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

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

properties of fup

lemma *fup1* [*simp*]: *fup*·*f*·⊥ = ⊥
⟨*proof*⟩

lemma *fup2* [*simp*]: *fup*·*f*·(*up*·*x*) = *f*·*x*
⟨*proof*⟩

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

12.8 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

13 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports *Finite-Set List Hilbert-Choice*
begin

13.1 The class of countable types

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

lemma *countable-classI*:

fixes $f :: 'a \Rightarrow nat$
assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
shows *OFCLASS*($'a$, *countable-class*)
⟨*proof*⟩

13.2 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 *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*⟩

13.3 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* ∨ *i* < *0* ∨ *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* ⇒ *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*⟩

end

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

theory *Lift*

imports *Discrete Up Cprod 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*))

14.1 Lift as a datatype

lemma *lift-distinct1*: ⊥ ≠ *Def x*
⟨*proof*⟩

lemma *lift-distinct2*: *Def x* ≠ ⊥
⟨*proof*⟩

lemma *Def-inject*: (*Def x* = *Def y*) = (*x* = *y*)
⟨*proof*⟩

lemma *lift-induct*: ⟦*P* ⊥; ⋀*x*. *P* (*Def x*)⟧ ⟹ *P y*
⟨*proof*⟩

rep-datatype *lift*
distinct *lift-distinct1 lift-distinct2*
inject *Def-inject*
induction *lift-induct*

lemma *Def-not-UU*: *Def a* ≠ *UU*
⟨*proof*⟩

⊥ and *Def*

lemma *Lift-exhaust*: *x* = ⊥ ∨ (∃ *y*. *x* = *Def y*)
⟨*proof*⟩

lemma *Lift-cases*: ⟦*x* = ⊥ ⟹ *P*; ∃ *a*. *x* = *Def a* ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

lemma *not-Undef-is-Def*: (*x* ≠ ⊥) = (∃ *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 *def-tac* 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*: $Def\ x \sqsubseteq Def\ y = (x = y)$
⟨*proof*⟩

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

14.2 Lift is flat

lemma *less-lift*: $(x::'a\ lift) \sqsubseteq y = (x = y \vee x = \bot)$
⟨*proof*⟩

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*⟩

14.3 Further operations

definition
flift1 :: $('a \Rightarrow 'b::pcpo) \Rightarrow ('a\ lift \to 'b)$ (**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 = $(FLIFT\ x.\ Def\ (f\ x))$

definition
liftpair :: $'a\ lift \times 'b\ lift \Rightarrow ('a \times 'b)\ lift$ **where**
liftpair x = $csplit\cdot(FLIFT\ x\ y.\ Def\ (x, y))\cdot x$

14.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

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

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

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

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

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

rewrites for *flift1*, *flift2*

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

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

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

14.5 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

15 Tr: The type of lifted booleans

theory *Tr*
imports *Lift*
begin

defaultsort *pcpo*

types
tr = *bool lift*

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

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

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

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*

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 :: $tr \to tr \to tr$ **where**
orelse-def: $tror = (\Lambda\ x\ y.\ If\ x\ then\ TT\ else\ y\ fi)$
abbreviation
orelse-syn :: $tr \Rightarrow tr \Rightarrow tr$ (- *orelse* - [*31*,*30*] *30*) **where**
$x\ orelse\ y == tror{\cdot}x{\cdot}y$

definition
neg :: $tr \to tr$ **where**
$neg = flift2\ Not$

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

translations
$\Lambda\ (CONST\ TT).\ t == CONST\ trifte{\cdot}t{\cdot}\bot$
$\Lambda\ (CONST\ FF).\ t == CONST\ trifte{\cdot}\bot{\cdot}t$

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*⟩

tactic for tr-thms with case split

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

distinctness for type *tr*

lemma *dist-less-tr* [*simp*]:
$\neg\ TT \sqsubseteq \bot\ \neg\ FF \sqsubseteq \bot\ \neg\ TT \sqsubseteq FF\ \neg\ FF \sqsubseteq TT$
⟨*proof*⟩

lemma *dist-eq-tr* [*simp*]:
$TT \neq \bot\ FF \neq \bot\ TT \neq FF\ \bot \neq TT\ \bot \neq FF\ FF \neq TT$
⟨*proof*⟩

lemmas about andalso, orelse, neg and if

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

lemma *andalso-thms* [*simp*]:
$(TT\ andalso\ y) = y$
$(FF\ andalso\ y) = FF$

$(\bot\ andalso\ y) = \bot$
$(y\ andalso\ TT) = y$
$(y\ andalso\ y) = y$
⟨*proof*⟩

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

lemma *neg-thms* [*simp*]:
$neg{\cdot}TT = FF$
$neg{\cdot}FF = TT$
$neg{\cdot}\bot = \bot$
⟨*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*⟩

15.1 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*⟩

15.2 Compactness

lemma *compact-TT* [*simp*]: *compact TT*
⟨*proof*⟩

lemma *compact-FF* [*simp*]: *compact FF*
⟨*proof*⟩

end

16 Ssum: The type of strict sums

theory *Ssum*
imports *Cprod Tr*
begin

defaultsort *pcpo*

16.1 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*)

16.2 Definitions of constructors

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

definition

sinr :: $'b \to ('a \mathbin{++} 'b)$ **where**
sinr = $(\Lambda\ b.\ \mathit{Abs\text{-}Ssum} <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b,\ \bot,\ b>)$

lemma *sinl-Ssum*: $<\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{TT})\cdot a,\ a,\ \bot> \in \mathit{Ssum}$
⟨*proof*⟩

lemma *sinr-Ssum*: $<\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b,\ \bot,\ b> \in \mathit{Ssum}$
⟨*proof*⟩

lemma *sinl-Abs-Ssum*: $\mathit{sinl}\cdot a = \mathit{Abs\text{-}Ssum} <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{TT})\cdot a,\ a,\ \bot>$
⟨*proof*⟩

lemma *sinr-Abs-Ssum*: $\mathit{sinr}\cdot b = \mathit{Abs\text{-}Ssum} <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b,\ \bot,\ b>$
⟨*proof*⟩

lemma *Rep-Ssum-sinl*: $\mathit{Rep\text{-}Ssum}\ (\mathit{sinl}\cdot a) = <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{TT})\cdot a,\ a,\ \bot>$
⟨*proof*⟩

lemma *Rep-Ssum-sinr*: $\mathit{Rep\text{-}Ssum}\ (\mathit{sinr}\cdot b) = <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b,\ \bot,\ b>$
⟨*proof*⟩

16.3 Properties of *sinl* and *sinr*

Ordering

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

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

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

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

Equality

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

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

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

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

16.4 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*⟩

16.5 Case analysis combinator

definition
sscase :: $('a \to 'c) \to ('b \to 'c) \to ('a \mathbin{++} 'b) \to 'c$ **where**
sscase = $(\Lambda\ f\ g\ s.\ (\Lambda\langle t,\ x,\ y\rangle.\ If\ t\ then\ f{\cdot}x\ else\ g{\cdot}y\ fi){\cdot}(Rep\text{-}Ssum\ s))$

translations
case s of XCONST sinl$\cdot x \Rightarrow t1 \mid XCONST\ sinr{\cdot}y \Rightarrow t2 == CONST\ sscase{\cdot}(\Lambda\ x.\ t1){\cdot}(\Lambda\ y.\ t2){\cdot}s$

translations
$\Lambda(XCONST\ sinl{\cdot}x).\ t == CONST\ sscase{\cdot}(\Lambda\ x.\ t){\cdot}\bot$
$\Lambda(XCONST\ sinr{\cdot}y).\ t == CONST\ sscase{\cdot}\bot{\cdot}(\Lambda\ y.\ t)$

lemma *beta-sscase*:
$sscase{\cdot}f{\cdot}g{\cdot}s = (\Lambda\langle t,\ x,\ y\rangle.\ If\ t\ then\ f{\cdot}x\ else\ g{\cdot}y\ fi){\cdot}(Rep\text{-}Ssum\ s)$
⟨*proof*⟩

lemma *sscase1* [*simp*]: $sscase{\cdot}f{\cdot}g{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *sscase2* [*simp*]: $x \neq \bot \Longrightarrow sscase{\cdot}f{\cdot}g{\cdot}(sinl{\cdot}x) = f{\cdot}x$
⟨*proof*⟩

lemma *sscase3* [*simp*]: $y \neq \bot \Longrightarrow sscase{\cdot}f{\cdot}g{\cdot}(sinr{\cdot}y) = g{\cdot}y$
⟨*proof*⟩

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

16.6 Strict sum preserves flatness

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

16.7 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

17 One: The unit domain

theory *One*
imports *Lift*
begin

types *one* = *unit lift*
translations
one <= (*type*) *unit lift*

constdefs
ONE :: *one*
ONE == *Def* ()

Exhaustion and Elimination for type *one*

lemma *Exh-one*: *t* = ⊥ ∨ *t* = *ONE*
⟨*proof*⟩

lemma *oneE*: $[\![p = \bot \Longrightarrow Q;\ p = ONE \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *dist-less-one* [*simp*]: $\neg\ ONE \sqsubseteq \bot$
⟨*proof*⟩

lemma *dist-eq-one* [*simp*]: $ONE \neq \bot\ \bot \neq ONE$
⟨*proof*⟩

lemma *compact-ONE* [*simp*]: *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 CONST ONE ⇒ *t* == *CONST one-when*·*t*·*x*
Λ (*CONST 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

18 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun Cprod Adm*
begin

defaultsort *pcpo*

18.1 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* $0 \cdot F \cdot x = x$
⟨*proof*⟩

lemma *iterate-Suc* [*simp*]: *iterate* $(Suc\ n) \cdot F \cdot x = F \cdot (iterate\ n \cdot F \cdot x)$
⟨*proof*⟩

declare *iterate.simps* [*simp del*]

lemma *iterate-Suc2*: *iterate* $(Suc\ n) \cdot F \cdot x = iterate\ n \cdot F \cdot (F \cdot x)$
⟨*proof*⟩

The sequence of function iterations is a chain. This property is essential since monotonicity of iterate makes no sense.

lemma *chain-iterate2*: $x \sqsubseteq F \cdot x \Longrightarrow chain\ (\lambda i.\ iterate\ i \cdot F \cdot x)$
⟨*proof*⟩

lemma *chain-iterate* [*simp*]: $chain\ (\lambda i.\ iterate\ i \cdot F \cdot \bot)$
⟨*proof*⟩

18.2 Least fixed point operator

definition
$fix :: ('a \to 'a) \to 'a$ **where**
$fix = (\Lambda\ F.\ \bigsqcup i.\ iterate\ i \cdot F \cdot \bot)$

Binder syntax for *fix*

syntax
$-FIX :: ['a, 'a] \Rightarrow 'a\ ((3FIX\ -./\ -)\ [1000,\ 10]\ 10)$

syntax (*xsymbols*)
$-FIX :: ['a, 'a] \Rightarrow 'a\ ((3\mu\ -./\ -)\ [1000,\ 10]\ 10)$

translations
$\mu\ x.\ t == CONST\ fix \cdot (\Lambda\ x.\ t)$

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*: $F{\cdot}x = x \Longrightarrow \mathit{fix}{\cdot}F \sqsubseteq x$
⟨*proof*⟩

lemma *fix-eqI*: $[\![F{\cdot}x = x;\ \forall z.\ F{\cdot}z = z \longrightarrow x \sqsubseteq z]\!] \Longrightarrow x = \mathit{fix}{\cdot}F$
⟨*proof*⟩

lemma *fix-eq2*: $f \equiv \mathit{fix}{\cdot}F \Longrightarrow f = F{\cdot}f$
⟨*proof*⟩

lemma *fix-eq3*: $f \equiv \mathit{fix}{\cdot}F \Longrightarrow f{\cdot}x = F{\cdot}f{\cdot}x$
⟨*proof*⟩

lemma *fix-eq4*: $f = \mathit{fix}{\cdot}F \Longrightarrow f = F{\cdot}f$
⟨*proof*⟩

lemma *fix-eq5*: $f = \mathit{fix}{\cdot}F \Longrightarrow f{\cdot}x = F{\cdot}f{\cdot}x$
⟨*proof*⟩

strictness of *fix*

lemma *fix-defined-iff*: $(\mathit{fix}{\cdot}F = \bot) = (F{\cdot}\bot = \bot)$
⟨*proof*⟩

lemma *fix-strict*: $F{\cdot}\bot = \bot \Longrightarrow \mathit{fix}{\cdot}F = \bot$
⟨*proof*⟩

lemma *fix-defined*: $F{\cdot}\bot \neq \bot \Longrightarrow \mathit{fix}{\cdot}F \neq \bot$
⟨*proof*⟩

fix applied to identity and constant functions

lemma *fix-id*: $(\mu\ x.\ x) = \bot$
⟨*proof*⟩

lemma *fix-const*: $(\mu\ x.\ c) = c$
⟨*proof*⟩

18.3 Fixed point induction

lemma *fix-ind*: $[\![\mathit{adm}\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F{\cdot}x)]\!] \Longrightarrow P\ (\mathit{fix}{\cdot}F)$
⟨*proof*⟩

lemma *def-fix-ind*:
$[\![f \equiv \mathit{fix}{\cdot}F;\ \mathit{adm}\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F{\cdot}x)]\!] \Longrightarrow P\ f$
⟨*proof*⟩

18.4 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*$\cdot(\Lambda$ *xs.* $\langle a,e,es\rangle)$
Letrec xs = a in e == *CONST CLetrec*$\cdot(\Lambda$ *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*:
$fix{\cdot}(F::'a \times 'b \to 'a \times 'b) =$
$\langle \mu\ x.\ cfst{\cdot}(F{\cdot}\langle x,\ \mu\ y.\ csnd{\cdot}(F{\cdot}\langle x,\ y\rangle)\rangle),$
$\mu\ y.\ csnd{\cdot}(F{\cdot}\langle \mu\ x.\ cfst{\cdot}(F{\cdot}\langle x,\ \mu\ y.\ csnd{\cdot}(F{\cdot}\langle x,\ y\rangle)\rangle),\ y\rangle)\rangle$
(**is** $fix{\cdot}F = \langle ?x,\ ?y\rangle$)
⟨*proof*⟩

18.5 Weak admissibility

definition
admw :: $('a \Rightarrow bool) \Rightarrow bool$ **where**
admw $P = (\forall F.\ (\forall n.\ P\ (iterate\ n{\cdot}F{\cdot}\bot)) \longrightarrow P\ (\bigsqcup i.\ iterate\ i{\cdot}F{\cdot}\bot))$

an admissible formula is also weak admissible

lemma *adm-impl-admw*: $adm\ P \Longrightarrow admw\ P$
⟨*proof*⟩

computational induction for weak admissible formulae

lemma *wfix-ind*: $[\![admw\ P; \forall n.\ P\ (iterate\ n{\cdot}F{\cdot}\bot)]\!] \Longrightarrow P\ (fix{\cdot}F)$
⟨*proof*⟩

lemma *def-wfix-ind*:
$[\![f \equiv fix{\cdot}F;\ admw\ P; \forall n.\ P\ (iterate\ n{\cdot}F{\cdot}\bot)]\!] \Longrightarrow P\ f$
⟨*proof*⟩

end

19 Fixrec: Package for defining recursive functions in HOLCF

theory *Fixrec*
imports *Sprod Ssum Up One Tr Fix*
uses (*Tools/fixrec-package.ML*)
begin

19.1 Maybe monad type

defaultsort *cpo*

pcpodef (**open**) *'a maybe* = *UNIV*::(*one* ++ *'a u*) *set*
⟨*proof*⟩

constdefs
fail :: *'a maybe*
fail ≡ *Abs-maybe* (*sinl*·*ONE*)

constdefs
return :: *'a* → *'a maybe* **where**
return ≡ Λ *x*. *Abs-maybe* (*sinr*·(*up*·*x*))

definition
maybe-when :: *'b* → (*'a* → *'b*) → *'a maybe* → *'b*::*pcpo* **where**
maybe-when = (Λ *f r m*. *sscase*·(Λ *x*. *f*)·(*fup*·*r*)·(*Rep-maybe m*))

lemma *maybeE*:
⟦*p* = ⊥ ⟹ *Q*; *p* = *fail* ⟹ *Q*; ⋀*x*. *p* = *return*·*x* ⟹ *Q*⟧ ⟹ *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 fail ⇒ *t1* | *return*·*x* ⇒ *t2* == *CONST maybe-when*·*t1*·(Λ *x*. *t2*)·*m*

19.1.1 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*⟩

19.1.2 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*⟩

19.1.3 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*⟩

19.1.4 Fatbar combinator

definition
fatbar :: (*'a* → *'b maybe*) → (*'a* → *'b maybe*) → (*'a* → *'b maybe*) **where**
fatbar = (Λ *a b x*. *a*·*x* +++ *b*·*x*)

abbreviation
fatbar-syn :: [*'a* → *'b maybe*, *'a* → *'b maybe*] ⇒ *'a* → *'b maybe* (**infixr** ‖ *60*)
where
m1 ‖ *m2* == *fatbar*·*m1*·*m2*

lemma *fatbar1*: *m*·*x* = ⊥ ⟹ (*m* ‖ *ms*)·*x* = ⊥
⟨*proof*⟩

lemma *fatbar2*: *m*·*x* = *fail* ⟹ (*m* ‖ *ms*)·*x* = *ms*·*x*
⟨*proof*⟩

lemma *fatbar3*: *m*·*x* = *return*·*y* ⟹ (*m* ‖ *ms*)·*x* = *return*·*y*
⟨*proof*⟩

lemmas *fatbar-simps* = *fatbar1 fatbar2 fatbar3*

lemma *run-fatbar1*: *m*·*x* = ⊥ ⟹ *run*·((*m* ‖ *ms*)·*x*) = ⊥
⟨*proof*⟩

lemma *run-fatbar2*: $m{\cdot}x = \mathit{fail} \Longrightarrow \mathit{run}{\cdot}((m \parallel \mathit{ms}){\cdot}x) = \mathit{run}{\cdot}(\mathit{ms}{\cdot}x)$
⟨*proof*⟩

lemma *run-fatbar3*: $m{\cdot}x = \mathit{return}{\cdot}y \Longrightarrow \mathit{run}{\cdot}((m \parallel \mathit{ms}){\cdot}x) = y$
⟨*proof*⟩

lemmas *run-fatbar-simps* [*simp*] = *run-fatbar1 run-fatbar2 run-fatbar3*

19.2 Case branch combinator

constdefs
branch :: $('a \to 'b\ \mathit{maybe}) \Rightarrow ('b \to 'c) \to ('a \to 'c\ \mathit{maybe})$
$\mathit{branch}\ p \equiv \Lambda\ r\ x.\ \mathit{bind}{\cdot}(p{\cdot}x){\cdot}(\Lambda\ y.\ \mathit{return}{\cdot}(r{\cdot}y))$

lemma *branch-rews*:
$p{\cdot}x = \bot \Longrightarrow \mathit{branch}\ p{\cdot}r{\cdot}x = \bot$
$p{\cdot}x = \mathit{fail} \Longrightarrow \mathit{branch}\ p{\cdot}r{\cdot}x = \mathit{fail}$
$p{\cdot}x = \mathit{return}{\cdot}y \Longrightarrow \mathit{branch}\ p{\cdot}r{\cdot}x = \mathit{return}{\cdot}(r{\cdot}y)$
⟨*proof*⟩

lemma *branch-return* [*simp*]: $\mathit{branch}\ \mathit{return}{\cdot}r{\cdot}x = \mathit{return}{\cdot}(r{\cdot}x)$
⟨*proof*⟩

19.3 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.run*·(*ms*·*x*)
-*Case2 m ms* == $m \parallel \mathit{ms}$

Parsing Case expressions

syntax
-*pat* :: '*a*
-*var* :: '*a*
-*noargs* :: '*a*

translations

-Case1 p r => CONST branch (-pat p)·(-var p r)
-var (-args x y) r => CONST csplit·(-var x (-var y r))
-var -noargs r => CONST unit-when·r

⟨*ML*⟩

Printing Case expressions

syntax
-match :: 'a

⟨*ML*⟩

translations
x <= -match Fixrec.return (-var x)

19.4 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
-var (*XCONST cpair*·*x*·*y*) *r* => *-var* (*-args x y*) *r*
-var (*XCONST spair*·*x*·*y*) *r* => *-var* (*-args x y*) *r*
-var (*XCONST sinl*·*x*) *r* => *-var x r*
-var (*XCONST sinr*·*x*) *r* => *-var x r*
-var (*XCONST up*·*x*) *r* => *-var x r*
-var (*XCONST TT*) *r* => *-var -noargs r*
-var (*XCONST FF*) *r* => *-var -noargs r*
-var (*XCONST ONE*) *r* => *-var -noargs r*

translations
-var (*CONST cpair*·*x*·*y*) *r* => *-var* (*-args x y*) *r*
-var (*CONST spair*·*x*·*y*) *r* => *-var* (*-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*⟩

19.5 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·*(*run·*(*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
-var - *r* => *-var -noargs r*
-var (*-as-pat x y*) *r* => *-var* (*-args x y*) *r*
-var (*-lazy-pat x*) *r* => *-var x r*

Print translations

translations
- <= *-match* (*CONST wild-pat*) *-noargs*
-as-pat x (*-match p v*) <= *-match* (*CONST as-pat p*) (*-args* (*-var 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.run 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*⟩

19.6 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*⟩

19.7 Mutual recursion

The following rules are used to prove unfolding theorems from fixed-point definitions of mutually recursive functions.

lemma *cpair-equalI*: $[\![x \equiv cfst{\cdot}p;\ y \equiv csnd{\cdot}p]\!] \Longrightarrow <x,y> \equiv p$
⟨*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*⟩

19.8 Initializing the fixrec package

⟨*ML*⟩

hide (**open**) *const return bind fail run*

end

20 Domain: Domain package

theory *Domain*
imports *Ssum Sprod Up One Tr Fixrec*
begin

defaultsort *pcpo*

20.1 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

20.2 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

21 SetPcpo: Set as a pointed cpo

theory *SetPcpo*
imports *Adm*
begin

instantiation *bool* :: *po*
begin

definition
less-bool-def: $(op \sqsubseteq) = (op \longrightarrow)$

instance
⟨*proof*⟩

end

lemma *less-set-eq*: $(op \sqsubseteq) = (op \subseteq)$
⟨*proof*⟩

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

lemma *Union-is-lub*: $A <<| Union\ A$
⟨*proof*⟩

instance *bool* :: *cpo* ⟨*proof*⟩

lemma *lub-eq-Union*: *lub* = *Union*
⟨*proof*⟩

instance *bool* :: *pcpo*
⟨*proof*⟩

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

lemmas *set-cpo-simps* = *less-set-eq lub-eq-Union UU-eq-empty*

21.1 Admissibility of set predicates

lemma *adm-nonempty*: *adm* $(\lambda A.\ \exists\, x.\ x \in A)$

⟨*proof*⟩

lemma *adm-in*: *adm* (λ*A*. *x* ∈ *A*)
⟨*proof*⟩

lemma *adm-not-in*: *adm* (λ*A*. *x* ∉ *A*)
⟨*proof*⟩

lemma *adm-Ball*: (⋀*x*. *adm* (λ*A*. *P A x*)) ⟹ *adm* (λ*A*. ∀ *x*∈*A*. *P A x*)
⟨*proof*⟩

lemma *adm-Bex*: *adm* (λ*A*. *Bex A P*)
⟨*proof*⟩

lemma *adm-subset*: *adm* (λ*A*. *A* ⊆ *B*)
⟨*proof*⟩

lemma *adm-superset*: *adm* (λ*A*. *B* ⊆ *A*)
⟨*proof*⟩

lemmas *adm-set-lemmas* =
adm-nonempty adm-in adm-not-in adm-Bex adm-Ball adm-subset adm-superset

21.2 Compactness

lemma *compact-empty*: *compact* {}
⟨*proof*⟩

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

lemma *finite-imp-compact*: *finite A* ⟹ *compact A*
⟨*proof*⟩

end

22 CompactBasis: Compact bases of domains

theory *CompactBasis*
imports *Bifinite SetPcpo*
begin

22.1 Ideals over a preorder

context *preorder*
begin

definition

ideal :: *'a set* ⇒ *bool* **where**
ideal A = ((∃ *x*. *x* ∈ *A*) ∧ (∀ *x*∈*A*. ∀ *y*∈*A*. ∃ *z*∈*A*. *x* ⊑ *z* ∧ *y* ⊑ *z*) ∧
(∀ *x y*. *x* ⊑ *y* ⟶ *y* ∈ *A* ⟶ *x* ∈ *A*))

lemma *idealI*:
assumes ∃ *x*. *x* ∈ *A*
assumes ⋀*x y*. ⟦*x* ∈ *A*; *y* ∈ *A*⟧ ⟹ ∃ *z*∈*A*. *x* ⊑ *z* ∧ *y* ⊑ *z*
assumes ⋀*x y*. ⟦*x* ⊑ *y*; *y* ∈ *A*⟧ ⟹ *x* ∈ *A*
shows *ideal A*
⟨*proof*⟩

lemma *idealD1*:
ideal A ⟹ ∃ *x*. *x* ∈ *A*
⟨*proof*⟩

lemma *idealD2*:
⟦*ideal A*; *x* ∈ *A*; *y* ∈ *A*⟧ ⟹ ∃ *z*∈*A*. *x* ⊑ *z* ∧ *y* ⊑ *z*
⟨*proof*⟩

lemma *idealD3*:
⟦*ideal A*; *x* ⊑ *y*; *y* ∈ *A*⟧ ⟹ *x* ∈ *A*
⟨*proof*⟩

lemma *ideal-directed-finite*:
assumes *A*: *ideal A*
shows ⟦*finite U*; *U* ⊆ *A*⟧ ⟹ ∃ *z*∈*A*. ∀ *x*∈*U*. *x* ⊑ *z*
⟨*proof*⟩

lemma *ideal-principal*: *ideal* {*x*. *x* ⊑ *z*}
⟨*proof*⟩

lemma *directed-image-ideal*:
assumes *A*: *ideal A*
assumes *f*: ⋀*x y*. *x* ⊑ *y* ⟹ *f x* ⊑ *f y*
shows *directed* (*f* ‘ *A*)
⟨*proof*⟩

lemma *adm-ideal*: *adm* (λ*A*. *ideal A*)
⟨*proof*⟩

lemma *lub-image-principal*:
assumes *f*: ⋀*x y*. *x* ⊑ *y* ⟹ *f x* ⊑ *f y*
shows (⨆ *x*∈{*x*. *x* ⊑ *y*}. *f x*) = *f y*
⟨*proof*⟩

end

22.2 Defining functions in terms of basis elements

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*⟩

locale *basis-take* = *preorder r* +
  fixes *take* :: *nat* ⇒ *'a*::*type* ⇒ *'a*
  assumes *take-less*: *r* (*take n a*) *a*
  assumes *take-take*: *take n* (*take n a*) = *take n a*
  assumes *take-mono*: *r a b* ⟹ *r* (*take n a*) (*take n b*)
  assumes *take-chain*: *r* (*take n a*) (*take* (*Suc n*) *a*)
  assumes *finite-range-take*: *finite* (*range* (*take n*))
  assumes *take-covers*: ∃ *n*. *take n a* = *a*

locale *ideal-completion* = *basis-take r* +
  fixes *principal* :: *'a*::*type* ⇒ *'b*::*cpo*
  fixes *rep* :: *'b*::*cpo* ⇒ *'a*::*type set*
  assumes *ideal-rep*: ⋀*x*. *preorder.ideal r* (*rep x*)
  assumes *cont-rep*: *cont rep*
  assumes *rep-principal*: ⋀*a*. *rep* (*principal a*) = {*b*. *r 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 *basis-fun-lemma0*:
  fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
  assumes *f-mono*: ⋀*a b*. *r a b* ⟹ *f a* ⊑ *f b*
  shows ∃ *u*. *f* ‘ *take i* ‘ *rep x* <<| *u*
⟨*proof*⟩

lemma *basis-fun-lemma1*:
  fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
  assumes *f-mono*: ⋀*a b*. *r a b* ⟹ *f a* ⊑ *f b*
  shows *chain* (λ*i*. *lub* (*f* ‘ *take i* ‘ *rep x*))

⟨*proof*⟩

lemma *basis-fun-lemma2*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ r\ a\ b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $f\ `\ rep\ x <<|\ (\bigsqcup i.\ lub\ (f\ `\ take\ i\ `\ rep\ x))$
⟨*proof*⟩

lemma *basis-fun-lemma*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ r\ a\ b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $\exists u.\ f\ `\ rep\ x <<|\ u$
⟨*proof*⟩

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

lemma *rep-contlub*:
$chain\ Y \Longrightarrow rep\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ rep\ (Y\ i))$
⟨*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*: $principal\ a \sqsubseteq principal\ b \longleftrightarrow r\ a\ b$
⟨*proof*⟩

lemma *principal-eq-iff*: $principal\ a = principal\ b \longleftrightarrow r\ a\ b \wedge r\ b\ a$
⟨*proof*⟩

lemma *repD*: $a \in rep\ x \Longrightarrow principal\ a \sqsubseteq x$
⟨*proof*⟩

lemma *principal-mono*: $r\ a\ b \Longrightarrow principal\ a \sqsubseteq principal\ b$
⟨*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*⟩

definition
basis-fun :: (*'a*::*type* ⇒ *'c*::*cpo*) ⇒ *'b* → *'c* **where**
basis-fun = (λ*f*. (Λ *x*. *lub* (*f* ‘ *rep x*)))

lemma *basis-fun-beta*:
fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
assumes *f-mono*: ⋀*a b*. *r 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*. *r a b* ⟹ *f a* ⊑ *f b*
shows *basis-fun f*·(*principal a*) = *f a*
⟨*proof*⟩

lemma *basis-fun-mono*:
assumes *f-mono*: ⋀*a b*. *r a b* ⟹ *f a* ⊑ *f b*
assumes *g-mono*: ⋀*a b*. *r a b* ⟹ *g a* ⊑ *g b*
assumes *less*: ⋀*a*. *f a* ⊑ *g a*
shows *basis-fun f* ⊑ *basis-fun g*
⟨*proof*⟩

lemma *compact-principal*: *compact* (*principal a*)
⟨*proof*⟩

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*: (⨆ *i*. *completion-approx i*·*x*) = *x*
⟨*proof*⟩

lemma *completion-approx-eq-principal*:
∃ *a*∈*rep x*. *completion-approx i*·*x* = *principal* (*take i a*)
⟨*proof*⟩

lemma *completion-approx-idem*:
completion-approx i·(completion-approx i·x) = completion-approx i·x
⟨*proof*⟩

lemma *finite-fixes-completion-approx*:
finite {x. completion-approx i·x = x} (**is** *finite ?S*)
⟨*proof*⟩

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes *P*: ⋀*a*. *P* (*principal a*)
shows *P x*
⟨*proof*⟩

end

22.3 Compact bases of bifinite domains

defaultsort *profinite*

typedef (**open**) *'a compact-basis = {x::'a::profinite. compact x}*
⟨*proof*⟩

lemma *compact-Rep-compact-basis* [*simp*]: *compact* (*Rep-compact-basis a*)
⟨*proof*⟩

lemma *Rep-Abs-compact-basis-approx* [*simp*]:
Rep-compact-basis (*Abs-compact-basis* (*approx n·x*)) = *approx n·x*
⟨*proof*⟩

lemma *compact-imp-Rep-compact-basis*:
compact x ⟹ ∃ *y*. *x = Rep-compact-basis y*
⟨*proof*⟩

instantiation *compact-basis* :: (*profinite*) *sq-ord*
begin

definition
compact-le-def:
(*op* ⊑) ≡ (λ*x y*. *Rep-compact-basis x* ⊑ *Rep-compact-basis y*)

instance ⟨*proof*⟩

end

instance *compact-basis* :: (*profinite*) *po*
⟨*proof*⟩

minimal compact element

definition
compact-bot :: *'a*::*bifinite compact-basis* **where**
compact-bot = *Abs-compact-basis* ⊥

lemma *Rep-compact-bot*: *Rep-compact-basis compact-bot* = ⊥
⟨*proof*⟩

lemma *compact-minimal* [*simp*]: *compact-bot* ⊑ *a*
⟨*proof*⟩

compacts

definition
compacts :: *'a* ⇒ *'a compact-basis set* **where**
compacts = (λ*x*. {*a*. *Rep-compact-basis a* ⊑ *x*})

lemma *ideal-compacts*: *preorder.ideal sq-le* (*compacts w*)
⟨*proof*⟩

lemma *compacts-Rep-compact-basis*:
compacts (*Rep-compact-basis b*) = {*a*. *a* ⊑ *b*}
⟨*proof*⟩

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

lemma *compacts-lessD*: *compacts x* ⊆ *compacts y* ⟹ *x* ⊑ *y*
⟨*proof*⟩

lemma *compacts-mono*: *x* ⊑ *y* ⟹ *compacts x* ⊆ *compacts y*
⟨*proof*⟩

lemma *less-compact-basis-iff*: (*x* ⊑ *y*) = (*compacts x* ⊆ *compacts y*)
⟨*proof*⟩

lemma *compact-basis-induct*:
⟦*adm P*; ⋀*a*. *P* (*Rep-compact-basis a*)⟧ ⟹ *P x*
⟨*proof*⟩

approximation on compact bases

definition
compact-approx :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis* **where**
compact-approx = (λ*n a*. *Abs-compact-basis* (*approx n*·(*Rep-compact-basis a*)))

lemma *Rep-compact-approx*:
Rep-compact-basis (*compact-approx n a*) = *approx n*·(*Rep-compact-basis a*)
⟨*proof*⟩

lemmas *approx-Rep-compact-basis* = *Rep-compact-approx* [*symmetric*]

lemma *compact-approx-le*: *compact-approx n a* $\sqsubseteq$ *a*
⟨*proof*⟩

lemma *compact-approx-mono1*:
i $\leq$ *j* $\Longrightarrow$ *compact-approx i a* $\sqsubseteq$ *compact-approx j a*
⟨*proof*⟩

lemma *compact-approx-mono*:
a $\sqsubseteq$ *b* $\Longrightarrow$ *compact-approx n a* $\sqsubseteq$ *compact-approx n b*
⟨*proof*⟩

lemma *ex-compact-approx-eq*: $\exists$ *n. compact-approx n a* = *a*
⟨*proof*⟩

lemma *compact-approx-idem*:
compact-approx n (*compact-approx n a*) = *compact-approx n a*
⟨*proof*⟩

lemma *finite-fixes-compact-approx*: *finite* {*a. compact-approx n a* = *a*}
⟨*proof*⟩

lemma *finite-range-compact-approx*: *finite* (*range* (*compact-approx n*))
⟨*proof*⟩

interpretation *compact-basis*:
ideal-completion [*sq-le compact-approx Rep-compact-basis compacts*]
⟨*proof*⟩

22.4 A compact basis for powerdomains

typedef *'a pd-basis* =
{*S*::*'a*::*profinite compact-basis set. finite S* $\wedge$ *S* $\neq$ {}}
⟨*proof*⟩

lemma *finite-Rep-pd-basis* [*simp*]: *finite* (*Rep-pd-basis u*)
⟨*proof*⟩

lemma *Rep-pd-basis-nonempty* [*simp*]: *Rep-pd-basis u* $\neq$ {}
⟨*proof*⟩

unit and plus

definition
PDUnit :: *'a compact-basis* $\Rightarrow$ *'a pd-basis* **where**
PDUnit = (λ*x. Abs-pd-basis* {*x*})

definition
PDPlus :: *'a pd-basis* $\Rightarrow$ *'a pd-basis* $\Rightarrow$ *'a pd-basis* **where**
PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* $\cup$ *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*: ⋀*a t*. *P t* ⟹ *P* (*PDPlus* (*PDUnit a*) *t*)
shows *P x*
⟨*proof*⟩

lemma *pd-basis-induct*:
assumes *PDUnit*: ⋀*a*. *P* (*PDUnit a*)
assumes *PDPlus*: ⋀*t u*. ⟦*P t*; *P u*⟧ ⟹ *P* (*PDPlus t u*)
shows *P x*
⟨*proof*⟩

fold-pd

definition
fold-pd ::
('*a compact-basis* ⇒ '*b*::*type*) ⇒ ('*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a pd-basis* ⇒ '*b*
where *fold-pd g f t* = *fold1 f* (*g* ‘ *Rep-pd-basis t*)

lemma *fold-pd-PDUnit*:
includes *ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDUnit x*) = *g x*
⟨*proof*⟩

lemma *fold-pd-PDPlus*:
includes *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*⟩

approx-pd

definition
approx-pd :: *nat* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
approx-pd n = (λ*t*. *Abs-pd-basis* (*compact-approx n* ‘ *Rep-pd-basis t*))

lemma *Rep-approx-pd*:
Rep-pd-basis (*approx-pd n t*) = *compact-approx n* ‘ *Rep-pd-basis t*
⟨*proof*⟩

lemma *approx-pd-simps* [*simp*]:
approx-pd n (*PDUnit a*) = *PDUnit* (*compact-approx n a*)
approx-pd n (*PDPlus t u*) = *PDPlus* (*approx-pd n t*) (*approx-pd n u*)
⟨*proof*⟩

lemma *approx-pd-idem*: *approx-pd n* (*approx-pd n t*) = *approx-pd n t*
⟨*proof*⟩

lemma *range-image-f*: *range* (*image f*) = *Pow* (*range f*)
⟨*proof*⟩

lemma *finite-range-approx-pd*: *finite* (*range* (*approx-pd n*))
⟨*proof*⟩

lemma *ex-approx-pd-eq*: ∃ *n*. *approx-pd n t* = *t*
⟨*proof*⟩

end

23 UpperPD: Upper powerdomain

theory *UpperPD*
imports *CompactBasis*
begin

23.1 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*]: $PDUnit\ compact\text{-}bot \leq\sharp\ t$
⟨*proof*⟩

lemma *PDUnit-upper-mono*: $x \sqsubseteq y \Longrightarrow PDUnit\ x \leq\sharp\ PDUnit\ y$
⟨*proof*⟩

lemma *PDPlus-upper-mono*: $[\![s \leq\sharp\ t;\ u \leq\sharp\ v]\!] \Longrightarrow PDPlus\ s\ u \leq\sharp\ PDPlus\ t\ v$
⟨*proof*⟩

lemma *PDPlus-upper-less*: $PDPlus\ t\ u \leq\sharp\ t$
⟨*proof*⟩

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
$(PDUnit\ a \leq\sharp\ PDUnit\ b) = a \sqsubseteq b$
⟨*proof*⟩

lemma *upper-le-PDPlus-PDUnit-iff*:
$(PDPlus\ t\ u \leq\sharp\ PDUnit\ a) = (t \leq\sharp\ PDUnit\ a \vee u \leq\sharp\ PDUnit\ a)$
⟨*proof*⟩

lemma *upper-le-PDPlus-iff*: $(t \leq\sharp\ PDPlus\ u\ v) = (t \leq\sharp\ u \wedge t \leq\sharp\ v)$
⟨*proof*⟩

lemma *upper-le-induct* [*induct set*: *upper-le*]:
assumes *le*: $t \leq\sharp\ u$
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 *approx-pd-upper-mono1*:
$i \leq j \Longrightarrow approx\text{-}pd\ i\ t \leq\sharp\ approx\text{-}pd\ j\ t$
⟨*proof*⟩

lemma *approx-pd-upper-le*: $approx\text{-}pd\ i\ t \leq\sharp\ t$
⟨*proof*⟩

lemma *approx-pd-upper-mono*:
$t \leq\sharp\ u \Longrightarrow approx\text{-}pd\ n\ t \leq\sharp\ approx\text{-}pd\ n\ u$
⟨*proof*⟩

23.2 Type definition

cpodef (**open**) $'a\ upper\text{-}pd =$
$\{S::'a::profinite\ pd\text{-}basis\ set.\ upper\text{-}le.ideal\ S\}$
⟨*proof*⟩

lemma *ideal-Rep-upper-pd*: $upper\text{-}le.ideal\ (Rep\text{-}upper\text{-}pd\ x)$

⟨*proof*⟩

definition
upper-principal :: *'a pd-basis* ⇒ *'a upper-pd* **where**
upper-principal t = *Abs-upper-pd* {*u*. *u* ≤♯ *t*}

lemma *Rep-upper-principal*:
Rep-upper-pd (*upper-principal t*) = {*u*. *u* ≤♯ *t*}
⟨*proof*⟩

interpretation *upper-pd*:
ideal-completion [*upper-le approx-pd upper-principal Rep-upper-pd*]
⟨*proof*⟩

lemma *upper-principal-less-iff* [*simp*]:
upper-principal t ⊑ *upper-principal u* ⟷ *t* ≤♯ *u*
⟨*proof*⟩

lemma *upper-principal-eq-iff*:
upper-principal t = *upper-principal u* ⟷ *t* ≤♯ *u* ∧ *u* ≤♯ *t*
⟨*proof*⟩

lemma *upper-principal-mono*:
t ≤♯ *u* ⟹ *upper-principal t* ⊑ *upper-principal u*
⟨*proof*⟩

lemma *compact-upper-principal*: *compact* (*upper-principal t*)
⟨*proof*⟩

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) ⊑ *ys*
⟨*proof*⟩

instance *upper-pd* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-upper-pd-pcpo*: ⊥ = *upper-principal* (*PDUnit compact-bot*)
⟨*proof*⟩

23.3 Approximation

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* (*approx-pd n t*)
⟨*proof*⟩

lemma *approx-eq-upper-principal*:
∃ *t*∈*Rep-upper-pd xs*. *approx n·xs* = *upper-principal* (*approx-pd n t*)
⟨*proof*⟩

lemma *compact-imp-upper-principal*:
compact xs ⟹ ∃ *t*. *xs* = *upper-principal t*
⟨*proof*⟩

lemma *upper-principal-induct*:
⟦*adm P*; ⋀*t*. *P* (*upper-principal t*)⟧ ⟹ *P xs*
⟨*proof*⟩

lemma *upper-principal-induct2*:
⟦⋀*ys*. *adm* (λ*xs*. *P xs ys*); ⋀*xs*. *adm* (λ*ys*. *P xs ys*);
⋀*t u*. *P* (*upper-principal t*) (*upper-principal u*)⟧ ⟹ *P xs ys*
⟨*proof*⟩

23.4 Monadic unit and plus

definition
upper-unit :: *'a* → *'a upper-pd* **where**
upper-unit = *compact-basis.basis-fun* (λ*a*. *upper-principal* (*PDUnit a*))

definition
upper-plus :: *'a upper-pd* → *'a upper-pd* → *'a upper-pd* **where**
upper-plus = *upper-pd.basis-fun* (λ*t*. *upper-pd.basis-fun* (λ*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*]:

$\{Rep\text{-}compact\text{-}basis\ a\}\sharp = upper\text{-}principal\ (PDUnit\ a)$
⟨*proof*⟩

lemma *upper-plus-principal* [*simp*]:
$upper\text{-}principal\ t\ +\sharp\ upper\text{-}principal\ u = upper\text{-}principal\ (PDPlus\ t\ u)$
⟨*proof*⟩

lemma *approx-upper-unit* [*simp*]:
$approx\ n{\cdot}\{x\}\sharp = \{approx\ n{\cdot}x\}\sharp$
⟨*proof*⟩

lemma *approx-upper-plus* [*simp*]:
$approx\ n{\cdot}(xs\ +\sharp\ ys) = (approx\ n{\cdot}xs)\ +\sharp\ (approx\ n{\cdot}ys)$
⟨*proof*⟩

lemma *upper-plus-assoc*: $(xs\ +\sharp\ ys)\ +\sharp\ zs = xs\ +\sharp\ (ys\ +\sharp\ zs)$
⟨*proof*⟩

lemma *upper-plus-commute*: $xs\ +\sharp\ ys = ys\ +\sharp\ xs$
⟨*proof*⟩

lemma *upper-plus-absorb*: $xs\ +\sharp\ xs = xs$
⟨*proof*⟩

interpretation *aci-upper-plus*: *ab-semigroup-idem-mult* [*op* $+\sharp$]
⟨*proof*⟩

lemma *upper-plus-left-commute*: $xs\ +\sharp\ (ys\ +\sharp\ zs) = ys\ +\sharp\ (xs\ +\sharp\ zs)$
⟨*proof*⟩

lemma *upper-plus-left-absorb*: $xs\ +\sharp\ (xs\ +\sharp\ ys) = xs\ +\sharp\ ys$
⟨*proof*⟩

lemmas *upper-plus-aci* = *aci-upper-plus.mult-ac-idem*

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* $ys]\!] \Longrightarrow$ *compact* $(xs +\sharp ys)$
⟨*proof*⟩

23.5 Induction rules

lemma *upper-pd-induct1*:
assumes P: *adm* P
assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\sharp; P\ ys]\!] \Longrightarrow P\ (\{x\}\sharp +\sharp ys)$
shows P $(xs::'a$ *upper-pd*$)$
⟨*proof*⟩

lemma *upper-pd-induct*:
assumes P: *adm* P
assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$

assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\sharp\ ys)$
shows $P\ (xs{::}'a\ upper\text{-}pd)$
⟨*proof*⟩

23.6 Monadic bind

definition
upper-bind-basis ::
$'a\ pd\text{-}basis \Rightarrow ('a \to 'b\ upper\text{-}pd) \to 'b\ upper\text{-}pd$ **where**
upper-bind-basis = *fold-pd*
$(\lambda a.\ \Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\sharp\ y{\cdot}f)$

lemma *ACI-upper-bind*:
ab-semigroup-idem-mult $(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\sharp\ y{\cdot}f)$
⟨*proof*⟩

lemma *upper-bind-basis-simps* [*simp*]:
upper-bind-basis $(PDUnit\ a) =$
$(\Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
upper-bind-basis $(PDPlus\ t\ u) =$
$(\Lambda\ f.\ upper\text{-}bind\text{-}basis\ t{\cdot}f +\sharp\ upper\text{-}bind\text{-}basis\ u{\cdot}f)$
⟨*proof*⟩

lemma *upper-bind-basis-mono*:
$t \leq\sharp\ u \Longrightarrow upper\text{-}bind\text{-}basis\ t \sqsubseteq upper\text{-}bind\text{-}basis\ u$
⟨*proof*⟩

definition
upper-bind :: $'a\ upper\text{-}pd \to ('a \to 'b\ upper\text{-}pd) \to 'b\ upper\text{-}pd$ **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·$\{x\}\sharp{\cdot}f = f{\cdot}x$
⟨*proof*⟩

lemma *upper-bind-plus* [*simp*]:
upper-bind·$(xs +\sharp\ ys){\cdot}f$ = *upper-bind*·$xs{\cdot}f +\sharp$ *upper-bind*·$ys{\cdot}f$
⟨*proof*⟩

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

23.7 Map and join

definition

upper-map :: $('a \to 'b) \to 'a$ *upper-pd* $\to 'b$ *upper-pd* **where**
upper-map = $(\Lambda\ f\ xs.\ upper\text{-}bind{\cdot}xs{\cdot}(\Lambda\ x.\ \{f{\cdot}x\}\sharp))$

definition
upper-join :: $'a$ *upper-pd upper-pd* $\to 'a$ *upper-pd* **where**
upper-join = $(\Lambda\ xss.\ upper\text{-}bind{\cdot}xss{\cdot}(\Lambda\ xs.\ xs))$

lemma *upper-map-unit* [*simp*]:
$upper\text{-}map{\cdot}f{\cdot}\{x\}\sharp = \{f{\cdot}x\}\sharp$
⟨*proof*⟩

lemma *upper-map-plus* [*simp*]:
$upper\text{-}map{\cdot}f{\cdot}(xs +\sharp ys) = upper\text{-}map{\cdot}f{\cdot}xs +\sharp upper\text{-}map{\cdot}f{\cdot}ys$
⟨*proof*⟩

lemma *upper-join-unit* [*simp*]:
$upper\text{-}join{\cdot}\{xs\}\sharp = xs$
⟨*proof*⟩

lemma *upper-join-plus* [*simp*]:
$upper\text{-}join{\cdot}(xss +\sharp yss) = upper\text{-}join{\cdot}xss +\sharp upper\text{-}join{\cdot}yss$
⟨*proof*⟩

lemma *upper-map-ident*: $upper\text{-}map{\cdot}(\Lambda\ x.\ x){\cdot}xs = xs$
⟨*proof*⟩

lemma *upper-map-map*:
$upper\text{-}map{\cdot}f{\cdot}(upper\text{-}map{\cdot}g{\cdot}xs) = upper\text{-}map{\cdot}(\Lambda\ x.\ f{\cdot}(g{\cdot}x)){\cdot}xs$
⟨*proof*⟩

lemma *upper-join-map-unit*:
$upper\text{-}join{\cdot}(upper\text{-}map{\cdot}upper\text{-}unit{\cdot}xs) = xs$
⟨*proof*⟩

lemma *upper-join-map-join*:
$upper\text{-}join{\cdot}(upper\text{-}map{\cdot}upper\text{-}join{\cdot}xsss) = upper\text{-}join{\cdot}(upper\text{-}join{\cdot}xsss)$
⟨*proof*⟩

lemma *upper-join-map-map*:
$upper\text{-}join{\cdot}(upper\text{-}map{\cdot}(upper\text{-}map{\cdot}f){\cdot}xss) =$
$upper\text{-}map{\cdot}f{\cdot}(upper\text{-}join{\cdot}xss)$
⟨*proof*⟩

lemma *upper-map-approx*: $upper\text{-}map{\cdot}(approx\ n){\cdot}xs = approx\ n{\cdot}xs$
⟨*proof*⟩

end

24 LowerPD: Lower powerdomain

theory *LowerPD*
imports *CompactBasis*
begin

24.1 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* ≤♭ *PDUnit b*) = *a* ⊑ *b*
⟨*proof*⟩

lemma *lower-le-PDUnit-PDPlus-iff*:
(*PDUnit a* ≤♭ *PDPlus t u*) = (*PDUnit a* ≤♭ *t* ∨ *PDUnit a* ≤♭ *u*)
⟨*proof*⟩

lemma *lower-le-PDPlus-iff*: (*PDPlus t u* ≤♭ *v*) = (*t* ≤♭ *v* ∧ *u* ≤♭ *v*)
⟨*proof*⟩

lemma *lower-le-induct* [*induct set*: *lower-le*]:
assumes *le*: *t* ≤♭ *u*
assumes *1*: ⋀*a b*. *a* ⊑ *b* ⟹ *P* (*PDUnit a*) (*PDUnit b*)
assumes *2*: ⋀*t u a*. *P* (*PDUnit a*) *t* ⟹ *P* (*PDUnit a*) (*PDPlus t u*)
assumes *3*: ⋀*t u v*. ⟦*P t v*; *P u v*⟧ ⟹ *P* (*PDPlus t u*) *v*
shows *P t u*

⟨*proof*⟩

lemma *approx-pd-lower-mono1*:
i ≤ *j* ⟹ *approx-pd i t* ≤♭ *approx-pd j t*
⟨*proof*⟩

lemma *approx-pd-lower-le*: *approx-pd i t* ≤♭ *t*
⟨*proof*⟩

lemma *approx-pd-lower-mono*:
t ≤♭ *u* ⟹ *approx-pd n t* ≤♭ *approx-pd n u*
⟨*proof*⟩

24.2 Type definition

cpodef (**open**) *'a lower-pd* =
{*S*::*'a*::*profinite pd-basis set*. *lower-le.ideal S*}
⟨*proof*⟩

lemma *ideal-Rep-lower-pd*: *lower-le.ideal* (*Rep-lower-pd x*)
⟨*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 approx-pd lower-principal Rep-lower-pd*]
⟨*proof*⟩

lemma *lower-principal-less-iff* [*simp*]:
lower-principal t ⊑ *lower-principal u* ⟷ *t* ≤♭ *u*
⟨*proof*⟩

lemma *lower-principal-eq-iff*:
lower-principal t = *lower-principal u* ⟷ *t* ≤♭ *u* ∧ *u* ≤♭ *t*
⟨*proof*⟩

lemma *lower-principal-mono*:
t ≤♭ *u* ⟹ *lower-principal t* ⊑ *lower-principal u*
⟨*proof*⟩

lemma *compact-lower-principal*: *compact* (*lower-principal t*)
⟨*proof*⟩

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*⟩

24.3 Approximation

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* (*approx-pd n t*)
⟨*proof*⟩

lemma *approx-eq-lower-principal*:
∃ *t*∈*Rep-lower-pd xs*. *approx n*·*xs* = *lower-principal* (*approx-pd n t*)
⟨*proof*⟩

lemma *compact-imp-lower-principal*:
compact xs ⟹ ∃ *t*. *xs* = *lower-principal t*
⟨*proof*⟩

lemma *lower-principal-induct*:
⟦*adm P*; ⋀*t*. *P* (*lower-principal t*)⟧ ⟹ *P xs*
⟨*proof*⟩

lemma *lower-principal-induct2*:
⟦⋀*ys*. *adm* (λ*xs*. *P xs ys*); ⋀*xs*. *adm* (λ*ys*. *P xs ys*);
⋀*t u*. *P* (*lower-principal t*) (*lower-principal u*)⟧ ⟹ *P xs ys*
⟨*proof*⟩

24.4 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 (λt. lower-pd.basis-fun (λu. 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*: *xs +♭ xs* = *xs*
⟨*proof*⟩

interpretation *aci-lower-plus*: *ab-semigroup-idem-mult* [*op +♭*]
⟨*proof*⟩

lemma *lower-plus-left-commute*: *xs +♭* (*ys +♭ zs*) = *ys +♭* (*xs +♭ zs*)
⟨*proof*⟩

lemma *lower-plus-left-absorb*: $xs +\flat (xs +\flat ys) = xs +\flat ys$
⟨*proof*⟩

lemmas *lower-plus-aci* = *aci-lower-plus.mult-ac-idem*

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*⟩

24.5 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*⟩

24.6 Monadic bind

definition
lower-bind-basis ::
$'a$ *pd-basis* $\Rightarrow$ $('a \to 'b$ *lower-pd*$) \to 'b$ *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 $(PDUnit\ a) =$
$(\Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
lower-bind-basis $(PDPlus\ t\ u) =$
$(\Lambda\ f.$ *lower-bind-basis* $t{\cdot}f +\flat$ *lower-bind-basis* $u{\cdot}f)$
⟨*proof*⟩

lemma *lower-bind-basis-mono*:
$t \le\flat u \Longrightarrow$ *lower-bind-basis* $t \sqsubseteq$ *lower-bind-basis* u
⟨*proof*⟩

definition
lower-bind :: $'a$ *lower-pd* $\to ('a \to 'b$ *lower-pd*$) \to 'b$ *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*⟩

24.7 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

25 ConvexPD: Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

25.1 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* ≤♮ *t*
⟨*proof*⟩

lemma *PDUnit-convex-mono*: *x* ⊑ *y* ⟹ *PDUnit x* ≤♮ *PDUnit y*
⟨*proof*⟩

lemma *PDPlus-convex-mono*: ⟦*s* ≤♮ *t*; *u* ≤♮ *v*⟧ ⟹ *PDPlus s u* ≤♮ *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*) = (∀ *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* ∧ *PDUnit a* $\leq\natural$ *u*)
⟨*proof*⟩

lemma *convex-le-PDUnit-lemma2*:
(*t* $\leq\natural$ *PDUnit b*) = (∀ *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* ∧ *u* $\leq\natural$ *PDUnit a*)
⟨*proof*⟩

lemma *convex-le-PDPlus-lemma*:
assumes *z*: *PDPlus t u* $\leq\natural$ *z*
shows ∃ *v w*. *z* = *PDPlus v w* ∧ *t* $\leq\natural$ *v* ∧ *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*⟧ ⟹ *P t v*
assumes *3*: $\bigwedge$*a b*. *a* $\sqsubseteq$ *b* ⟹ *P* (*PDUnit a*) (*PDUnit b*)
assumes *4*: $\bigwedge$*t u v w*. ⟦*P t v*; *P u w*⟧ ⟹ *P* (*PDPlus t u*) (*PDPlus v w*)
shows *P t u*
⟨*proof*⟩

lemma *approx-pd-convex-mono1*:
i ≤ *j* ⟹ *approx-pd i t* $\leq\natural$ *approx-pd j t*
⟨*proof*⟩

lemma *approx-pd-convex-le*: *approx-pd i t* $\leq\natural$ *t*
⟨*proof*⟩

lemma *approx-pd-convex-mono*:
t $\leq\natural$ *u* ⟹ *approx-pd n t* $\leq\natural$ *approx-pd n u*
⟨*proof*⟩

25.2 Type definition

cpodef (**open**) *'a convex-pd* =
{*S*::*'a*::*profinite pd-basis set*. *convex-le.ideal S*}

⟨*proof*⟩

lemma *ideal-Rep-convex-pd*: *convex-le.ideal* (*Rep-convex-pd xs*)
⟨*proof*⟩

lemma *Rep-convex-pd-mono*: $xs \sqsubseteq ys \Longrightarrow$ *Rep-convex-pd xs* $\subseteq$ *Rep-convex-pd ys*
⟨*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 approx-pd convex-principal Rep-convex-pd*]
⟨*proof*⟩

lemma *convex-principal-less-iff* [*simp*]:
$$\textit{convex-principal } t \sqsubseteq \textit{convex-principal } u \longleftrightarrow t \leq\natural u$$
⟨*proof*⟩

lemma *convex-principal-eq-iff* [*simp*]:
$$\textit{convex-principal } t = \textit{convex-principal } u \longleftrightarrow t \leq\natural u \wedge u \leq\natural t$$
⟨*proof*⟩

lemma *convex-principal-mono*:
$$t \leq\natural u \Longrightarrow \textit{convex-principal } t \sqsubseteq \textit{convex-principal } u$$
⟨*proof*⟩

lemma *compact-convex-principal*: *compact* (*convex-principal t*)
⟨*proof*⟩

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*
⟨*proof*⟩

instance *convex-pd* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-convex-pd-pcpo*: $\bot$ = *convex-principal* (*PDUnit compact-bot*)
⟨*proof*⟩

25.3 Approximation

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* (*approx-pd n t*)
⟨*proof*⟩

lemma *approx-eq-convex-principal*:
∃ *t*∈*Rep-convex-pd xs*. *approx n*·*xs* = *convex-principal* (*approx-pd n t*)
⟨*proof*⟩

lemma *compact-imp-convex-principal*:
compact xs ⟹ ∃ *t*. *xs* = *convex-principal t*
⟨*proof*⟩

lemma *convex-principal-induct*:
⟦*adm P*; ⋀*t*. *P* (*convex-principal t*)⟧ ⟹ *P xs*
⟨*proof*⟩

lemma *convex-principal-induct2*:
⟦⋀*ys*. *adm* (λ*xs*. *P xs ys*); ⋀*xs*. *adm* (λ*ys*. *P xs ys*);
⋀*t u*. *P* (*convex-principal t*) (*convex-principal u*)⟧ ⟹ *P xs ys*
⟨*proof*⟩

25.4 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*: $xs +\natural xs = xs$
⟨*proof*⟩

interpretation *aci-convex-plus*: *ab-semigroup-idem-mult* [$op +\natural$]
⟨*proof*⟩

lemma *convex-plus-left-commute*: $xs +\natural (ys +\natural zs) = ys +\natural (xs +\natural zs)$
⟨*proof*⟩

lemma *convex-plus-left-absorb*: $xs +\natural (xs +\natural ys) = xs +\natural ys$
⟨*proof*⟩

lemmas *convex-plus-aci* = *aci-convex-plus.mult-ac-idem*

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*⟩

25.5 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*⟩

25.6 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*) =
(Λ *f*. *f*·(*Rep-compact-basis a*))
convex-bind-basis (*PDPlus t u*) =
(Λ *f*. *convex-bind-basis t*·*f* +♮ *convex-bind-basis u*·*f*)
⟨*proof*⟩

lemma *monofun-LAM*:
⟦*cont f*; *cont g*; ⋀*x*. *f x* ⊑ *g x*⟧ ⟹ (Λ *x*. *f x*) ⊑ (Λ *x*. *g x*)
⟨*proof*⟩

lemma *convex-bind-basis-mono*:
t ≤♮ *u* ⟹ *convex-bind-basis t* ⊑ *convex-bind-basis u*
⟨*proof*⟩

definition
convex-bind :: 'a *convex-pd* → ('a → 'b *convex-pd*) → '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*⟩

25.7 Map and join

definition
convex-map :: ('a → 'b) → 'a *convex-pd* → 'b *convex-pd* **where**
convex-map = (Λ *f xs*. *convex-bind*·*xs*·(Λ *x*. {*f*·*x*}♮))

definition
convex-join :: 'a *convex-pd convex-pd* → '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*⟩

25.8 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*⟩

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*⟩

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

theory *HOLCF*
imports *Sprod Ssum Up Lift Discrete One Tr Domain ConvexPD Main*
uses
holcf-logic.ML
Tools/cont-consts.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