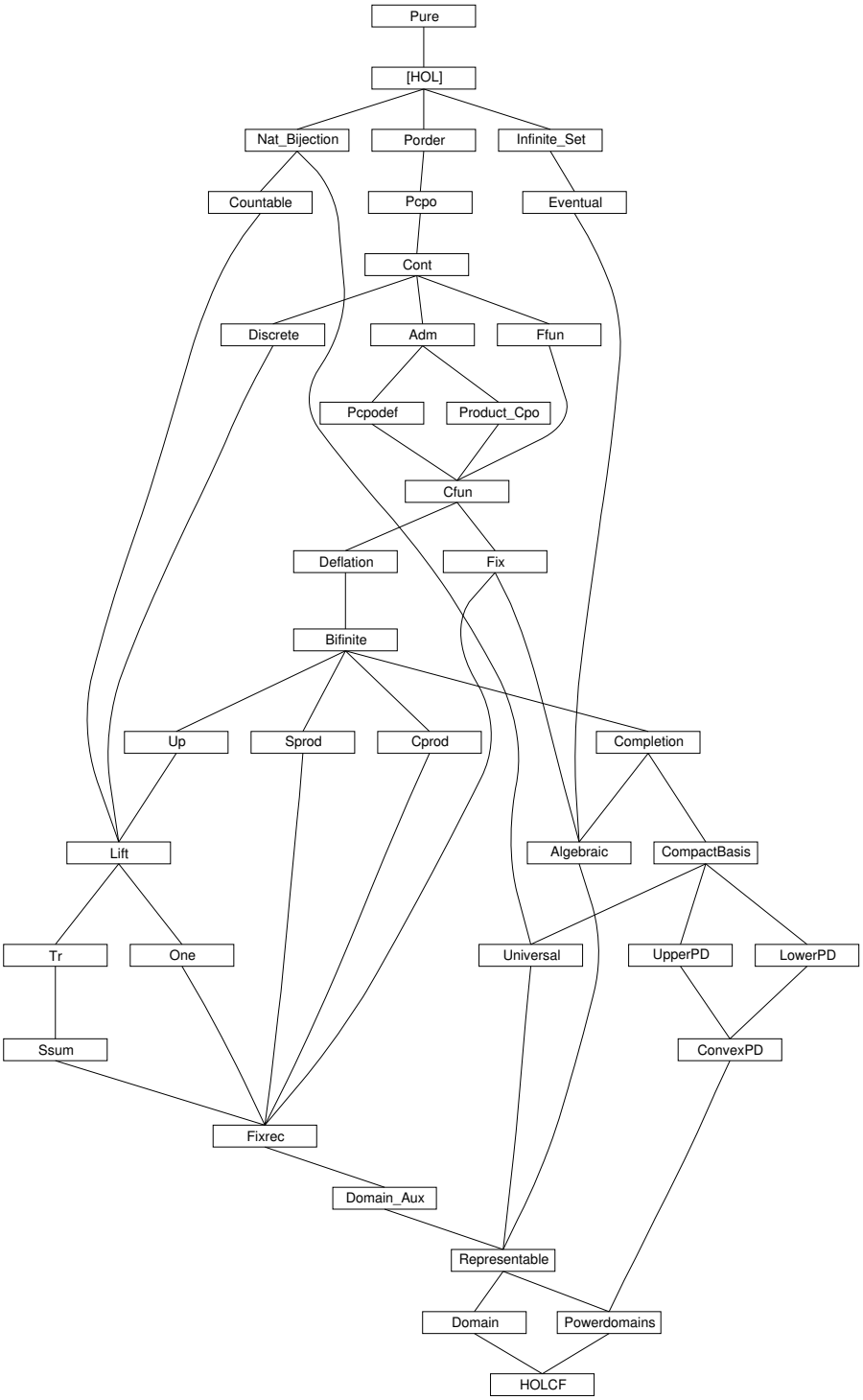
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

June 21, 2010

Contents


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

1 Porder: Partial orders

theory *Porder*
imports *Main*
begin

1.1 Type class for partial orders

class *below* =
  fixes *below* :: $'a \Rightarrow 'a \Rightarrow bool$
begin

notation
  below (**infixl** $<<$ *55*)

notation (*xsymbols*)
  below (**infixl** $\sqsubseteq$ *55*)

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

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

end

class *po* = *below* +
  assumes *below-refl* [*iff*]: $x \sqsubseteq x$
  assumes *below-trans*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
  assumes *below-antisym*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq x \Longrightarrow x = y$
begin

minimal fixes least element

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

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

lemma *below-antisym-inverse*: $x = y \Longrightarrow x \sqsubseteq y \wedge y \sqsubseteq x$
  ⟨*proof*⟩

lemma *box-below*: $a \sqsubseteq b \Longrightarrow c \sqsubseteq a \Longrightarrow b \sqsubseteq d \Longrightarrow c \sqsubseteq d$
  ⟨*proof*⟩

lemma *po-eq-conv*: $x = y \longleftrightarrow x \sqsubseteq y \wedge y \sqsubseteq x$
  ⟨*proof*⟩

lemma *rev-below-trans*: $y \sqsubseteq z \Longrightarrow x \sqsubseteq y \Longrightarrow x \sqsubseteq z$
  ⟨*proof*⟩

lemma *not-below2not-eq*: $\neg\ x \sqsubseteq y \Longrightarrow x \neq y$
⟨*proof*⟩

end

lemmas *HOLCF-trans-rules* [*trans*] =
below-trans
below-antisym
below-eq-trans
eq-below-trans

context *po*
begin

1.2 Upper bounds

definition *is-ub* :: $'a\ set \Rightarrow 'a \Rightarrow bool$ (**infixl** $<|$ *55*) **where**
$S <|\ x \longleftrightarrow (\forall y.\ y \in S \longrightarrow y \sqsubseteq x)$

lemma *is-ubI*: $(\bigwedge x.\ x \in S \Longrightarrow x \sqsubseteq u) \Longrightarrow S <|\ u$
⟨*proof*⟩

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

lemma *ub-imageI*: $(\bigwedge x.\ x \in S \Longrightarrow f\ x \sqsubseteq u) \Longrightarrow (\lambda x.\ f\ x)\ `\ S <|\ u$
⟨*proof*⟩

lemma *ub-imageD*: $[\![f\ `\ S <|\ u; x \in S]\!] \Longrightarrow f\ x \sqsubseteq u$
⟨*proof*⟩

lemma *ub-rangeI*: $(\bigwedge i.\ S\ i \sqsubseteq x) \Longrightarrow range\ S <|\ x$
⟨*proof*⟩

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

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

lemma *is-ub-insert* [*simp*]: $(insert\ x\ A) <|\ y = (x \sqsubseteq y \wedge A <|\ y)$
⟨*proof*⟩

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

1.3 Least upper bounds

definition *is-lub* :: $'a\ set \Rightarrow 'a \Rightarrow bool$ (**infixl** $<<|$ *55*) **where**
$S <<|\ x \longleftrightarrow S <|\ x \wedge (\forall u.\ S <|\ u \longrightarrow x \sqsubseteq u)$

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

end

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*)

context *po*
begin

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

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

access to some definition as inference rule

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

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

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

lubs are unique

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

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

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

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

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

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

lemma *is-lub-bin*: $x \sqsubseteq y \Longrightarrow \{x, y\} <<| y$
⟨*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 *chain* :: $(nat \Rightarrow 'a) \Rightarrow bool$ **where**
— Here we use countable chains and I prefer to code them as functions!
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-less*: $[\![chain\ Y; i < j]\!] \Longrightarrow Y\ i \sqsubseteq Y\ j$
⟨*proof*⟩

lemma *chain-mono*: $[\![chain\ Y; i \leq j]\!] \Longrightarrow Y\ i \sqsubseteq Y\ j$
⟨*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 Finite chains

definition *max-in-chain* :: *nat* ⇒ (*nat* ⇒ 'a) ⇒ *bool* **where**
— finite chains, needed for monotony of continuous functions
max-in-chain i C ⟷ ($\forall j.\ i \leq j \longrightarrow C\ i = C\ j$)

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

results about finite chains

lemma *max-in-chainI*: ($\bigwedge j.\ i \leq j \Longrightarrow Y\ i = Y\ j$) ⟹ *max-in-chain i Y*
⟨*proof*⟩

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

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

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

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

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

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

lemma *finite-range-has-max*:
fixes *f* :: *nat* ⇒ 'a **and** *r* :: 'a ⇒ 'a ⇒ *bool*
assumes *mono*: $\bigwedge i\ j.\ i \leq j \Longrightarrow r\ (f\ i)\ (f\ j)$
assumes *finite-range*: *finite* (*range f*)
shows $\exists k.\ \forall i.\ r\ (f\ i)\ (f\ k)$
⟨*proof*⟩

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

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

lemma *bin-chainmax*:
$x \sqsubseteq y \Longrightarrow$ *max-in-chain* (*Suc 0*) (λi. *if i=0 then x else y*)
⟨*proof*⟩

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

the maximal element in a chain is its lub

lemma *lub-chain-maxelem*: ⟦$Y\ i = c$; $\forall i.\ Y\ i \sqsubseteq c$⟧ ⟹ *lub* (*range Y*) = *c*
⟨*proof*⟩

1.6 Directed sets

definition *directed* :: *'a set* ⇒ *bool* **where**
directed S ⟷ ($\exists x.\ x \in S$) ∧ ($\forall x{\in}S.\ \forall y{\in}S.\ \exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$)

lemma *directedI*:
assumes *1*: $\exists z.\ z \in S$
assumes *2*: $\bigwedge x\ y.$ ⟦$x \in S$; $y \in S$⟧ ⟹ $\exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$
shows *directed S*
⟨*proof*⟩

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

lemma *directedD2*: ⟦*directed S*; $x \in S$; $y \in S$⟧ ⟹ $\exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$
⟨*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: $\bigwedge U.\ [\![\mathit{finite}\ U;\ U \subseteq S]\!] \Longrightarrow \exists\, z{\in}S.\ U <|\ z$
shows *directed* S
⟨*proof*⟩

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

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

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

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

lemma *directed-chain*: *chain* $S \Longrightarrow$ *directed* (*range* S)
⟨*proof*⟩

lemmata for improved admissibility introdution rule

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

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

end

end

2 Pcpo: Classes cpo and pcpo

theory *Pcpo*
imports *Porder*
begin

2.1 Complete partial orders

The class cpo of chain complete partial orders

class *cpo* = *po* +
  assumes *cpo*: *chain S* $\Longrightarrow \exists x.$ *range S* <<| *x*
begin

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

lemma *cpo-lubI*: *chain S* $\Longrightarrow$ *range S* <<| $(\bigsqcup i.\ S\ i)$
  ⟨*proof*⟩

lemma *thelubE*: ⟦*chain S*; $(\bigsqcup i.\ S\ i) = l$⟧ $\Longrightarrow$ *range S* <<| *l*
  ⟨*proof*⟩

Properties of the lub

lemma *is-ub-thelub*: *chain S* $\Longrightarrow S\ x \sqsubseteq (\bigsqcup i.\ S\ i)$
  ⟨*proof*⟩

lemma *is-lub-thelub*:
  ⟦*chain S*; *range S* <| *x*⟧ $\Longrightarrow (\bigsqcup i.\ S\ i) \sqsubseteq x$
  ⟨*proof*⟩

lemma *lub-range-mono*:
  ⟦*range X* $\subseteq$ *range Y*; *chain Y*; *chain X*⟧
    $\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

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

lemma *maxinch-is-thelub*:
  chain Y $\Longrightarrow$ *max-in-chain i Y* $= ((\bigsqcup i.\ Y\ i) = Y\ i)$
⟨*proof*⟩

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

lemma *lub-mono*:
  ⟦*chain X*; *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*; *chain Y*; $\forall k.\ X\ k = Y\ k$⟧
    $\Longrightarrow (\bigsqcup i.\ X\ i) = (\bigsqcup i.\ Y\ i)$
  ⟨*proof*⟩

lemma *lub-eq*:
  $(\bigwedge i.\ X\ i = Y\ i) \Longrightarrow (\bigsqcup i.\ X\ i) = (\bigsqcup i.\ Y\ i)$
  ⟨*proof*⟩

more results about mono and = of lubs of chains

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

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

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

lemma *ch2ch-lub*:
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*:
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*:
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*⟩

end

2.2 Pointed cpos

The class pcpo of pointed cpos

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

definition *UU* :: *'a* **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*⟩

end

Simproc to rewrite $\bot = x$ to $x = \bot$.

⟨*ML*⟩

context *pcpo*
begin

useful lemmas about $\bot$

lemma *below-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 *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*⟩

end

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

class *chfin* = *po* +
assumes *chfin*: $chain\ Y \Longrightarrow \exists n.\ max\text{-}in\text{-}chain\ n\ Y$
begin

subclass *cpo*
⟨*proof*⟩

lemma *chfin2finch*: *chain* $Y \Longrightarrow$ *finite-chain* Y
⟨*proof*⟩

end

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

subclass *chfin*
⟨*proof*⟩

end

class *flat* = *pcpo* +
assumes *ax-flat*: $x \sqsubseteq y \Longrightarrow x = \bot \vee x = y$
begin

subclass *chfin*
⟨*proof*⟩

lemma *flat-below-iff*:
shows $(x \sqsubseteq y) = (x = \bot \vee x = y)$
⟨*proof*⟩

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

end

Discrete cpos

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

subclass *po*
⟨*proof*⟩

In a discrete cpo, every chain is constant

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

subclass *cpo*
⟨*proof*⟩

end

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

default-sort *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
$cont :: ('a{::}cpo \Rightarrow 'b{::}cpo) \Rightarrow bool$
where
$cont\ f = (\forall Y.\ chain\ Y \longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i))$

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 Equivalence of alternate definition

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*⟧ ⟹ *range* (λ*i*. *f* (*Y i*)) <| *f u*
⟨*proof*⟩

a lemma about binary chains

lemma *binchain-cont*:
  ⟦*cont f*; *x* ⊑ *y*⟧ ⟹ *range* (λ*i*::*nat*. *f* (*if i* = *0 then x else y*)) <<| *f y*
⟨*proof*⟩

continuity implies monotonicity

lemma *cont2mono*: *cont f* ⟹ *monofun f*
⟨*proof*⟩

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

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

continuity implies preservation of lubs

lemma *cont2contlubE*:
  ⟦*cont f*; *chain Y*⟧ ⟹ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*))
⟨*proof*⟩

lemma *contI2*:
  assumes *mono*: *monofun f*
  assumes *below*: ⋀*Y*. ⟦*chain Y*; *chain* (λ*i*. *f* (*Y i*))⟧
    ⟹ *f* (⨆ *i*. *Y i*) ⊑ (⨆ *i*. *f* (*Y i*))
  shows *cont f*
⟨*proof*⟩

3.3 Collection of continuity rules

⟨*ML*⟩

3.4 Continuity of basic functions

The identity function is continuous

lemma *cont-id* [*simp*, *cont2cont*]: *cont* (λ*x*. *x*)
⟨*proof*⟩

constant functions are continuous

lemma *cont-const* [*simp*, *cont2cont*]: *cont* (λ*x*. *c*)
⟨*proof*⟩

application of functions is continuous

lemma *cont-apply*:
fixes $f :: 'a::cpo \Rightarrow 'b::cpo \Rightarrow 'c::cpo$ **and** $t :: 'a \Rightarrow 'b$
assumes *1*: $cont\ (\lambda x.\ t\ x)$
assumes *2*: $\bigwedge x.\ cont\ (\lambda y.\ f\ x\ y)$
assumes *3*: $\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)$
shows $cont\ (\lambda x.\ (f\ x)\ (t\ x))$
⟨*proof*⟩

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

if-then-else is continuous

lemma *cont-if* [*simp*, *cont2cont*]:
$[\![cont\ f;\ cont\ g]\!] \Longrightarrow cont\ (\lambda x.\ if\ b\ then\ f\ x\ else\ g\ x)$
⟨*proof*⟩

3.5 Finite chains and flat pcpos

monotone functions map finite chains to finite chains

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

The same holds for continuous functions

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

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

some properties of flat

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

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

functions with discrete domain

lemma *cont-discrete-cpo* [*simp*, *cont2cont*]: $cont\ (f::'a::discrete\text{-}cpo \Rightarrow 'b::cpo)$
⟨*proof*⟩

end

4 Discrete: Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

4.1 Discrete ordering

instantiation *discr :: (type) below*
begin

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

instance ⟨*proof*⟩
end

4.2 Discrete cpo class instance

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

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

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*⟩

4.3 *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

5 Adm: Admissibility and compactness

theory *Adm*
imports *Cont*
begin

default-sort *cpo*

5.1 Definitions

definition
adm :: ('*a*::*cpo* ⇒ *bool*) ⇒ *bool* **where**
adm P = (∀ *Y*. *chain Y* ⟶ (∀ *i*. *P* (*Y i*)) ⟶ *P* (⨆ *i*. *Y i*))

lemma *admI*:
(⋀*Y*. ⟦*chain Y*; ∀ *i*. *P* (*Y i*)⟧ ⟹ *P* (⨆ *i*. *Y i*)) ⟹ *adm P*
⟨*proof*⟩

lemma *admD*: ⟦*adm P*; *chain Y*; ⋀*i*. *P* (*Y i*)⟧ ⟹ *P* (⨆ *i*. *Y i*)
⟨*proof*⟩

lemma *admD2*: ⟦*adm* (λ*x*. ¬ *P x*); *chain Y*; *P* (⨆ *i*. *Y i*)⟧ ⟹ ∃ *i*. *P* (*Y i*)
⟨*proof*⟩

lemma *triv-admI*: ∀ *x*. *P x* ⟹ *adm P*
⟨*proof*⟩

improved admissibility introduction

lemma *admI2*:
(⋀*Y*. ⟦*chain Y*; ∀ *i*. *P* (*Y i*); ∀ *i*. ∃ *j*>*i*. *Y i* ≠ *Y j* ∧ *Y i* ⊑ *Y j*⟧
⟹ *P* (⨆ *i*. *Y i*)) ⟹ *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*: $(\bigwedge y.\ adm\ (\lambda x.\ P\ x\ y)) \Longrightarrow adm\ (\lambda x.\ \forall y.\ P\ x\ y)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

$\Longrightarrow$ *adm* ($\lambda x.$ *P x* = *Q x*)
⟨*proof*⟩

lemma *adm-not-conj*:
⟦*adm* ($\lambda x.$ ¬ *P x*); *adm* ($\lambda x.$ ¬ *Q x*)⟧ $\Longrightarrow$ *adm* ($\lambda x.$ ¬ (*P x* ∧ *Q x*))
⟨*proof*⟩

admissibility and continuity

lemma *adm-below*: ⟦*cont u*; *cont v*⟧ $\Longrightarrow$ *adm* ($\lambda x.$ *u x* ⊑ *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-below*: *cont t* $\Longrightarrow$ *adm* ($\lambda x.$ ¬ *t x* ⊑ *u*)
⟨*proof*⟩

5.4 Compactness

definition
compact :: *'a*::*cpo* ⇒ *bool* **where**
compact k = *adm* ($\lambda x.$ ¬ *k* ⊑ *x*)

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

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

lemma *compactI2*:
(⋀*Y*. ⟦*chain Y*; *x* ⊑ (⨆ *i*. *Y i*)⟧ $\Longrightarrow$ ∃ *i*. *x* ⊑ *Y i*) $\Longrightarrow$ *compact x*
⟨*proof*⟩

lemma *compactD2*:
⟦*compact x*; *chain Y*; *x* ⊑ (⨆ *i*. *Y i*)⟧ $\Longrightarrow$ ∃ *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*)⟧ $\Longrightarrow$ ∃ *i*. *max-in-chain i Y*
⟨*proof*⟩

admissibility and compactness

lemma *adm-compact-not-below*: ⟦*compact k*; *cont t*⟧ $\Longrightarrow$ *adm* ($\lambda x.$ ¬ *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-below adm-eq adm-not-below
adm-compact-not-below adm-compact-neq adm-neq-compact adm-not-UU

end

6 Pcpodef: Subtypes of pcpos

theory *Pcpodef*
imports *Adm*
uses (*Tools/pcpodef.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.

⟨*ML*⟩

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

⟨*ML*⟩

6.2 Proving a subtype is finite

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

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

6.3 Proving a subtype is chain-finite

lemma *monofun-Rep*:
assumes *below*: *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 *below*: *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 *below*: *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 *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
shows $chain\ S \Longrightarrow range\ S <<|\ Abs\ (\bigsqcup i.\ Rep\ (S\ i))$
⟨*proof*⟩

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

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

6.4.1 Continuity of *Rep* and *Abs*

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

theorem *typedef-cont-Rep*:
fixes $Abs :: 'a{::}cpo \Rightarrow 'b{::}cpo$
assumes *type*: *type-definition Rep Abs A*
and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
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 *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
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 *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
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* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *compact* (*Rep k*) ⟹ *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* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *z-in-A*: *z* ∈ *A*
and *z-least*: ⋀*x*. *x* ∈ *A* ⟹ *z* ⊑ *x*
shows *OFCLASS*(*'b*, *pcpo-class*)
⟨*proof*⟩

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

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

6.6.1 Strictness of *Rep* and *Abs*

For a sub-pcpo where ⊥ 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 *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *Abs* ⊥ = ⊥
⟨*proof*⟩

theorem *typedef-Rep-strict*:
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *Rep* ⊥ = ⊥
⟨*proof*⟩

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

6.8 HOLCF type definition package

⟨*ML*⟩

end

7 Ffun: Class instances for the full function space

theory *Ffun*

imports *Cont*
begin

7.1 Full function space is a partial order

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

definition
below-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-below*: $(f \sqsubseteq g) = (\forall x.\ f\ x \sqsubseteq g\ x)$
⟨*proof*⟩

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

7.2 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*)
⟹ (⨆ *i*. *S i*) = (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*⟩

7.3 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*⟩

7.4 Propagation of monotonicity and continuity

the lub of a chain of monotone functions is monotone

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

the lub of a chain of continuous functions is continuous

lemma *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 *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

8 Product-Cpo: The cpo of cartesian products

theory *Product-Cpo*
imports *Adm*
begin

default-sort *cpo*

8.1 Unit type is a pcpo

instantiation *unit* :: *below*
begin

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

instance ⟨*proof*⟩
end

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

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

instance *unit* :: *pcpo*

⟨*proof*⟩

8.2 Product type is a partial order

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

definition
below-prod-def: $(op \sqsubseteq) \equiv \lambda p1\ p2.\ (fst\ p1 \sqsubseteq fst\ p2 \wedge snd\ p1 \sqsubseteq snd\ p2)$

instance ⟨*proof*⟩
end

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

8.3 Monotonicity of *Pair*, *fst*, *snd*

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

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

Pair (-,-) is monotone in both arguments

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

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

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

lemma *ch2ch-Pair* [*simp*]:
chain $X \Longrightarrow$ *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ (X\ i,\ Y\ i))$
⟨*proof*⟩

fst and *snd* are monotone

lemma *fst-monofun*: $x \sqsubseteq y \Longrightarrow fst\ x \sqsubseteq fst\ y$
⟨*proof*⟩

lemma *snd-monofun*: $x \sqsubseteq y \Longrightarrow snd\ x \sqsubseteq snd\ y$
⟨*proof*⟩

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

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

lemmas *ch2ch-fst* [*simp*] = *ch2ch-monofun* [*OF monofun-fst*]

lemmas *ch2ch-snd* [*simp*] = *ch2ch-monofun* [*OF monofun-snd*]

lemma *prod-chain-cases*:
assumes *chain Y*
obtains *A B*
where *chain A* **and** *chain B* **and** $Y = (\lambda i.\ (A\ i,\ B\ i))$
⟨*proof*⟩

8.4 Product type is a cpo

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

lemma *thelub-Pair*:
$[\![chain\ (A{::}nat \Rightarrow {'a}{::}cpo);\ chain\ (B{::}nat \Rightarrow {'b}{::}cpo)]\!]$
$\Longrightarrow (\bigsqcup i.\ (A\ i,\ B\ i)) = (\bigsqcup i.\ A\ i,\ \bigsqcup i.\ B\ i)$
⟨*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*:
$chain\ (S{::}nat \Rightarrow {'a}{::}cpo \times {'b}{::}cpo)$
$\Longrightarrow (\bigsqcup i.\ S\ i) = (\bigsqcup i.\ fst\ (S\ i),\ \bigsqcup i.\ snd\ (S\ i))$
⟨*proof*⟩

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

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

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

8.5 Product type is pointed

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

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

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

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

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

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

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

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

8.6 Continuity of *Pair*, *fst*, *snd*

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

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

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

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

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

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

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

lemma *cont2cont-split*:
assumes *f1*: $\bigwedge a\ b.$ *cont* $(\lambda x.\ f\ x\ a\ b)$
assumes *f2*: $\bigwedge x\ b.$ *cont* $(\lambda a.\ f\ x\ a\ b)$
assumes *f3*: $\bigwedge x\ a.$ *cont* $(\lambda b.\ f\ x\ a\ b)$
assumes *g*: *cont* $(\lambda x.\ g\ x)$
shows *cont* $(\lambda x.\ split\ (\lambda a\ b.\ f\ x\ a\ b)\ (g\ x))$

⟨*proof*⟩

lemma *cont-fst-snd-D1*:
  cont (λ*p*. *f* (*fst p*) (*snd p*)) ⟹ *cont* (λ*x*. *f x y*)
⟨*proof*⟩

lemma *cont-fst-snd-D2*:
  cont (λ*p*. *f* (*fst p*) (*snd p*)) ⟹ *cont* (λ*y*. *f x y*)
⟨*proof*⟩

lemma *cont2cont-split'* [*simp*, *cont2cont*]:
  assumes *f*: *cont* (λ*p*. *f* (*fst p*) (*fst* (*snd p*)) (*snd* (*snd p*)))
  assumes *g*: *cont* (λ*x*. *g x*)
  shows *cont* (λ*x*. *split* (*f x*) (*g x*))
⟨*proof*⟩

8.7 Compactness and chain-finiteness

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

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

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

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

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

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

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

end

9 Cfun: The type of continuous functions

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

default-sort *cpo*

9.1 Definition of continuous function type

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

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

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

type-notation (*xsymbols*)
cfun $((- \rightarrow / -)\ [1, 0]\ 0)$

notation
Rep-CFun $((\text{-}\$/\text{-})\ [999,1000]\ 999)$

notation (*xsymbols*)
Rep-CFun $((\text{-}\cdot/\text{-})\ [999,1000]\ 999)$

notation (*HTML* **output**)
Rep-CFun $((\text{-}\cdot/\text{-})\ [999,1000]\ 999)$

9.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: $'a$

⟨*ML*⟩

To avoid eta-contraction of body:

⟨*ML*⟩
Syntax for nested abstractions

syntax
-Lambda :: $[cargs, 'a] \Rightarrow logic$ $((3LAM\ \text{-}./\ \text{-})\ [1000, 10]\ 10)$

syntax (*xsymbols*)
-Lambda :: $[cargs, 'a] \Rightarrow logic$ $((3\Lambda\ \text{-}./\ \text{-})\ [1000, 10]\ 10)$

⟨*ML*⟩

Dummy patterns for continuous abstraction

translations
$\Lambda\ \text{-}.\ t => CONST\ Abs\text{-}CFun\ (\lambda\ \text{-}.\ t)$

9.3 Continuous function space is pointed

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

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

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

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

instance *cfun* :: (*cpo*, *pcpo*) *pcpo*
⟨*proof*⟩

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

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

function application is strict in its first argument

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

lemma *LAM-strict* [*simp*]: $(\Lambda\ x.\ \bot) = \bot$
⟨*proof*⟩

for compatibility with old HOLCF-Version

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

9.4 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*: *cont* $f \Longrightarrow (\Lambda\ x.\ f\ x) \cdot u = f\ u$
⟨*proof*⟩

Beta-reduction simproc

Given the term $(\Lambda\ x.\ f\ x) \cdot y$, the procedure tries to construct the theorem $(\Lambda\ x.\ f\ x) \cdot y \equiv f\ y$. If this theorem cannot be completely solved by the cont2cont rules, then the procedure returns the ordinary conditional *beta-cfun* rule.

The simproc does not solve any more goals that would be solved by using *beta-cfun* as a simp rule. The advantage of the simproc is that it can avoid deeply-nested calls to the simplifier that would otherwise be caused by large continuity side conditions.

⟨*ML*⟩

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-below*: $f \sqsubseteq g = (\forall x.\ f{\cdot}x \sqsubseteq g{\cdot}x)$
⟨*proof*⟩

lemma *below-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*⟩

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

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

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

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

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

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

monotonicity of application

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

lemma *monofun-cfun-arg*: $x \sqsubseteq y \Longrightarrow f\cdot x \sqsubseteq f\cdot y$
⟨*proof*⟩

lemma *monofun-cfun*: $[\![f \sqsubseteq g;\ x \sqsubseteq y]\!] \Longrightarrow f\cdot x \sqsubseteq g\cdot y$
⟨*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* $F \Longrightarrow (\bigsqcup i.\ F\ i) = (\Lambda\ x.\ \bigsqcup i.\ F\ i{\cdot}x)$
⟨*proof*⟩

9.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

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

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

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

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

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

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

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

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

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

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

9.7 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*: $chain\ (Y{::}nat \Rightarrow 'a{::}cpo \to 'b{::}chfin)$
$\Longrightarrow\ !s.\ ?\ n.\ (LUB\ i.\ Y\ i)\$s = Y\ n\$s$
⟨*proof*⟩

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

9.8 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-below*:
$\forall x.\ f{\cdot}(g{\cdot}x) = x \Longrightarrow (g{\cdot}x \sqsubseteq g{\cdot}y) = (x \sqsubseteq y)$

⟨*proof*⟩

lemma *injection-defined-rev*:
⟦∀ *x*. *f*·(*g*·*x*) = *x*; *g*·*z* = ⊥⟧ ⟹ *z* = ⊥
⟨*proof*⟩

lemma *injection-defined*:
⟦∀ *x*. *f*·(*g*·*x*) = *x*; *z* ≠ ⊥⟧ ⟹ *g*·*z* ≠ ⊥
⟨*proof*⟩

propagation of flatness and chain-finiteness by retractions

lemma *chfin2chfin*:
∀ *y*. (*f*::'*a*::*chfin* → '*b*)·(*g*·*y*) = *y*
⟹ ∀ *Y*::*nat* ⇒ '*b*. *chain Y* ⟶ (∃ *n*. *max-in-chain n Y*)
⟨*proof*⟩

lemma *flat2flat*:
∀ *y*. (*f*::'*a*::*flat* → '*b*::*pcpo*)·(*g*·*y*) = *y*
⟹ ∀ *x y*::'*b*. *x* ⊑ *y* ⟶ *x* = ⊥ ∨ *x* = *y*
⟨*proof*⟩

a result about functions with flat codomain

lemma *flat-eqI*: ⟦(*x*::'*a*::*flat*) ⊑ *y*; *x* ≠ ⊥⟧ ⟹ *x* = *y*
⟨*proof*⟩

lemma *flat-codom*:
f·*x* = (*c*::'*b*::*flat*) ⟹ *f*·⊥ = ⊥ ∨ (∀ *z*. *f*·*z* = *c*)
⟨*proof*⟩

9.9 Identity and composition

definition
ID :: '*a* → '*a* **where**
ID = (Λ *x*. *x*)

definition
cfcomp :: ('*b* → '*c*) → ('*a* → '*b*) → '*a* → '*c* **where**
oo-def: *cfcomp* = (Λ *f g x*. *f*·(*g*·*x*))

abbreviation
cfcomp-syn :: ['*b* → '*c*, '*a* → '*b*] ⇒ '*a* → '*c* (**infixr** *oo 100*) **where**
f oo g == *cfcomp*·*f*·*g*

lemma *ID1* [*simp*]: *ID*·*x* = *x*
⟨*proof*⟩

lemma *cfcomp1*: (*f oo g*) = (Λ *x*. *f*·(*g*·*x*))
⟨*proof*⟩

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

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

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

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

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

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

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

9.10 Strictified functions

default-sort *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 *cont-strictify2*: $cont\ (\lambda x.\ if\ x = \bot\ then\ \bot\ else\ f\cdot x)$
⟨*proof*⟩

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*]: $x \neq \bot \Longrightarrow strictify\cdot f\cdot x = f\cdot x$
⟨*proof*⟩

9.11 Continuity of let-bindings

lemma *cont2cont-Let*:
assumes f: *cont* $(\lambda x.\ f\ x)$
assumes *g1*: $\bigwedge y.$ *cont* $(\lambda x.\ g\ x\ y)$
assumes *g2*: $\bigwedge x.$ *cont* $(\lambda y.\ g\ x\ y)$
shows *cont* $(\lambda x.$ *let* $y = f\ x$ *in* $g\ x\ y)$
⟨*proof*⟩

lemma *cont2cont-Let'* [*simp*, *cont2cont*]:
assumes f: *cont* $(\lambda x.\ f\ x)$
assumes g: *cont* $(\lambda p.\ g\ (fst\ p)\ (snd\ p))$
shows *cont* $(\lambda x.$ *let* $y = f\ x$ *in* $g\ x\ y)$
⟨*proof*⟩

end

10 Deflation: Continuous deflations and ep-pairs

theory *Deflation*
imports *Cfun*
begin

default-sort *cpo*

10.1 Continuous deflations

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

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

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

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

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

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

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

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

end

lemma *deflation-strict*: *deflation* $d \Longrightarrow d{\cdot}\bot = \bot$
⟨*proof*⟩

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

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

lemma *deflation-UU*: *deflation* $\bot$
⟨*proof*⟩

lemma *deflation-below-iff*:
$[\![\textit{deflation}\ p;\ \textit{deflation}\ q]\!] \Longrightarrow p \sqsubseteq q \longleftrightarrow (\forall x.\ p{\cdot}x = x \longrightarrow q{\cdot}x = x)$
⟨*proof*⟩

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

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

lemma *deflation-below-comp2*:
$[\![\textit{deflation}\ f;\ \textit{deflation}\ g;\ f \sqsubseteq g]\!] \Longrightarrow g{\cdot}(f{\cdot}x) = f{\cdot}x$
⟨*proof*⟩

10.2 Deflations with finite range

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

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

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

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

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

end

10.3 Continuous embedding-projection pairs

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

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

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

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

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

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

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

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

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

Deflations from ep-pairs

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

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

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

assumes *finite-deflation d*
shows *finite-deflation* (*e oo d oo p*)
⟨*proof*⟩

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

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

end

10.4 Uniqueness of ep-pairs

lemma *ep-pair-unique-e-lemma*:
assumes *1*: *ep-pair e1 p* **and** *2*: *ep-pair e2 p*
shows *e1* $\sqsubseteq$ *e2*
⟨*proof*⟩

lemma *ep-pair-unique-e*:
⟦*ep-pair e1 p*; *ep-pair e2 p*⟧ $\Longrightarrow$ *e1* = *e2*
⟨*proof*⟩

lemma *ep-pair-unique-p-lemma*:
assumes *1*: *ep-pair e p1* **and** *2*: *ep-pair e p2*
shows *p1* $\sqsubseteq$ *p2*
⟨*proof*⟩

lemma *ep-pair-unique-p*:
⟦*ep-pair e p1*; *ep-pair e p2*⟧ $\Longrightarrow$ *p1* = *p2*
⟨*proof*⟩

10.5 Composing ep-pairs

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

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

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

constrains $p :: 'b{::}pcpo \to 'a{::}pcpo$
begin

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

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

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

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

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

end

end

11 Bifinite: Bifinite domains and approximation

theory *Bifinite*
imports *Deflation*
begin

11.1 Omega-profinite and bifinite domains

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

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

lemma *approx-below*: $approx\ i{\cdot}x \sqsubseteq x$
⟨*proof*⟩

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

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

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

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

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

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

lemma *approx-approx1*:
$i \leq j \Longrightarrow approx\ i{\cdot}(approx\ j{\cdot}x) = approx\ i{\cdot}x$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

11.2 Instance for product type

definition
cprod-map :: $('a \to 'b) \to ('c \to 'd) \to 'a \times 'c \to 'b \times 'd$

where
cprod-map = (Λ *f g p*. (*f*·(*fst p*), *g*·(*snd p*)))

lemma *cprod-map-Pair* [*simp*]: *cprod-map*·*f*·*g*·(*x*, *y*) = (*f*·*x*, *g*·*y*)
⟨*proof*⟩

lemma *cprod-map-ID*: *cprod-map*·*ID*·*ID* = *ID*
⟨*proof*⟩

lemma *cprod-map-map*:
cprod-map·*f1*·*g1*·(*cprod-map*·*f2*·*g2*·*p*) =
cprod-map·(Λ *x*. *f1*·(*f2*·*x*))·(Λ *x*. *g1*·(*g2*·*x*))·*p*
⟨*proof*⟩

lemma *ep-pair-cprod-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*cprod-map*·*e1*·*e2*) (*cprod-map*·*p1*·*p2*)
⟨*proof*⟩

lemma *deflation-cprod-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*cprod-map*·*d1*·*d2*)
⟨*proof*⟩

lemma *finite-deflation-cprod-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*cprod-map*·*d1*·*d2*)
⟨*proof*⟩

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

definition
approx-prod-def:
approx = (λ*n*. *cprod-map*·(*approx n*)·(*approx n*))

instance ⟨*proof*⟩

end

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

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

lemma *fst-approx*: *fst* (*approx i*·*p*) = *approx i*·(*fst p*)
⟨*proof*⟩

lemma *snd-approx*: *snd* (*approx i*·*p*) = *approx i*·(*snd p*)
⟨*proof*⟩

11.3 Instance for continuous function space

definition
cfun-map :: ('*b* → '*a*) → ('*c* → '*d*) → ('*a* → '*c*) → ('*b* → '*d*)
where
cfun-map = (Λ *a b f x*. *b*·(*f*·(*a*·*x*)))

lemma *cfun-map-beta* [*simp*]: *cfun-map*·*a*·*b*·*f*·*x* = *b*·(*f*·(*a*·*x*))
⟨*proof*⟩

lemma *cfun-map-ID*: *cfun-map*·*ID*·*ID* = *ID*
⟨*proof*⟩

lemma *cfun-map-map*:
cfun-map·*f1*·*g1*·(*cfun-map*·*f2*·*g2*·*p*) =
cfun-map·(Λ *x*. *f2*·(*f1*·*x*))·(Λ *x*. *g1*·(*g2*·*x*))·*p*
⟨*proof*⟩

lemma *ep-pair-cfun-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*cfun-map*·*p1*·*e2*) (*cfun-map*·*e1*·*p2*)
⟨*proof*⟩

lemma *deflation-cfun-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*cfun-map*·*d1*·*d2*)
⟨*proof*⟩

lemma *finite-range-cfun-map*:
assumes *a*: *finite* (*range* (λ*x*. *a*·*x*))
assumes *b*: *finite* (*range* (λ*y*. *b*·*y*))
shows *finite* (*range* (λ*f*. *cfun-map*·*a*·*b*·*f*)) (**is** *finite* (*range ?h*))
⟨*proof*⟩

lemma *finite-deflation-cfun-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*cfun-map*·*d1*·*d2*)
⟨*proof*⟩

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

definition
approx-cfun-def:
approx = (λ*n*. *cfun-map*·(*approx n*)·(*approx n*))

instance ⟨*proof*⟩

end

instance *cfun* :: (*profinite*, *bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-cfun*: *approx* $n{\cdot}f{\cdot}x$ = *approx* $n{\cdot}(f{\cdot}(approx\ n{\cdot}x))$
⟨*proof*⟩

end

12 Up: The type of lifted values

theory *Up*
imports *Bifinite*
begin

default-sort *cpo*

12.1 Definition of new type for lifting

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

type-notation (*xsymbols*)
u ((-$\perp$) [*1000*] *999*)

primrec *Ifup* :: $('a \to 'b{::}pcpo) \Rightarrow 'a\ u \Rightarrow 'b$ **where**
Ifup f Ibottom = $\perp$
| *Ifup f* (*Iup x*) = $f{\cdot}x$

12.2 Ordering on lifted cpo

instantiation u :: (*cpo*) *below*
begin

definition
below-up-def:
($op \sqsubseteq$) ≡ ($\lambda x\ y.$ *case x of Ibottom* ⇒ *True* | *Iup a* ⇒
(*case y of Ibottom* ⇒ *False* | *Iup b* ⇒ $a \sqsubseteq b$))

instance ⟨*proof*⟩
end

lemma *minimal-up* [*iff*]: *Ibottom* $\sqsubseteq z$
⟨*proof*⟩

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

lemma *Iup-below* [*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 $u :: (finite\text{-}po)\ finite\text{-}po$
⟨*proof*⟩

12.4 Lifted cpo is a cpo

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

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

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

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

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

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

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

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

lemma *up-chain-lemma*:
$chain\ Y \Longrightarrow$

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

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

instance $u :: (cpo)\ cpo$
⟨*proof*⟩

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 ⇒ t == CONST fup·(Λ x. t)·l
Λ(XCONST up·x). t == CONST fup·(Λ 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-below* [*simp*]: $up{\cdot}x \sqsubseteq up{\cdot}y \longleftrightarrow x \sqsubseteq y$
⟨*proof*⟩

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

lemma *up-induct* [*case-names bottom up*, *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 ⟹
$(\exists\, A.\ chain\ A \wedge (\bigsqcup i.\ Y\ i) = up{\cdot}(\bigsqcup i.\ A\ i) \wedge$
$(\exists\, j.\ \forall\, i.\ Y\ (i + j) = up{\cdot}(A\ i))) \vee Y = (\lambda i.\ \perp)$
⟨*proof*⟩

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

lemma *compact-upD*: *compact (up·x) ⟹ 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*]: $fup{\cdot}f{\cdot}\bot = \bot$
⟨*proof*⟩

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

lemma *fup3* [*simp*]: $fup{\cdot}up{\cdot}x = x$
⟨*proof*⟩

12.8 Map function for lifted cpo

definition
$u\text{-}map :: ('a \to 'b) \to 'a\ u \to 'b\ u$
where
$u\text{-}map = (\Lambda\ f.\ fup{\cdot}(up\ oo\ f))$

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

lemma *u-map-up* [*simp*]: $u\text{-}map{\cdot}f{\cdot}(up{\cdot}x) = up{\cdot}(f{\cdot}x)$
⟨*proof*⟩

lemma *u-map-ID*: $u\text{-}map{\cdot}ID = ID$
⟨*proof*⟩

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

lemma *ep-pair-u-map*: *ep-pair e p* $\Longrightarrow$ *ep-pair* $(u\text{-}map{\cdot}e)$ $(u\text{-}map{\cdot}p)$
⟨*proof*⟩

lemma *deflation-u-map*: *deflation d* $\Longrightarrow$ *deflation* $(u\text{-}map{\cdot}d)$
⟨*proof*⟩

lemma *finite-deflation-u-map*:
assumes *finite-deflation d* **shows** *finite-deflation* $(u\text{-}map{\cdot}d)$
⟨*proof*⟩

12.9 Lifted cpo is a bifinite domain

instantiation $u :: (profinite)\ bifinite$
begin

definition
approx-up-def:
$approx = (\lambda n.\ u\text{-}map{\cdot}(approx\ n))$

instance ⟨*proof*⟩

end

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

end

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

theory *Lift*
imports *Discrete Up Countable*
begin

default-sort *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*))

13.1 Lift as a datatype

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

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

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

⊥ 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*: $(x \neq \bot) = (\exists\, y.\ x = Def\ y)$
⟨*proof*⟩

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

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

⟨*ML*⟩

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

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

lemma *Def-below-Def*: $Def\ x \sqsubseteq Def\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *Def-below-iff* [*simp*]: $Def\ x \sqsubseteq y \longleftrightarrow Def\ x = y$
⟨*proof*⟩

13.2 Lift is flat

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

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

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

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

13.3 Further operations

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

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

13.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

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

⟨*proof*⟩

lemma *cont-lift-case2*: *cont* (λ*x*. *lift-case* ⊥ *f x*)
⟨*proof*⟩

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

lemma *FLIFT-mono*:
(⋀*x*. *f x* ⊑ *g x*) ⟹ (*FLIFT x*. *f x*) ⊑ (*FLIFT x*. *g x*)
⟨*proof*⟩

lemma *cont2cont-flift1* [*simp*, *cont2cont*]:
⟦⋀*y*. *cont* (λ*x*. *f x y*)⟧ ⟹ *cont* (λ*x*. *FLIFT y*. *f x y*)
⟨*proof*⟩

lemma *cont2cont-lift-case* [*simp*]:
⟦⋀*y*. *cont* (λ*x*. *f x y*); *cont g*⟧ ⟹ *cont* (λ*x*. *lift-case UU* (*f x*) (*g x*))
⟨*proof*⟩

rewrites for *flift1*, *flift2*

lemma *flift1-Def* [*simp*]: *flift1 f*·(*Def x*) = (*f x*)
⟨*proof*⟩

lemma *flift2-Def* [*simp*]: *flift2 f*·(*Def x*) = *Def* (*f x*)
⟨*proof*⟩

lemma *flift1-strict* [*simp*]: *flift1 f*·⊥ = ⊥
⟨*proof*⟩

lemma *flift2-strict* [*simp*]: *flift2 f*·⊥ = ⊥
⟨*proof*⟩

lemma *flift2-defined* [*simp*]: *x* ≠ ⊥ ⟹ (*flift2 f*)·*x* ≠ ⊥
⟨*proof*⟩

lemma *flift2-defined-iff* [*simp*]: (*flift2 f*·*x* = ⊥) = (*x* = ⊥)
⟨*proof*⟩

13.5 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

14 Tr: The type of lifted booleans

theory *Tr*
imports *Lift*
begin

14.1 Type definition and constructors

types
tr = *bool lift*

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

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

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

Exhaustion and Elimination for type *tr*

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

lemma *trE* [*case-names bottom TT FF*]:
$[\![p = \bot \Longrightarrow Q; p = TT \Longrightarrow Q; p = FF \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *tr-induct* [*case-names bottom TT FF*]:
$[\![P\ \bot; P\ TT; P\ FF]\!] \Longrightarrow P\ x$
⟨*proof*⟩

distinctness for type *tr*

lemma *dist-below-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*⟩

lemma *TT-below-iff* [*simp*]: $TT \sqsubseteq x \longleftrightarrow x = TT$
⟨*proof*⟩

lemma *FF-below-iff* [*simp*]: $FF \sqsubseteq x \longleftrightarrow x = FF$
⟨*proof*⟩

lemma *not-below-TT-iff* [*simp*]: $\neg (x \sqsubseteq TT) \longleftrightarrow x = FF$
⟨*proof*⟩

lemma *not-below-FF-iff* [*simp*]: $\neg (x \sqsubseteq FF) \longleftrightarrow x = TT$
⟨*proof*⟩

14.2 Case analysis

default-sort *pcpo*

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

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

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

14.3 Boolean connectives

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

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

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

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

tactic for tr-thms with case split

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

lemmas about andalso, orelse, neg and if

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

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

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

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

lemma *split-If2*:
P (*If2 Q x y*) = ((*Q* = ⊥ ⟶ *P* ⊥) ∧ (*Q* = *TT* ⟶ *P x*) ∧ (*Q* = *FF* ⟶ *P y*))
⟨*proof*⟩

⟨*ML*⟩

14.4 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*:
t ≠ ⊥ ⟹ ((*t andalso s*) = *FF*) = (*t* = *FF* ∨ *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*⟩

14.5 Compactness

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

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

end

15 Ssum: The type of strict sums

theory *Ssum*
imports *Tr*
begin

default-sort *pcpo*

15.1 Definition of strict sum type

pcpodef (*Ssum*) $('a, 'b)$ *ssum* (**infixr** ++ *10*) =
$\{p :: tr \times ('a \times 'b).$
$(fst\ p \sqsubseteq TT \longleftrightarrow snd\ (snd\ p) = \bot) \wedge$
$(fst\ p \sqsubseteq FF \longleftrightarrow fst\ (snd\ p) = \bot)\}$
⟨*proof*⟩

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

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

type-notation (*xsymbols*)
ssum ((- ⊕/ -) [*21*, *20*] *20*)
type-notation (*HTML* **output**)
ssum ((- ⊕/ -) [*21*, *20*] *20*)

15.2 Definitions of constructors

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

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

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

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

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

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

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

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

15.3 Properties of *sinl* and *sinr*

Ordering

lemma *sinl-below* [*simp*]: (*sinl*·*x* ⊑ *sinl*·*y*) = (*x* ⊑ *y*)
⟨*proof*⟩

lemma *sinr-below* [*simp*]: (*sinr*·*x* ⊑ *sinr*·*y*) = (*x* ⊑ *y*)
⟨*proof*⟩

lemma *sinl-below-sinr* [*simp*]: (*sinl*·*x* ⊑ *sinr*·*y*) = (*x* = ⊥)
⟨*proof*⟩

lemma *sinr-below-sinl* [*simp*]: (*sinr*·*x* ⊑ *sinl*·*y*) = (*x* = ⊥)

⟨*proof*⟩

Equality

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

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

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

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

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

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

Strictness

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

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

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

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

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

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

Compactness

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

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

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

lemma *compact-sinrD*: *compact* (*sinr*·*x*) ⟹ *compact x*
⟨*proof*⟩

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

lemma *compact-sinr-iff* [*simp*]: *compact* (*sinr*·*x*) = *compact x*
⟨*proof*⟩

15.4 Case analysis

lemma *Exh-Ssum*:
z = ⊥ ∨ (∃ *a*. *z* = *sinl*·*a* ∧ *a* ≠ ⊥) ∨ (∃ *b*. *z* = *sinr*·*b* ∧ *b* ≠ ⊥)
⟨*proof*⟩

lemma *ssumE* [*case-names bottom sinl sinr*, *cases type*: *ssum*]:
⟦*p* = ⊥ ⟹ *Q*;
⋀*x*. ⟦*p* = *sinl*·*x*; *x* ≠ ⊥⟧ ⟹ *Q*;
⋀*y*. ⟦*p* = *sinr*·*y*; *y* ≠ ⊥⟧ ⟹ *Q*⟧ ⟹ *Q*
⟨*proof*⟩

lemma *ssum-induct* [*case-names bottom sinl sinr*, *induct type*: *ssum*]:
⟦*P* ⊥;
⋀*x*. *x* ≠ ⊥ ⟹ *P* (*sinl*·*x*);
⋀*y*. *y* ≠ ⊥ ⟹ *P* (*sinr*·*y*)⟧ ⟹ *P x*
⟨*proof*⟩

lemma *ssumE2* [*case-names sinl sinr*]:
⟦⋀*x*. *p* = *sinl*·*x* ⟹ *Q*; ⋀*y*. *p* = *sinr*·*y* ⟹ *Q*⟧ ⟹ *Q*
⟨*proof*⟩

lemma *below-sinlD*: *p* ⊑ *sinl*·*x* ⟹ ∃ *y*. *p* = *sinl*·*y* ∧ *y* ⊑ *x*
⟨*proof*⟩

lemma *below-sinrD*: *p* ⊑ *sinr*·*x* ⟹ ∃ *y*. *p* = *sinr*·*y* ∧ *y* ⊑ *x*
⟨*proof*⟩

15.5 Case analysis combinator

definition
sscase :: (′*a* → ′*c*) → (′*b* → ′*c*) → (′*a* ++ ′*b*) → ′*c* **where**
sscase = (Λ *f g s*. (λ(*t*, *x*, *y*). *If t then f*·*x else g*·*y fi*) (*Rep-Ssum s*))

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

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

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

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

15.6 Strict sum preserves flatness

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

15.7 Map function for strict sums

definition
ssum-map :: $('a \to 'b) \to ('c \to 'd) \to 'a \oplus 'c \to 'b \oplus 'd$
where
$ssum\text{-}map = (\Lambda\ f\ g.\ sscase{\cdot}(sinl\ oo\ f){\cdot}(sinr\ oo\ g))$

lemma *ssum-map-strict* [*simp*]: $ssum\text{-}map{\cdot}f{\cdot}g{\cdot}\bot = \bot$
⟨*proof*⟩

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

lemma *ssum-map-sinr* [*simp*]: $x \neq \bot \Longrightarrow ssum\text{-}map{\cdot}f{\cdot}g{\cdot}(sinr{\cdot}x) = sinr{\cdot}(g{\cdot}x)$
⟨*proof*⟩

lemma *ssum-map-sinl'*: $f{\cdot}\bot = \bot \Longrightarrow ssum\text{-}map{\cdot}f{\cdot}g{\cdot}(sinl{\cdot}x) = sinl{\cdot}(f{\cdot}x)$
⟨*proof*⟩

lemma *ssum-map-sinr'*: $g{\cdot}\bot = \bot \Longrightarrow ssum\text{-}map{\cdot}f{\cdot}g{\cdot}(sinr{\cdot}x) = sinr{\cdot}(g{\cdot}x)$
⟨*proof*⟩

lemma *ssum-map-ID*: $ssum\text{-}map{\cdot}ID{\cdot}ID = ID$
⟨*proof*⟩

lemma *ssum-map-map*:
$[\![f1{\cdot}\bot = \bot;\ g1{\cdot}\bot = \bot]\!] \Longrightarrow$

ssum-map·f1·g1·(ssum-map·f2·g2·p) =
ssum-map·(Λ x. f1·(f2·x))·(Λ x. g1·(g2·x))·p
⟨*proof*⟩

lemma *ep-pair-ssum-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*ssum-map·e1·e2*) (*ssum-map·p1·p2*)
⟨*proof*⟩

lemma *deflation-ssum-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*ssum-map·d1·d2*)
⟨*proof*⟩

lemma *finite-deflation-ssum-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*ssum-map·d1·d2*)
⟨*proof*⟩

15.8 Strict sum is a bifinite domain

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

definition
approx-ssum-def:
approx = (λ*n*. *ssum-map·*(*approx n*)·(*approx n*))

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

16 Sprod: The type of strict products

theory *Sprod*
imports *Bifinite*
begin

default-sort *pcpo*

16.1 Definition of strict product type

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

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

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

type-notation (*xsymbols*)
sprod ((- ⊗/ -) [*21*,*20*] *20*)
type-notation (*HTML* **output**)
sprod ((- ⊗/ -) [*21*,*20*] *20*)

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

16.2 Definitions of constants

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

definition
ssnd :: ('*a* ** '*b*) → '*b* **where**
ssnd = (Λ *p*. *snd* (*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*)

16.3 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*] *below-Sprod-def*
Rep-Sprod-strict Rep-Sprod-spair

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

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

lemma *sprod-induct* [*case-names bottom spair*, *induct type*: *sprod*]:
$[\![P\ \bot;\ \bigwedge x\ y.\ [\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow P\ (:x,\ y:)]\!] \Longrightarrow P\ x$
⟨*proof*⟩

16.4 Properties of *spair*

lemma *spair-strict1* [*simp*]: (:⊥, y:) = ⊥
⟨*proof*⟩

lemma *spair-strict2* [*simp*]: (:x, ⊥:) = ⊥
⟨*proof*⟩

lemma *spair-strict-iff* [*simp*]: ((:x, y:) = ⊥) = ($x = \bot \vee y = \bot$)
⟨*proof*⟩

lemma *spair-below-iff*:
((:a, b:) ⊑ (: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:) = ⊥
⟨*proof*⟩

lemma *spair-strict-rev*: (:x, y:) ≠ ⊥ ⟹ $x \neq \bot \wedge y \neq \bot$
⟨*proof*⟩

lemma *spair-defined*: $[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow$ (:x, y:) ≠ ⊥
⟨*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*⟩

lemma *sprodE2*: $(\bigwedge x\ y.\ p = (:x,\ y:) \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

16.5 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 *below-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-below*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) = (x \sqsubseteq a \wedge y \sqsubseteq b)$
⟨*proof*⟩

lemma *sfst-below-iff*: $sfst{\cdot}x \sqsubseteq y = x \sqsubseteq (:y,\ ssnd{\cdot}x:)$
⟨*proof*⟩

lemma *ssnd-below-iff*: $ssnd{\cdot}x \sqsubseteq y = x \sqsubseteq (:sfst{\cdot}x,\ y:)$
⟨*proof*⟩

16.6 Compactness

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

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

lemma *compact-spair*: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact\ (:x,\ y:)$
⟨*proof*⟩

lemma *compact-spair-iff*:
$compact\ (:x,\ y:) = (x = \bot \vee y = \bot \vee (compact\ x \wedge compact\ y))$
⟨*proof*⟩

16.7 Properties of *ssplit*

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

lemma *ssplit2* [*simp*]: $[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow ssplit{\cdot}f{\cdot}(:x,\ y:) = f{\cdot}x{\cdot}y$
⟨*proof*⟩

lemma *ssplit3* [*simp*]: $ssplit{\cdot}spair{\cdot}z = z$
⟨*proof*⟩

16.8 Strict product preserves flatness

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

16.9 Map function for strict products

definition
sprod-map :: $('a \to 'b) \to ('c \to 'd) \to 'a \otimes 'c \to 'b \otimes 'd$
where

sprod-map = (Λ *f g*. *ssplit*·(Λ *x y*. (:*f*·*x*, *g*·*y*:)))

lemma *sprod-map-strict* [*simp*]: *sprod-map*·*a*·*b*·⊥ = ⊥
⟨*proof*⟩

lemma *sprod-map-spair* [*simp*]:
x ≠ ⊥ ⟹ *y* ≠ ⊥ ⟹ *sprod-map*·*f*·*g*·(:*x*, *y*:) = (:*f*·*x*, *g*·*y*:)
⟨*proof*⟩

lemma *sprod-map-spair'*:
f·⊥ = ⊥ ⟹ *g*·⊥ = ⊥ ⟹ *sprod-map*·*f*·*g*·(:*x*, *y*:) = (:*f*·*x*, *g*·*y*:)
⟨*proof*⟩

lemma *sprod-map-ID*: *sprod-map*·*ID*·*ID* = *ID*
⟨*proof*⟩

lemma *sprod-map-map*:
⟦*f1*·⊥ = ⊥; *g1*·⊥ = ⊥⟧ ⟹
sprod-map·*f1*·*g1*·(*sprod-map*·*f2*·*g2*·*p*) =
sprod-map·(Λ *x*. *f1*·(*f2*·*x*))·(Λ *x*. *g1*·(*g2*·*x*))·*p*
⟨*proof*⟩

lemma *ep-pair-sprod-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*sprod-map*·*e1*·*e2*) (*sprod-map*·*p1*·*p2*)
⟨*proof*⟩

lemma *deflation-sprod-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*sprod-map*·*d1*·*d2*)
⟨*proof*⟩

lemma *finite-deflation-sprod-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*sprod-map*·*d1*·*d2*)
⟨*proof*⟩

16.10 Strict product is a bifinite domain

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

definition
approx-sprod-def:
approx = (λ*n*. *sprod-map*·(*approx n*)·(*approx n*))

instance ⟨*proof*⟩

end

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

end

17 One: The unit domain

theory *One*
imports *Lift*
begin

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

definition
ONE :: one
where
ONE == Def ()

Exhaustion and Elimination for type *one*

lemma *Exh-one*: $t = \bot \vee t = ONE$
⟨*proof*⟩

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

lemma *one-induct* [*case-names bottom ONE*]: $[\![P\ \bot;\ P\ ONE]\!] \Longrightarrow P\ x$
⟨*proof*⟩

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

lemma *below-ONE* [*simp*]: $x \sqsubseteq ONE$
⟨*proof*⟩

lemma *ONE-below-iff* [*simp*]: $ONE \sqsubseteq x \longleftrightarrow x = ONE$
⟨*proof*⟩

lemma *ONE-defined* [*simp*]: $ONE \neq \bot$
⟨*proof*⟩

lemma *one-neq-iffs* [*simp*]:
$x \neq ONE \longleftrightarrow x = \bot$
$ONE \neq x \longleftrightarrow x = \bot$
$x \neq \bot \longleftrightarrow x = ONE$

$\bot \neq x \longleftrightarrow x = ONE$
⟨*proof*⟩

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

Case analysis function for type *one*

definition
one-when :: *'a::pcpo* → *one* → *'a* **where**
one-when = (Λ *a*. *strictify*·(Λ -. *a*))

translations
case x of XCONST ONE ⇒ *t* == *CONST one-when*·*t*·*x*
Λ (*XCONST ONE*). *t* == *CONST one-when*·*t*

lemma *one-when1* [*simp*]: (*case* ⊥ *of ONE* ⇒ *t*) = ⊥
⟨*proof*⟩

lemma *one-when2* [*simp*]: (*case ONE of ONE* ⇒ *t*) = *t*
⟨*proof*⟩

lemma *one-when3* [*simp*]: (*case x of ONE* ⇒ *ONE*) = *x*
⟨*proof*⟩

end

18 Cprod: The cpo of cartesian products

theory *Cprod*
imports *Bifinite*
begin

default-sort *cpo*

18.1 Continuous case function for unit type

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

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

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

18.2 Continuous version of split function

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

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

18.3 Convert all lemmas to the continuous versions

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

lemma *csplit-Pair* [*simp*]: $csplit{\cdot}f{\cdot}(x,\ y) = f{\cdot}x{\cdot}y$
⟨*proof*⟩

end

19 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun*
begin

default-sort *pcpo*

19.1 Iteration

primrec *iterate* :: $nat \Rightarrow ('a{::}cpo \to 'a) \to ('a \to 'a)$ **where**
  $iterate\ 0 = (\Lambda\ F\ x.\ x)$
| $iterate\ (Suc\ n) = (\Lambda\ F\ x.\ F{\cdot}(iterate\ n{\cdot}F{\cdot}x))$

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*⟩

lemma *iterate-iterate*:
  $iterate\ m{\cdot}F{\cdot}(iterate\ n{\cdot}F{\cdot}x) = iterate\ (m + n){\cdot}F{\cdot}x$
⟨*proof*⟩

The sequence of function iterations is a chain.

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

19.2 Least fixed point operator

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

Binder syntax for *fix*

abbreviation
fix-syn :: $('a \Rightarrow 'a) \Rightarrow 'a$ (**binder** *FIX* 10) **where**
fix-syn ($\lambda x.$ f x) $\equiv$ *fix*$\cdot$(Λ $x.$ f x)

notation (*xsymbols*)
fix-syn (**binder** μ 10)

Properties of *fix*

direct connection between *fix* and iteration

lemma *fix-def2*: *fix*$\cdot F$ = ($\bigsqcup i.$ *iterate* $i{\cdot}F{\cdot}\bot$)
⟨*proof*⟩

lemma *iterate-below-fix*: *iterate* $n{\cdot}f{\cdot}\bot \sqsubseteq$ *fix*$\cdot f$
⟨*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-below*: $F{\cdot}x \sqsubseteq x \Longrightarrow$ *fix*$\cdot F \sqsubseteq x$
⟨*proof*⟩

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

lemma *fix-eqI*:
assumes *fixed*: $F{\cdot}x = x$ **and** *least*: $\bigwedge z.\ F{\cdot}z = z \Longrightarrow x \sqsubseteq z$
shows *fix*$\cdot F = x$
⟨*proof*⟩

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

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

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

19.3 Fixed point induction

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

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

lemma *fix-ind2*:
assumes *adm*: *adm P*
assumes *0*: $P\ \bot$ **and** *1*: $P\ (F{\cdot}\bot)$
assumes *step*: $\bigwedge x.\ [\![P\ x;\ P\ (F{\cdot}x)]\!] \Longrightarrow P\ (F{\cdot}(F{\cdot}x))$
shows $P\ (\mathit{fix}{\cdot}F)$
⟨*proof*⟩

lemma *parallel-fix-ind*:
assumes *adm*: *adm* $(\lambda x.\ P\ (\mathit{fst}\ x)\ (\mathit{snd}\ x))$
assumes *base*: $P\ \bot\ \bot$
assumes *step*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (F{\cdot}x)\ (G{\cdot}y)$
shows $P\ (\mathit{fix}{\cdot}F)\ (\mathit{fix}{\cdot}G)$
⟨*proof*⟩

19.4 Fixed-points on product types

Bekic's Theorem: Simultaneous fixed points over pairs can be written in terms of separate fixed points.

lemma *fix-cprod*:
fix·(*F*::'*a* × '*b* → '*a* × '*b*) =
(μ *x*. *fst* (*F*·(*x*, μ *y*. *snd* (*F*·(*x*, *y*)))),
μ *y*. *snd* (*F*·(μ *x*. *fst* (*F*·(*x*, μ *y*. *snd* (*F*·(*x*, *y*)))), *y*)))
(**is** *fix*·*F* = (*?x*, *?y*))
⟨*proof*⟩

end

20 Fixrec: Package for defining recursive functions in HOLCF

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

20.1 Pattern-match monad

default-sort *cpo*

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

definition
fail :: '*a match* **where**
fail = *Abs-match* (*sinl*·*ONE*)

definition
succeed :: '*a* → '*a match* **where**
succeed = (Λ *x*. *Abs-match* (*sinr*·(*up*·*x*)))

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

lemma *matchE* [*case-names bottom fail succeed*, *cases type*: *match*]:
⟦*p* = ⊥ ⟹ *Q*; *p* = *fail* ⟹ *Q*; ⋀*x*. *p* = *succeed*·*x* ⟹ *Q*⟧ ⟹ *Q*
⟨*proof*⟩

lemma *succeed-defined* [*simp*]: *succeed*·*x* ≠ ⊥

⟨*proof*⟩

lemma *fail-defined* [*simp*]: *fail* ≠ ⊥
⟨*proof*⟩

lemma *succeed-eq* [*simp*]: (*succeed*·*x* = *succeed*·*y*) = (*x* = *y*)
⟨*proof*⟩

lemma *succeed-neq-fail* [*simp*]:
succeed·*x* ≠ *fail* *fail* ≠ *succeed*·*x*
⟨*proof*⟩

lemma *match-case-simps* [*simp*]:
match-case·*f*·*r*·⊥ = ⊥
match-case·*f*·*r*·*fail* = *f*
match-case·*f*·*r*·(*succeed*·*x*) = *r*·*x*
⟨*proof*⟩

translations
case m of XCONST fail ⇒ *t1* | *XCONST succeed*·*x* ⇒ *t2*
== *CONST match-case*·*t1*·(Λ *x*. *t2*)·*m*

20.1.1 Run operator

definition
run :: *'a match* → *'a*::*pcpo* **where**
run = *match-case*·⊥·*ID*

rewrite rules for run

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

lemma *run-fail* [*simp*]: *run*·*fail* = ⊥
⟨*proof*⟩

lemma *run-succeed* [*simp*]: *run*·(*succeed*·*x*) = *x*
⟨*proof*⟩

20.1.2 Monad plus operator

definition
mplus :: *'a match* → *'a match* → *'a match* **where**
mplus = (Λ *m1 m2*. *case m1 of fail* ⇒ *m2* | *succeed*·*x* ⇒ *m1*)

abbreviation
mplus-syn :: [*'a match*, *'a match*] ⇒ *'a match* (**infixr** +++ *65*) **where**
m1 +++ *m2* == *mplus*·*m1*·*m2*

rewrite rules for mplus

lemma *mplus-strict* [*simp*]: $\bot$ +++ $m = \bot$
⟨*proof*⟩

lemma *mplus-fail* [*simp*]: *fail* +++ $m = m$
⟨*proof*⟩

lemma *mplus-succeed* [*simp*]: *succeed*·x +++ m = *succeed*·x
⟨*proof*⟩

lemma *mplus-fail2* [*simp*]: m +++ *fail* = m
⟨*proof*⟩

lemma *mplus-assoc*: (x +++ y) +++ z = x +++ (y +++ z)
⟨*proof*⟩

20.2 Match functions for built-in types

default-sort *pcpo*

definition
match-UU :: $'a \to 'c$ *match* $\to 'c$ *match*
where
match-UU = *strictify*·(Λ x k. *fail*)

definition
match-cpair :: $'a$::*cpo* × $'b$::*cpo* $\to$ ($'a \to 'b \to 'c$ *match*) $\to 'c$ *match*
where
match-cpair = (Λ x k. *csplit*·k·x)

definition
match-spair :: $'a \otimes 'b \to$ ($'a \to 'b \to 'c$ *match*) $\to 'c$ *match*
where
match-spair = (Λ x k. *ssplit*·k·x)

definition
match-sinl :: $'a \oplus 'b \to$ ($'a \to 'c$ *match*) $\to 'c$ *match*
where
match-sinl = (Λ x k. *sscase*·k·(Λ b. *fail*)·x)

definition
match-sinr :: $'a \oplus 'b \to$ ($'b \to 'c$ *match*) $\to 'c$ *match*
where
match-sinr = (Λ x k. *sscase*·(Λ a. *fail*)·k·x)

definition
match-up :: $'a$::*cpo* $u \to$ ($'a \to 'c$ *match*) $\to 'c$ *match*
where
match-up = (Λ x k. *fup*·k·x)

definition
  match-ONE :: *one* → *'c match* → *'c match*
where
  match-ONE = (Λ *ONE k*. *k*)

definition
  match-TT :: *tr* → *'c match* → *'c match*
where
  match-TT = (Λ *x k*. *If x then k else fail fi*)

definition
  match-FF :: *tr* → *'c match* → *'c match*
where
  match-FF = (Λ *x k*. *If x then fail else k fi*)

lemma *match-UU-simps* [*simp*]:
  match-UU·⊥·*k* = ⊥
  x ≠ ⊥ ⟹ *match-UU*·*x*·*k* = *fail*
⟨*proof*⟩

lemma *match-cpair-simps* [*simp*]:
  match-cpair·(*x*, *y*)·*k* = *k*·*x*·*y*
⟨*proof*⟩

lemma *match-spair-simps* [*simp*]:
  ⟦*x* ≠ ⊥; *y* ≠ ⊥⟧ ⟹ *match-spair*·(:*x*, *y*:)·*k* = *k*·*x*·*y*
  match-spair·⊥·*k* = ⊥
⟨*proof*⟩

lemma *match-sinl-simps* [*simp*]:
  x ≠ ⊥ ⟹ *match-sinl*·(*sinl*·*x*)·*k* = *k*·*x*
  y ≠ ⊥ ⟹ *match-sinl*·(*sinr*·*y*)·*k* = *fail*
  match-sinl·⊥·*k* = ⊥
⟨*proof*⟩

lemma *match-sinr-simps* [*simp*]:
  x ≠ ⊥ ⟹ *match-sinr*·(*sinl*·*x*)·*k* = *fail*
  y ≠ ⊥ ⟹ *match-sinr*·(*sinr*·*y*)·*k* = *k*·*y*
  match-sinr·⊥·*k* = ⊥
⟨*proof*⟩

lemma *match-up-simps* [*simp*]:
  match-up·(*up*·*x*)·*k* = *k*·*x*
  match-up·⊥·*k* = ⊥
⟨*proof*⟩

lemma *match-ONE-simps* [*simp*]:
  match-ONE·*ONE*·*k* = *k*
  match-ONE·⊥·*k* = ⊥

⟨*proof*⟩

lemma *match-TT-simps* [*simp*]:
match-TT·*TT*·*k* = *k*
match-TT·*FF*·*k* = *fail*
match-TT·⊥·*k* = ⊥
⟨*proof*⟩

lemma *match-FF-simps* [*simp*]:
match-FF·*FF*·*k* = *k*
match-FF·*TT*·*k* = *fail*
match-FF·⊥·*k* = ⊥
⟨*proof*⟩

20.3 Mutual recursion

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

lemma *Pair-equalI*: ⟦*x* ≡ *fst p*; *y* ≡ *snd p*⟧ ⟹ (*x*, *y*) ≡ *p*
⟨*proof*⟩

lemma *Pair-eqD1*: (*x*, *y*) = (*x*′, *y*′) ⟹ *x* = *x*′
⟨*proof*⟩

lemma *Pair-eqD2*: (*x*, *y*) = (*x*′, *y*′) ⟹ *y* = *y*′
⟨*proof*⟩

lemma *def-cont-fix-eq*:
⟦*f* ≡ *fix*·(*Abs-CFun F*); *cont F*⟧ ⟹ *f* = *F f*
⟨*proof*⟩

lemma *def-cont-fix-ind*:
⟦*f* ≡ *fix*·(*Abs-CFun F*); *cont F*; *adm P*; *P* ⊥; ⋀*x*. *P x* ⟹ *P* (*F x*)⟧ ⟹ *P f*
⟨*proof*⟩

lemma for proving rewrite rules

lemma *ssubst-lhs*: ⟦*t* = *s*; *P s* = *Q*⟧ ⟹ *P t* = *Q*
⟨*proof*⟩

20.4 Initializing the fixrec package

⟨*ML*⟩

hide-const (**open**) *succeed fail run*

end

21 Completion: Defining bifinite domains by ideal completion

theory *Completion*
imports *Bifinite*
begin

21.1 Ideals over a preorder

locale *preorder* =
  fixes $r :: 'a::type \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *50*)
  assumes *r-refl*: $x \preceq x$
  assumes *r-trans*: $\llbracket x \preceq y; y \preceq z \rrbracket \Longrightarrow x \preceq z$
begin

definition
  ideal :: $'a\ set \Rightarrow bool$ **where**
  $ideal\ A = ((\exists x.\ x \in A) \wedge (\forall x{\in}A.\ \forall y{\in}A.\ \exists z{\in}A.\ x \preceq z \wedge y \preceq z) \wedge$
    $(\forall x\ y.\ x \preceq y \longrightarrow y \in A \longrightarrow x \in A))$

lemma *idealI*:
  assumes $\exists x.\ x \in A$
  assumes $\bigwedge x\ y.\ \llbracket x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
  assumes $\bigwedge x\ y.\ \llbracket x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
  shows *ideal A*
⟨*proof*⟩

lemma *idealD1*:
  $ideal\ A \Longrightarrow \exists x.\ x \in A$
⟨*proof*⟩

lemma *idealD2*:
  $\llbracket ideal\ A; x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
⟨*proof*⟩

lemma *idealD3*:
  $\llbracket ideal\ A; x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
⟨*proof*⟩

lemma *ideal-directed-finite*:
  assumes *A*: *ideal A*
  shows $\llbracket finite\ U;\ U \subseteq A \rrbracket \Longrightarrow \exists z{\in}A.\ \forall x{\in}U.\ x \preceq z$
⟨*proof*⟩

lemma *ideal-principal*: *ideal* $\{x.\ x \preceq z\}$
⟨*proof*⟩

lemma *ex-ideal*: $\exists A.\ ideal\ A$
⟨*proof*⟩

lemma *directed-image-ideal*:
assumes *A*: *ideal A*
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows *directed* (*f* ' *A*)
⟨*proof*⟩

lemma *lub-image-principal*:
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows $(\bigsqcup x{\in}\{x.\ x \preceq y\}.\ f\ x) = f\ y$
⟨*proof*⟩

The set of ideals is a cpo

lemma *ideal-UN*:
fixes *A* :: *nat* ⇒ *'a set*
assumes *ideal-A*: $\bigwedge i.$ *ideal* (*A i*)
assumes *chain-A*: $\bigwedge i\ j.\ i \leq j \Longrightarrow A\ i \subseteq A\ j$
shows *ideal* $(\bigcup i.\ A\ i)$
⟨*proof*⟩

lemma *typedef-ideal-po*:
fixes *Abs* :: *'a set* ⇒ *'b*::*below*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *below*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow$ *Rep x* ⊆ *Rep y*
shows *OFCLASS*(*'b*, *po-class*)
⟨*proof*⟩

lemma
fixes *Abs* :: *'a set* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *below*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow$ *Rep x* ⊆ *Rep y*
assumes *S*: *chain S*
shows *typedef-ideal-lub*: *range S* <<| *Abs* $(\bigcup i.$ *Rep* (*S i*))
and *typedef-ideal-rep-contlub*: *Rep* $(\bigsqcup i.\ S\ i) = (\bigcup i.$ *Rep* (*S i*))
⟨*proof*⟩

lemma *typedef-ideal-cpo*:
fixes *Abs* :: *'a set* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *below*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow$ *Rep x* ⊆ *Rep y*
shows *OFCLASS*(*'b*, *cpo-class*)
⟨*proof*⟩

end

interpretation *below*: *preorder below* :: *'a*::*po* ⇒ *'a* ⇒ *bool*
⟨*proof*⟩

21.2 Lemmas about least upper bounds

lemma *finite-directed-contains-lub*:
$[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \exists\, u{\in}S.\ S <<|\ u$
⟨*proof*⟩

lemma *lub-finite-directed-in-self*:
$[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \mathit{lub}\ S \in S$
⟨*proof*⟩

lemma *finite-directed-has-lub*: $[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \exists\, u.\ S <<|\ u$
⟨*proof*⟩

lemma *is-ub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ x \in S]\!] \Longrightarrow x \sqsubseteq \mathit{lub}\ S$
⟨*proof*⟩

lemma *is-lub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ S <|\ x]\!] \Longrightarrow \mathit{lub}\ S \sqsubseteq x$
⟨*proof*⟩

21.3 Locale for ideal completion

locale *basis-take* = *preorder* +
  fixes *take* :: *nat* ⇒ *'a*::*type* ⇒ *'a*
  assumes *take-less*: $\mathit{take}\ n\ a \preceq a$
  assumes *take-take*: $\mathit{take}\ n\ (\mathit{take}\ n\ a) = \mathit{take}\ n\ a$
  assumes *take-mono*: $a \preceq b \Longrightarrow \mathit{take}\ n\ a \preceq \mathit{take}\ n\ b$
  assumes *take-chain*: $\mathit{take}\ n\ a \preceq \mathit{take}\ (\mathit{Suc}\ n)\ a$
  assumes *finite-range-take*: *finite* (*range* (*take n*))
  assumes *take-covers*: $\exists\, n.\ \mathit{take}\ n\ a = a$
begin

lemma *take-chain-less*: $m < n \Longrightarrow \mathit{take}\ m\ a \preceq \mathit{take}\ n\ a$
⟨*proof*⟩

lemma *take-chain-le*: $m \leq n \Longrightarrow \mathit{take}\ m\ a \preceq \mathit{take}\ n\ a$
⟨*proof*⟩

end

locale *ideal-completion* = *basis-take* +
  fixes *principal* :: *'a*::*type* ⇒ *'b*::*cpo*
  fixes *rep* :: *'b*::*cpo* ⇒ *'a*::*type set*
  assumes *ideal-rep*: $\bigwedge x.\ \mathit{preorder.ideal}\ r\ (\mathit{rep}\ x)$
  assumes *rep-contlub*: $\bigwedge Y.\ \mathit{chain}\ Y \Longrightarrow \mathit{rep}\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ \mathit{rep}\ (Y\ i))$
  assumes *rep-principal*: $\bigwedge a.\ \mathit{rep}\ (\mathit{principal}\ a) = \{b.\ b \preceq a\}$
  assumes *subset-repD*: $\bigwedge x\ y.\ \mathit{rep}\ x \subseteq \mathit{rep}\ y \Longrightarrow x \sqsubseteq y$
begin

lemma *finite-take-rep*: *finite* (*take n* ‘ *rep x*)
⟨*proof*⟩

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

lemma *below-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-below*: $a \in rep\ x \longleftrightarrow principal\ a \sqsubseteq x$
⟨*proof*⟩

lemma *principal-below-iff-mem-rep*: $principal\ a \sqsubseteq x \longleftrightarrow a \in rep\ x$
⟨*proof*⟩

lemma *principal-below-iff* [*simp*]: $principal\ a \sqsubseteq principal\ b \longleftrightarrow a \preceq b$
⟨*proof*⟩

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

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

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

lemma *belowI*: $(\bigwedge a.\ principal\ a \sqsubseteq x \Longrightarrow principal\ a \sqsubseteq u) \Longrightarrow x \sqsubseteq u$
⟨*proof*⟩

lemma *lub-principal-rep*: $principal\ `\ rep\ x <<|\ x$
⟨*proof*⟩

21.4 Defining functions in terms of basis elements

definition
basis-fun :: $('a::type \Rightarrow 'c::cpo) \Rightarrow 'b \to 'c$ **where**
$basis\text{-}fun = (\lambda f.\ (\Lambda\ x.\ lub\ (f\ `\ rep\ x)))$

lemma *basis-fun-lemma0*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $\exists u.\ f\ `\ take\ i\ `\ rep\ x <<|\ u$
⟨*proof*⟩

lemma *basis-fun-lemma1*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$

shows *chain* (λ*i*. *lub* (*f* ‘ *take i* ‘ *rep x*))
⟨*proof*⟩

lemma *basis-fun-lemma2*:
fixes *f* :: ′*a*::*type* ⇒ ′*c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows *f* ‘ *rep x* <<| (⨆ *i*. *lub* (*f* ‘ *take i* ‘ *rep x*))
⟨*proof*⟩

lemma *basis-fun-lemma*:
fixes *f* :: ′*a*::*type* ⇒ ′*c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows ∃ *u*. *f* ‘ *rep x* <<| *u*
⟨*proof*⟩

lemma *basis-fun-beta*:
fixes *f* :: ′*a*::*type* ⇒ ′*c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows *basis-fun f*·*x* = *lub* (*f* ‘ *rep x*)
⟨*proof*⟩

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

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

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

21.5 Bifiniteness of ideal completions

definition
completion-approx :: *nat* ⇒ ′*b* → ′*b* **where**
completion-approx = (λ*i*. *basis-fun* (λ*a*. *principal* (*take i a*)))

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

lemma *completion-approx-principal*:
completion-approx i·(*principal a*) = *principal* (*take i a*)

⟨*proof*⟩

lemma *chain-completion-approx*: *chain completion-approx*
⟨*proof*⟩

lemma *lub-completion-approx*: $(\bigsqcup i.\ \textit{completion-approx}\ i{\cdot}x) = x$
⟨*proof*⟩

lemma *completion-approx-eq-principal*:
$\exists\, a{\in}\textit{rep}\ x.\ \textit{completion-approx}\ i{\cdot}x = \textit{principal}\ (\textit{take}\ i\ a)$
⟨*proof*⟩

lemma *completion-approx-idem*:
$\textit{completion-approx}\ i{\cdot}(\textit{completion-approx}\ i{\cdot}x) = \textit{completion-approx}\ i{\cdot}x$
⟨*proof*⟩

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

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes *P*: $\bigwedge a.\ P\ (\textit{principal}\ a)$
shows *P x*
⟨*proof*⟩

lemma *principal-induct2*:
$[\![\bigwedge y.\ \textit{adm}\ (\lambda x.\ P\ x\ y);\ \bigwedge x.\ \textit{adm}\ (\lambda y.\ P\ x\ y);$
$\bigwedge a\ b.\ P\ (\textit{principal}\ a)\ (\textit{principal}\ b)]\!] \Longrightarrow P\ x\ y$
⟨*proof*⟩

lemma *compact-imp-principal*: $\textit{compact}\ x \Longrightarrow \exists\, a.\ x = \textit{principal}\ a$
⟨*proof*⟩

end

end

22 Eventual: Eventually-constant sequences

theory *Eventual*
imports *Infinite-Set*
begin

22.1 Lemmas about MOST

lemma *MOST-INFM*:
assumes *inf*: *infinite* $(\textit{UNIV}::'a\ \textit{set})$

shows $MOST\ x{::}'a.\ P\ x \Longrightarrow INFM\ x{::}'a.\ P\ x$
⟨*proof*⟩

lemma *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
⟨*proof*⟩

lemma *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
⟨*proof*⟩

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
⟨*proof*⟩

lemma *INFM-finite-Bex-distrib*:
$finite\ A \Longrightarrow (INFM\ y.\ \exists x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists x{\in}A.\ INFM\ y.\ P\ x\ y)$
⟨*proof*⟩

lemma *MOST-finite-Ball-distrib*:
$finite\ A \Longrightarrow (MOST\ y.\ \forall x{\in}A.\ P\ x\ y) \longleftrightarrow (\forall x{\in}A.\ MOST\ y.\ P\ x\ y)$
⟨*proof*⟩

lemma *MOST-ge-nat*: $MOST\ n{::}nat.\ m \leq n$
⟨*proof*⟩

22.2 Eventually constant sequences

definition
$eventually\text{-}constant :: (nat \Rightarrow 'a) \Rightarrow bool$
where
$eventually\text{-}constant\ S = (\exists x.\ MOST\ i.\ S\ i = x)$

lemma *eventually-constant-MOST-MOST*:
$eventually\text{-}constant\ S \longleftrightarrow (MOST\ m.\ MOST\ n.\ S\ n = S\ m)$
⟨*proof*⟩

lemma *eventually-constantI*: $MOST\ i.\ S\ i = x \Longrightarrow eventually\text{-}constant\ S$
⟨*proof*⟩

lemma *eventually-constant-comp*:
$eventually\text{-}constant\ (\lambda i.\ S\ i) \Longrightarrow eventually\text{-}constant\ (\lambda i.\ f\ (S\ i))$
⟨*proof*⟩

lemma *eventually-constant-Suc-iff*:
$eventually\text{-}constant\ (\lambda i.\ S\ (Suc\ i)) \longleftrightarrow eventually\text{-}constant\ (\lambda i.\ S\ i)$
⟨*proof*⟩

lemma *eventually-constant-SucD*:
$eventually\text{-}constant\ (\lambda i.\ S\ (Suc\ i)) \Longrightarrow eventually\text{-}constant\ (\lambda i.\ S\ i)$
⟨*proof*⟩

22.3 Limits of eventually constant sequences

definition
eventual :: $(nat \Rightarrow 'a) \Rightarrow 'a$ **where**
eventual $S = (THE\ x.\ MOST\ i.\ S\ i = x)$

lemma *eventual-eqI*: $MOST\ i.\ S\ i = x \Longrightarrow eventual\ S = x$
⟨*proof*⟩

lemma *MOST-eq-eventual*:
eventually-constant $S \Longrightarrow MOST\ i.\ S\ i = eventual\ S$
⟨*proof*⟩

lemma *eventual-mem-range*:
eventually-constant $S \Longrightarrow eventual\ S \in range\ S$
⟨*proof*⟩

lemma *eventually-constant-MOST-iff*:
assumes S: *eventually-constant* S
shows $(MOST\ n.\ P\ (S\ n)) \longleftrightarrow P\ (eventual\ S)$
⟨*proof*⟩

lemma *MOST-eventual*:
$[\![eventually\text{-}constant\ S;\ MOST\ n.\ P\ (S\ n)]\!] \Longrightarrow P\ (eventual\ S)$
⟨*proof*⟩

lemma *eventually-constant-MOST-Suc-eq*:
eventually-constant $S \Longrightarrow MOST\ n.\ S\ (Suc\ n) = S\ n$
⟨*proof*⟩

lemma *eventual-comp*:
eventually-constant $S \Longrightarrow eventual\ (\lambda i.\ f\ (S\ i)) = f\ (eventual\ (\lambda i.\ S\ i))$
⟨*proof*⟩

end

23 Algebraic: Algebraic deflations

theory *Algebraic*
imports *Completion Fix Eventual*
begin

23.1 Constructing finite deflations by iteration

lemma *finite-deflation-imp-deflation*:
finite-deflation $d \Longrightarrow$ *deflation* d
⟨*proof*⟩

lemma *le-Suc-induct*:
assumes *le*: $i \leq j$
assumes *step*: $\bigwedge i.\ P\ i\ (Suc\ i)$
assumes *refl*: $\bigwedge i.\ P\ i\ i$
assumes *trans*: $\bigwedge i\ j\ k.\ [\![P\ i\ j;\ P\ j\ k]\!] \Longrightarrow P\ i\ k$
shows $P\ i\ j$
⟨*proof*⟩

definition
eventual-iterate :: $('a \to 'a{::}cpo) \Rightarrow ('a \to 'a)$
where
eventual-iterate f = *eventual* $(\lambda n.\ iterate\ n{\cdot}f)$

A pre-deflation is like a deflation, but not idempotent.

locale *pre-deflation* =
fixes $f :: 'a \to 'a{::}cpo$
assumes *below*: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
assumes *finite-range*: *finite* (*range* $(\lambda x.\ f{\cdot}x)$)
begin

lemma *iterate-below*: $iterate\ i{\cdot}f{\cdot}x \sqsubseteq x$
⟨*proof*⟩

lemma *iterate-fixed*: $f{\cdot}x = x \Longrightarrow iterate\ i{\cdot}f{\cdot}x = x$
⟨*proof*⟩

lemma *antichain-iterate-app*: $i \leq j \Longrightarrow iterate\ j{\cdot}f{\cdot}x \sqsubseteq iterate\ i{\cdot}f{\cdot}x$
⟨*proof*⟩

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

lemma *eventually-constant-iterate-app*:
eventually-constant $(\lambda i.\ iterate\ i{\cdot}f{\cdot}x)$
⟨*proof*⟩

lemma *eventually-constant-iterate*:
eventually-constant $(\lambda n.\ iterate\ n{\cdot}f)$
⟨*proof*⟩

abbreviation
$d :: 'a \to 'a$
where
$d \equiv$ *eventual-iterate* f

lemma *MOST-d*: $MOST\ n.\ P\ (iterate\ n{\cdot}f) \Longrightarrow P\ d$
⟨*proof*⟩

lemma *f-d*: $f{\cdot}(d{\cdot}x) = d{\cdot}x$

⟨*proof*⟩

lemma *d-fixed-iff*: $d{\cdot}x = x \longleftrightarrow f{\cdot}x = x$
⟨*proof*⟩

lemma *finite-deflation-d*: *finite-deflation d*
⟨*proof*⟩

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

end

lemma *finite-deflation-eventual-iterate*:
pre-deflation d $\Longrightarrow$ *finite-deflation* (*eventual-iterate d*)
⟨*proof*⟩

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

lemma *eventual-iterate-oo-fixed-iff*:
assumes *finite-deflation d*
assumes *f*: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
shows *eventual-iterate* $(d \mathbin{oo} f){\cdot}x = x \longleftrightarrow d{\cdot}x = x \wedge f{\cdot}x = x$
⟨*proof*⟩

lemma *eventual-mono*:
assumes *A*: *eventually-constant A*
assumes *B*: *eventually-constant B*
assumes *below*: $\bigwedge n.\ A\ n \sqsubseteq B\ n$
shows *eventual A* $\sqsubseteq$ *eventual B*
⟨*proof*⟩

lemma *eventual-iterate-mono*:
assumes *f*: *pre-deflation f* **and** *g*: *pre-deflation g* **and** $f \sqsubseteq g$
shows *eventual-iterate f* $\sqsubseteq$ *eventual-iterate g*
⟨*proof*⟩

lemma *cont2cont-eventual-iterate-oo*:
assumes *d*: *finite-deflation d*
assumes *cont*: *cont f* **and** *below*: $\bigwedge x\ y.\ f\ x{\cdot}y \sqsubseteq y$
shows *cont* $(\lambda x.$ *eventual-iterate* $(d \mathbin{oo} f\ x))$
(**is** *cont ?e*)
⟨*proof*⟩

23.2 Type constructor for finite deflations

default-sort *profinite*

typedef (**open**) *'a fin-defl* = $\{d::'a \to 'a.$ *finite-deflation* $d\}$
⟨*proof*⟩

instantiation *fin-defl* :: (*profinite*) *below*
begin

definition *below-fin-defl-def*:
$op \sqsubseteq \equiv \lambda x\ y.$ *Rep-fin-defl* $x \sqsubseteq$ *Rep-fin-defl* y

instance ⟨*proof*⟩
end

instance *fin-defl* :: (*profinite*) *po*
⟨*proof*⟩

lemma *finite-deflation-Rep-fin-defl*: *finite-deflation* (*Rep-fin-defl* d)
⟨*proof*⟩

lemma *deflation-Rep-fin-defl*: *deflation* (*Rep-fin-defl* d)
⟨*proof*⟩

interpretation *Rep-fin-defl*: *finite-deflation Rep-fin-defl d*
⟨*proof*⟩

lemma *fin-defl-belowI*:
$(\bigwedge x.$ *Rep-fin-defl* $a{\cdot}x = x \Longrightarrow$ *Rep-fin-defl* $b{\cdot}x = x) \Longrightarrow a \sqsubseteq b$
⟨*proof*⟩

lemma *fin-defl-belowD*:
$[\![a \sqsubseteq b;$ *Rep-fin-defl* $a{\cdot}x = x]\!] \Longrightarrow$ *Rep-fin-defl* $b{\cdot}x = x$
⟨*proof*⟩

lemma *fin-defl-eqI*:
$(\bigwedge x.$ *Rep-fin-defl* $a{\cdot}x = x \longleftrightarrow$ *Rep-fin-defl* $b{\cdot}x = x) \Longrightarrow a = b$
⟨*proof*⟩

lemma *Abs-fin-defl-mono*:
$[\![$*finite-deflation* $a;$ *finite-deflation* $b;$ $a \sqsubseteq b]\!]$
$\Longrightarrow$ *Abs-fin-defl* $a \sqsubseteq$ *Abs-fin-defl* b
⟨*proof*⟩

23.3 Take function for finite deflations

definition
defl-approx :: *nat* ⇒ $('a \to 'a) \Rightarrow ('a \to 'a)$
where

defl-approx i d = eventual-iterate (approx i oo d)

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

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

lemma *defl-approx-fixed-iff*:
deflation d ⟹ defl-approx i d·x = x ⟷ approx i·x = x ∧ d·x = x
⟨*proof*⟩

lemma *defl-approx-below*:
⟦a ⊑ b; deflation a; deflation b⟧ ⟹ defl-approx i a ⊑ defl-approx i b
⟨*proof*⟩

lemma *cont2cont-defl-approx*:
assumes *cont*: *cont f* **and** *below*: *⋀x y. f x·y ⊑ y*
shows *cont (λx. defl-approx i (f x))*
⟨*proof*⟩

definition
fd-take :: nat ⇒ 'a fin-defl ⇒ 'a fin-defl
where
fd-take i d = Abs-fin-defl (defl-approx i (Rep-fin-defl d))

lemma *Rep-fin-defl-fd-take*:
Rep-fin-defl (fd-take i d) = defl-approx i (Rep-fin-defl d)
⟨*proof*⟩

lemma *fd-take-fixed-iff*:
Rep-fin-defl (fd-take i d)·x = x ⟷
approx i·x = x ∧ Rep-fin-defl d·x = x
⟨*proof*⟩

lemma *fd-take-below*: *fd-take n d ⊑ d*
⟨*proof*⟩

lemma *fd-take-idem*: *fd-take n (fd-take n d) = fd-take n d*
⟨*proof*⟩

lemma *fd-take-mono*: *a ⊑ b ⟹ fd-take n a ⊑ fd-take n b*
⟨*proof*⟩

lemma *approx-fixed-le-lemma*: *⟦i ≤ j; approx i·x = x⟧ ⟹ approx j·x = x*
⟨*proof*⟩

lemma *fd-take-chain*: $m \leq n \Longrightarrow$ *fd-take* m $a \sqsubseteq$ *fd-take* n a
⟨*proof*⟩

lemma *finite-range-fd-take*: *finite* (*range* (*fd-take* n))
⟨*proof*⟩

lemma *fd-take-covers*: $\exists\, n.$ *fd-take* n $a = a$
⟨*proof*⟩

interpretation *fin-defl*: *basis-take below fd-take*
⟨*proof*⟩

23.4 Defining algebraic deflations by ideal completion

typedef (**open**) $'a$ *alg-defl* =
{S::$'a$ *fin-defl set*. *below.ideal* S}
⟨*proof*⟩

instantiation *alg-defl* :: (*profinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-alg-defl* $x \subseteq$ *Rep-alg-defl* y

instance ⟨*proof*⟩
end

instance *alg-defl* :: (*profinite*) *po*
⟨*proof*⟩

instance *alg-defl* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-alg-defl-lub*:
chain $Y \Longrightarrow$ *Rep-alg-defl* $(\bigsqcup i.\ Y\ i) = (\bigcup i.$ *Rep-alg-defl* $(Y\ i))$
⟨*proof*⟩

lemma *ideal-Rep-alg-defl*: *below.ideal* (*Rep-alg-defl xs*)
⟨*proof*⟩

definition
alg-defl-principal :: $'a$ *fin-defl* ⇒ $'a$ *alg-defl* **where**
alg-defl-principal $t =$ *Abs-alg-defl* $\{u.\ u \sqsubseteq t\}$

lemma *Rep-alg-defl-principal*:
Rep-alg-defl (*alg-defl-principal* t) = $\{u.\ u \sqsubseteq t\}$
⟨*proof*⟩

interpretation *alg-defl*:

ideal-completion below fd-take alg-defl-principal Rep-alg-defl
⟨*proof*⟩

Algebraic deflations are pointed

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

lemma *alg-defl-minimal*:
alg-defl-principal (*Abs-fin-defl* ⊥) ⊑ *x*
⟨*proof*⟩

instance *alg-defl* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-alg-defl-pcpo*: ⊥ = *alg-defl-principal* (*Abs-fin-defl* ⊥)
⟨*proof*⟩

Algebraic deflations are profinite

instantiation *alg-defl* :: (*profinite*) *profinite*
begin

definition
approx-alg-defl-def: *approx* = *alg-defl.completion-approx*

instance
⟨*proof*⟩

end

instance *alg-defl* :: (*bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-alg-defl-principal* [*simp*]:
approx n·(*alg-defl-principal t*) = *alg-defl-principal* (*fd-take n t*)
⟨*proof*⟩

lemma *approx-eq-alg-defl-principal*:
∃ *t*∈*Rep-alg-defl xs*. *approx n*·*xs* = *alg-defl-principal* (*fd-take n t*)
⟨*proof*⟩

23.5 Applying algebraic deflations

definition
cast :: *'a alg-defl* → *'a* → *'a*
where
cast = *alg-defl.basis-fun Rep-fin-defl*

lemma *cast-alg-defl-principal*:
cast·(*alg-defl-principal a*) = *Rep-fin-defl a*
⟨*proof*⟩

lemma *deflation-cast*: *deflation* (*cast*·*d*)
⟨*proof*⟩

lemma *finite-deflation-cast*:
compact d ⟹ *finite-deflation* (*cast*·*d*)
⟨*proof*⟩

interpretation *cast*: *deflation cast*·*d*
⟨*proof*⟩

declare *cast.idem* [*simp*]

lemma *cast-approx*: *cast*·(*approx n*·*A*) = *defl-approx n* (*cast*·*A*)
⟨*proof*⟩

lemma *cast-approx-fixed-iff*:
cast·(*approx i*·*A*)·*x* = *x* ⟷ *approx i*·*x* = *x* ∧ *cast*·*A*·*x* = *x*
⟨*proof*⟩

lemma *defl-approx-cast*: *defl-approx i* (*cast*·*A*) = *cast*·(*approx i*·*A*)
⟨*proof*⟩

lemma *cast-below-imp-below*: *cast*·*A* ⊑ *cast*·*B* ⟹ *A* ⊑ *B*
⟨*proof*⟩

lemma *cast-eq-imp-eq*: *cast*·*A* = *cast*·*B* ⟹ *A* = *B*
⟨*proof*⟩

lemma *cast-strict1* [*simp*]: *cast*·⊥ = ⊥
⟨*proof*⟩

lemma *cast-strict2* [*simp*]: *cast*·*A*·⊥ = ⊥
⟨*proof*⟩

23.6 Deflation membership relation

definition
in-deflation :: *'a*::*profinite* ⇒ *'a alg-defl* ⇒ *bool* (**infixl** ::: *50*)
where
x ::: *A* ⟷ *cast*·*A*·*x* = *x*

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

lemma *in-deflationI*: *cast*·*A*·*x* = *x* ⟹ *x* ::: *A*
⟨*proof*⟩

lemma *cast-fixed*: *x* ::: *A* ⟹ *cast*·*A*·*x* = *x*

⟨*proof*⟩

lemma *cast-in-deflation* [*simp*]: $cast{\cdot}A{\cdot}x ::: A$
⟨*proof*⟩

lemma *bottom-in-deflation* [*simp*]: $\bot ::: A$
⟨*proof*⟩

lemma *subdeflationD*: $A \sqsubseteq B \Longrightarrow x ::: A \Longrightarrow x ::: B$
⟨*proof*⟩

lemma *rev-subdeflationD*: $x ::: A \Longrightarrow A \sqsubseteq B \Longrightarrow x ::: B$
⟨*proof*⟩

lemma *subdeflationI*: $(\bigwedge x.\ x ::: A \Longrightarrow x ::: B) \Longrightarrow A \sqsubseteq B$
⟨*proof*⟩

Identity deflation:

lemma $cast{\cdot}(\bigsqcup i.\ alg\text{-}defl\text{-}principal\ (Abs\text{-}fin\text{-}defl\ (approx\ i))){\cdot}x = x$
⟨*proof*⟩

23.7 Bifinite domains and algebraic deflations

This lemma says that if we have an ep-pair from a bifinite domain into a universal domain, then e oo p is an algebraic deflation.

lemma
assumes *ep-pair e p*
constrains $e :: 'a{::}profinite \to 'b{::}profinite$
shows $\exists d.\ cast{\cdot}d = e\ oo\ p$
⟨*proof*⟩

This lemma says that if we have an ep-pair from a cpo into a bifinite domain, and e oo p is an algebraic deflation, then the cpo is bifinite.

lemma
assumes *ep-pair e p*
constrains $e :: 'a{::}cpo \to 'b{::}profinite$
assumes *d*: $\bigwedge x.\ cast{\cdot}d{\cdot}x = e{\cdot}(p{\cdot}x)$
obtains $a :: nat \Rightarrow 'a \to 'a$ **where**
$\bigwedge i.\ finite\text{-}deflation\ (a\ i)$
$(\bigsqcup i.\ a\ i) = ID$
⟨*proof*⟩

end

24 CompactBasis: Compact bases of domains

theory *CompactBasis*

imports *Completion*
begin

24.1 Compact bases of bifinite domains

default-sort *profinite*

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

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

instantiation *compact-basis* :: (*profinite*) *below*
begin

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

instance ⟨*proof*⟩

end

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

Take function for compact basis

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

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

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

interpretation *compact-basis*:
basis-take below compact-take
⟨*proof*⟩

Ideal completion

definition
approximants :: *'a* ⇒ *'a compact-basis set* **where**
approximants = (λ*x*. {*a*. *Rep-compact-basis a* $\sqsubseteq$ *x*})

interpretation *compact-basis*:
ideal-completion below compact-take Rep-compact-basis approximants

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

24.2 A compact basis for powerdomains

typedef *'a pd-basis* =
{*S*::*'a*::*profinite compact-basis set*. *finite S* ∧ *S* ≠ {}}
⟨*proof*⟩

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

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

unit and plus

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

definition
PDPlus :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* ∪ *Rep-pd-basis u*)

lemma *Rep-PDUnit*:
Rep-pd-basis (*PDUnit x*) = {*x*}
⟨*proof*⟩

lemma *Rep-PDPlus*:
Rep-pd-basis (*PDPlus u v*) = *Rep-pd-basis u* ∪ *Rep-pd-basis v*
⟨*proof*⟩

lemma *PDUnit-inject* [*simp*]: (*PDUnit a* = *PDUnit b*) = (*a* = *b*)
⟨*proof*⟩

lemma *PDPlus-assoc*: *PDPlus* (*PDPlus t u*) *v* = *PDPlus t* (*PDPlus u v*)
⟨*proof*⟩

lemma *PDPlus-commute*: *PDPlus t u* = *PDPlus u t*

⟨*proof*⟩

lemma *PDPlus-absorb*: *PDPlus t t = t*
⟨*proof*⟩

lemma *pd-basis-induct1*:
assumes *PDUnit*: ⋀*a*. *P* (*PDUnit a*)
assumes *PDPlus*: ⋀*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*:
assumes *class.ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDUnit x*) = *g x*
⟨*proof*⟩

lemma *fold-pd-PDPlus*:
assumes *class.ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDPlus t u*) = *f* (*fold-pd g f t*) (*fold-pd g f u*)
⟨*proof*⟩

Take function for powerdomain basis

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

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

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

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

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

lemma *pd-take-covers*: ∃ *n*. *pd-take n t* = *t*
⟨*proof*⟩

end

25 Universal: A universal bifinite domain

theory *Universal*
imports *CompactBasis Nat-Bijection*
begin

25.1 Basis datatype

types *ubasis* = *nat*

definition
node :: *nat* ⇒ *ubasis* ⇒ *ubasis set* ⇒ *ubasis*
where
node i a S = *Suc* (*prod-encode* (*i*, *prod-encode* (*a*, *set-encode S*)))

lemma *node-not-0* [*simp*]: *node i a S* ≠ *0*
⟨*proof*⟩

lemma *node-gt-0* [*simp*]: *0* < *node i a S*
⟨*proof*⟩

lemma *node-inject* [*simp*]:
⟦*finite S*; *finite T*⟧
⟹ *node i a S* = *node j b T* ⟷ *i* = *j* ∧ *a* = *b* ∧ *S* = *T*
⟨*proof*⟩

lemma *node-gt0*: *i* < *node i a S*
⟨*proof*⟩

lemma *node-gt1*: *a* < *node i a S*
⟨*proof*⟩

lemma *nat-less-power2*: *n* < *2^n*
⟨*proof*⟩

lemma *node-gt2*: ⟦*finite S*; *b* ∈ *S*⟧ ⟹ *b* < *node i a S*
⟨*proof*⟩

lemma *eq-prod-encode-pairI*:

⟦*fst* (*prod-decode x*) = *a*; *snd* (*prod-decode x*) = *b*⟧ ⟹ *x* = *prod-encode* (*a*, *b*)
⟨*proof*⟩

lemma *node-cases*:
assumes *1*: *x* = *0* ⟹ *P*
assumes *2*: ⋀*i a S*. ⟦*finite S*; *x* = *node i a S*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *node-induct*:
assumes *1*: *P 0*
assumes *2*: ⋀*i a S*. ⟦*P a*; *finite S*; ∀ *b*∈*S*. *P b*⟧ ⟹ *P* (*node i a S*)
shows *P x*
⟨*proof*⟩

25.2 Basis ordering

inductive
ubasis-le :: *nat* ⇒ *nat* ⇒ *bool*
where
ubasis-le-refl: *ubasis-le a a*
| *ubasis-le-trans*:
⟦*ubasis-le a b*; *ubasis-le b c*⟧ ⟹ *ubasis-le a c*
| *ubasis-le-lower*:
finite S ⟹ *ubasis-le a* (*node i a S*)
| *ubasis-le-upper*:
⟦*finite S*; *b* ∈ *S*; *ubasis-le a b*⟧ ⟹ *ubasis-le* (*node i a S*) *b*

lemma *ubasis-le-minimal*: *ubasis-le 0 x*
⟨*proof*⟩

25.2.1 Generic take function

function
ubasis-until :: (*ubasis* ⇒ *bool*) ⇒ *ubasis* ⇒ *ubasis*
where
ubasis-until P 0 = *0*
| *finite S* ⟹ *ubasis-until P* (*node i a S*) =
(*if P* (*node i a S*) *then node i a S else ubasis-until P a*)
⟨*proof*⟩

termination *ubasis-until*
⟨*proof*⟩

lemma *ubasis-until*: *P 0* ⟹ *P* (*ubasis-until P x*)
⟨*proof*⟩

lemma *ubasis-until'*: *0* < *ubasis-until P x* ⟹ *P* (*ubasis-until P x*)
⟨*proof*⟩

lemma *ubasis-until-same*: $P\ x \Longrightarrow \mathit{ubasis\text{-}until}\ P\ x = x$
⟨*proof*⟩

lemma *ubasis-until-idem*:
$P\ 0 \Longrightarrow \mathit{ubasis\text{-}until}\ P\ (\mathit{ubasis\text{-}until}\ P\ x) = \mathit{ubasis\text{-}until}\ P\ x$
⟨*proof*⟩

lemma *ubasis-until-0*:
$\forall x.\ x \neq 0 \longrightarrow \neg\ P\ x \Longrightarrow \mathit{ubasis\text{-}until}\ P\ x = 0$
⟨*proof*⟩

lemma *ubasis-until-less*: $\mathit{ubasis\text{-}le}\ (\mathit{ubasis\text{-}until}\ P\ x)\ x$
⟨*proof*⟩

lemma *ubasis-until-chain*:
assumes *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\mathit{ubasis\text{-}le}\ (\mathit{ubasis\text{-}until}\ P\ x)\ (\mathit{ubasis\text{-}until}\ Q\ x)$
⟨*proof*⟩

lemma *ubasis-until-mono*:
assumes $\bigwedge i\ a\ S\ b.\ [\![\mathit{finite}\ S;\ P\ (\mathit{node}\ i\ a\ S);\ b \in S;\ \mathit{ubasis\text{-}le}\ a\ b]\!] \Longrightarrow P\ b$
shows $\mathit{ubasis\text{-}le}\ a\ b \Longrightarrow \mathit{ubasis\text{-}le}\ (\mathit{ubasis\text{-}until}\ P\ a)\ (\mathit{ubasis\text{-}until}\ P\ b)$
⟨*proof*⟩

lemma *finite-range-ubasis-until*:
$\mathit{finite}\ \{x.\ P\ x\} \Longrightarrow \mathit{finite}\ (\mathit{range}\ (\mathit{ubasis\text{-}until}\ P))$
⟨*proof*⟩

25.2.2 Take function for *ubasis*

definition
$\mathit{ubasis\text{-}take} :: \mathit{nat} \Rightarrow \mathit{ubasis} \Rightarrow \mathit{ubasis}$
where
$\mathit{ubasis\text{-}take}\ n = \mathit{ubasis\text{-}until}\ (\lambda x.\ x \leq n)$

lemma *ubasis-take-le*: $\mathit{ubasis\text{-}take}\ n\ x \leq n$
⟨*proof*⟩

lemma *ubasis-take-same*: $x \leq n \Longrightarrow \mathit{ubasis\text{-}take}\ n\ x = x$
⟨*proof*⟩

lemma *ubasis-take-idem*: $\mathit{ubasis\text{-}take}\ n\ (\mathit{ubasis\text{-}take}\ n\ x) = \mathit{ubasis\text{-}take}\ n\ x$
⟨*proof*⟩

lemma *ubasis-take-0* [*simp*]: $\mathit{ubasis\text{-}take}\ 0\ x = 0$
⟨*proof*⟩

lemma *ubasis-take-less*: $\mathit{ubasis\text{-}le}\ (\mathit{ubasis\text{-}take}\ n\ x)\ x$
⟨*proof*⟩

lemma *ubasis-take-chain*: *ubasis-le* (*ubasis-take n x*) (*ubasis-take* (*Suc n*) *x*)
⟨*proof*⟩

lemma *ubasis-take-mono*:
assumes *ubasis-le x y*
shows *ubasis-le* (*ubasis-take n x*) (*ubasis-take n y*)
⟨*proof*⟩

lemma *finite-range-ubasis-take*: *finite* (*range* (*ubasis-take n*))
⟨*proof*⟩

lemma *ubasis-take-covers*: ∃ *n*. *ubasis-take n x* = *x*
⟨*proof*⟩

interpretation *udom*: *preorder ubasis-le*
⟨*proof*⟩

interpretation *udom*: *basis-take ubasis-le ubasis-take*
⟨*proof*⟩

25.3 Defining the universal domain by ideal completion

typedef (**open**) *udom* = {*S*. *udom.ideal S*}
⟨*proof*⟩

instantiation *udom* :: *below*
begin

definition
x ⊑ *y* ⟷ *Rep-udom x* ⊆ *Rep-udom y*

instance ⟨*proof*⟩
end

instance *udom* :: *po*
⟨*proof*⟩

instance *udom* :: *cpo*
⟨*proof*⟩

lemma *Rep-udom-lub*:
chain Y ⟹ *Rep-udom* (⨆ *i*. *Y i*) = (⋃ *i*. *Rep-udom* (*Y i*))
⟨*proof*⟩

lemma *ideal-Rep-udom*: *udom.ideal* (*Rep-udom xs*)
⟨*proof*⟩

definition

udom-principal :: *nat* ⇒ *udom* **where**
udom-principal t = *Abs-udom* {*u*. *ubasis-le u t*}

lemma *Rep-udom-principal*:
Rep-udom (*udom-principal t*) = {*u*. *ubasis-le u t*}
⟨*proof*⟩

interpretation *udom*:
ideal-completion ubasis-le ubasis-take udom-principal Rep-udom
⟨*proof*⟩

Universal domain is pointed

lemma *udom-minimal*: *udom-principal 0* ⊑ *x*
⟨*proof*⟩

instance *udom* :: *pcpo*
⟨*proof*⟩

lemma *inst-udom-pcpo*: ⊥ = *udom-principal 0*
⟨*proof*⟩

Universal domain is bifinite

instantiation *udom* :: *bifinite*
begin

definition
approx-udom-def: *approx* = *udom.completion-approx*

instance
⟨*proof*⟩

end

lemma *approx-udom-principal* [*simp*]:
approx n·(*udom-principal x*) = *udom-principal* (*ubasis-take n x*)
⟨*proof*⟩

lemma *approx-eq-udom-principal*:
∃ *a*∈*Rep-udom x*. *approx n*·*x* = *udom-principal* (*ubasis-take n a*)
⟨*proof*⟩

25.4 Universality of *udom*

default-sort *bifinite*

25.4.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
fixes *A* :: *'a*::*po set*

shows $[\![\mathit{finite}\ A;\ A \neq \{\}]\!] \Longrightarrow \exists\, x{\in}A.\ \forall\, y{\in}A.\ x \sqsubseteq y \longrightarrow x = y$
⟨*proof*⟩

definition
choose :: *'a compact-basis set* ⇒ *'a compact-basis*
where
choose A = (*SOME x*. *x* ∈ {*x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*})

lemma *choose-lemma*:
$[\![\mathit{finite}\ A;\ A \neq \{\}]\!] \Longrightarrow \mathit{choose}\ A \in \{x{\in}A.\ \forall\, y{\in}A.\ x \sqsubseteq y \longrightarrow x = y\}$
⟨*proof*⟩

lemma *maximal-choose*:
$[\![\mathit{finite}\ A;\ y \in A;\ \mathit{choose}\ A \sqsubseteq y]\!] \Longrightarrow \mathit{choose}\ A = y$
⟨*proof*⟩

lemma *choose-in*: $[\![\mathit{finite}\ A;\ A \neq \{\}]\!] \Longrightarrow \mathit{choose}\ A \in A$
⟨*proof*⟩

function
choose-pos :: *'a compact-basis set* ⇒ *'a compact-basis* ⇒ *nat*
where
choose-pos A x =
(*if finite A* ∧ *x* ∈ *A* ∧ *x* ≠ *choose A*
then Suc (*choose-pos* (*A* − {*choose A*}) *x*) *else 0*)
⟨*proof*⟩

termination *choose-pos*
⟨*proof*⟩

declare *choose-pos.simps* [*simp del*]

lemma *choose-pos-choose*: *finite A* ⟹ *choose-pos A* (*choose A*) = *0*
⟨*proof*⟩

lemma *inj-on-choose-pos* [*OF refl*]:
$[\![\mathit{card}\ A = n;\ \mathit{finite}\ A]\!] \Longrightarrow \mathit{inj\text{-}on}\ (\mathit{choose\text{-}pos}\ A)\ A$
⟨*proof*⟩

lemma *choose-pos-bounded* [*OF refl*]:
$[\![\mathit{card}\ A = n;\ \mathit{finite}\ A;\ x \in A]\!] \Longrightarrow \mathit{choose\text{-}pos}\ A\ x < n$
⟨*proof*⟩

lemma *choose-pos-lessD*:
$[\![\mathit{choose\text{-}pos}\ A\ x < \mathit{choose\text{-}pos}\ A\ y;\ \mathit{finite}\ A;\ x \in A;\ y \in A]\!] \Longrightarrow \neg\ x \sqsubseteq y$
⟨*proof*⟩

25.4.2 Rank of basis elements

primrec
cb-take :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis*
where
cb-take 0 = (λ*x*. *compact-bot*)
| *cb-take* (*Suc n*) = *compact-take n*

lemma *cb-take-covers*: ∃ *n*. *cb-take n x* = *x*
⟨*proof*⟩

lemma *cb-take-less*: *cb-take n x* ⊑ *x*
⟨*proof*⟩

lemma *cb-take-idem*: *cb-take n* (*cb-take n x*) = *cb-take n x*
⟨*proof*⟩

lemma *cb-take-mono*: *x* ⊑ *y* ⟹ *cb-take n x* ⊑ *cb-take n y*
⟨*proof*⟩

lemma *cb-take-chain-le*: *m* ≤ *n* ⟹ *cb-take m x* ⊑ *cb-take n x*
⟨*proof*⟩

lemma *range-const*: *range* (λ*x*. *c*) = {*c*}
⟨*proof*⟩

lemma *finite-range-cb-take*: *finite* (*range* (*cb-take n*))
⟨*proof*⟩

definition
rank :: *'a compact-basis* ⇒ *nat*
where
rank x = (*LEAST n*. *cb-take n x* = *x*)

lemma *compact-approx-rank*: *cb-take* (*rank x*) *x* = *x*
⟨*proof*⟩

lemma *rank-leD*: *rank x* ≤ *n* ⟹ *cb-take n x* = *x*
⟨*proof*⟩

lemma *rank-leI*: *cb-take n x* = *x* ⟹ *rank x* ≤ *n*
⟨*proof*⟩

lemma *rank-le-iff*: *rank x* ≤ *n* ⟷ *cb-take n x* = *x*
⟨*proof*⟩

lemma *rank-compact-bot* [*simp*]: *rank compact-bot* = *0*
⟨*proof*⟩

lemma *rank-eq-0-iff* [*simp*]: *rank x* = *0* ⟷ *x* = *compact-bot*

⟨*proof*⟩

definition
rank-le :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-le x = {*y*. *rank y* ≤ *rank x*}

definition
rank-lt :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-lt x = {*y*. *rank y* < *rank x*}

definition
rank-eq :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-eq x = {*y*. *rank y* = *rank x*}

lemma *rank-eq-cong*: *rank x* = *rank y* ⟹ *rank-eq x* = *rank-eq y*
⟨*proof*⟩

lemma *rank-lt-cong*: *rank x* = *rank y* ⟹ *rank-lt x* = *rank-lt y*
⟨*proof*⟩

lemma *rank-eq-subset*: *rank-eq x* ⊆ *rank-le x*
⟨*proof*⟩

lemma *rank-lt-subset*: *rank-lt x* ⊆ *rank-le x*
⟨*proof*⟩

lemma *finite-rank-le*: *finite* (*rank-le x*)
⟨*proof*⟩

lemma *finite-rank-eq*: *finite* (*rank-eq x*)
⟨*proof*⟩

lemma *finite-rank-lt*: *finite* (*rank-lt x*)
⟨*proof*⟩

lemma *rank-lt-Int-rank-eq*: *rank-lt x* ∩ *rank-eq x* = {}
⟨*proof*⟩

lemma *rank-lt-Un-rank-eq*: *rank-lt x* ∪ *rank-eq x* = *rank-le x*
⟨*proof*⟩

25.4.3 Sequencing basis elements

definition
place :: *'a compact-basis* ⇒ *nat*
where

place x = card (rank-lt x) + choose-pos (rank-eq x) x

lemma *place-bounded*: *place x < card (rank-le x)*
⟨*proof*⟩

lemma *place-ge*: *card (rank-lt x) ≤ place x*
⟨*proof*⟩

lemma *place-rank-mono*:
fixes *x y* :: *'a compact-basis*
shows *rank x < rank y ⟹ place x < place y*
⟨*proof*⟩

lemma *place-eqD*: *place x = place y ⟹ x = y*
⟨*proof*⟩

lemma *inj-place*: *inj place*
⟨*proof*⟩

25.4.4 Embedding and projection on basis elements

definition
sub :: *'a compact-basis ⇒ 'a compact-basis*
where
sub x = (case rank x of 0 ⇒ compact-bot | Suc k ⇒ cb-take k x)

lemma *rank-sub-less*: *x ≠ compact-bot ⟹ rank (sub x) < rank x*
⟨*proof*⟩

lemma *place-sub-less*: *x ≠ compact-bot ⟹ place (sub x) < place x*
⟨*proof*⟩

lemma *sub-below*: *sub x ⊑ x*
⟨*proof*⟩

lemma *rank-less-imp-below-sub*: *⟦x ⊑ y; rank x < rank y⟧ ⟹ x ⊑ sub y*
⟨*proof*⟩

function
basis-emb :: *'a compact-basis ⇒ ubasis*
where
basis-emb x = (if x = compact-bot then 0 else
node (place x) (basis-emb (sub x))
(basis-emb ‘ {y. place y < place x ∧ x ⊑ y}))
⟨*proof*⟩

termination *basis-emb*
⟨*proof*⟩

declare *basis-emb.simps* [*simp del*]

lemma *basis-emb-compact-bot* [*simp*]: *basis-emb compact-bot = 0*
⟨*proof*⟩

lemma *fin1*: *finite* {*y*. *place y < place x* ∧ *x* ⊑ *y*}
⟨*proof*⟩

lemma *fin2*: *finite* (*basis-emb* ‘ {*y*. *place y < place x* ∧ *x* ⊑ *y*})
⟨*proof*⟩

lemma *rank-place-mono*:
⟦*place x < place y*; *x* ⊑ *y*⟧ ⟹ *rank x < rank y*
⟨*proof*⟩

lemma *basis-emb-mono*:
x ⊑ *y* ⟹ *ubasis-le* (*basis-emb x*) (*basis-emb y*)
⟨*proof*⟩

lemma *inj-basis-emb*: *inj basis-emb*
⟨*proof*⟩

definition
basis-prj :: *ubasis* ⇒ *'a compact-basis*
where
basis-prj x = *inv basis-emb*
(*ubasis-until* (λ*x*. *x* ∈ *range* (*basis-emb* :: *'a compact-basis* ⇒ *ubasis*)) *x*)

lemma *basis-prj-basis-emb*: ⋀*x*. *basis-prj* (*basis-emb x*) = *x*
⟨*proof*⟩

lemma *basis-prj-node*:
⟦*finite S*; *node i a S* ∉ *range* (*basis-emb* :: *'a compact-basis* ⇒ *nat*)⟧
⟹ *basis-prj* (*node i a S*) = (*basis-prj a* :: *'a compact-basis*)
⟨*proof*⟩

lemma *basis-prj-0*: *basis-prj 0 = compact-bot*
⟨*proof*⟩

lemma *node-eq-basis-emb-iff*:
finite S ⟹ *node i a S = basis-emb x* ⟷
x ≠ *compact-bot* ∧ *i = place x* ∧ *a = basis-emb* (*sub x*) ∧
S = basis-emb ‘ {*y*. *place y < place x* ∧ *x* ⊑ *y*}
⟨*proof*⟩

lemma *basis-prj-mono*: *ubasis-le a b* ⟹ *basis-prj a* ⊑ *basis-prj b*
⟨*proof*⟩

lemma *basis-emb-prj-less*: *ubasis-le* (*basis-emb* (*basis-prj x*)) *x*

⟨*proof*⟩

hide-const (**open**)
node
choose
choose-pos
place
sub

25.4.5 EP-pair from any bifinite domain into *udom*

definition
udom-emb :: *'a*::*bifinite* → *udom*
where
udom-emb = *compact-basis.basis-fun* (λ*x*. *udom-principal* (*basis-emb x*))

definition
udom-prj :: *udom* → *'a*::*bifinite*
where
udom-prj = *udom.basis-fun* (λ*x*. *Rep-compact-basis* (*basis-prj x*))

lemma *udom-emb-principal*:
udom-emb·(*Rep-compact-basis x*) = *udom-principal* (*basis-emb x*)
⟨*proof*⟩

lemma *udom-prj-principal*:
udom-prj·(*udom-principal x*) = *Rep-compact-basis* (*basis-prj x*)
⟨*proof*⟩

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
⟨*proof*⟩

end

26 Domain-Aux: Domain package support

theory *Domain-Aux*
imports *Ssum Sprod Fixrec*
uses
(*Tools/Domain/domain-take-proofs.ML*)
begin

26.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
fixes *abs* :: *'a* → *'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-below*: $(abs{\cdot}x \sqsubseteq abs{\cdot}y) = (x \sqsubseteq y)$
⟨*proof*⟩

lemma *rep-below*: $(rep{\cdot}x \sqsubseteq rep{\cdot}y) = (x \sqsubseteq y)$
⟨*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 *casedist-rule*: $rep{\cdot}x = \bot \vee P \Longrightarrow x = \bot \vee P$
⟨*proof*⟩

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

lemma *compact-rep-rev*: *compact* (*rep*·*x*) ⟹ *compact x*
⟨*proof*⟩

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

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

lemma *iso-swap*: (*x* = *abs*·*y*) = (*rep*·*x* = *y*)
⟨*proof*⟩

end

26.2 Proofs about take functions

This section contains lemmas that are used in a module that supports the domain isomorphism package; the module contains proofs related to take functions and the finiteness predicate.

lemma *deflation-abs-rep*:
fixes *abs* **and** *rep* **and** *d*
assumes *abs-iso*: ⋀*x*. *rep*·(*abs*·*x*) = *x*
assumes *rep-iso*: ⋀*y*. *abs*·(*rep*·*y*) = *y*
shows *deflation d* ⟹ *deflation* (*abs oo d oo rep*)
⟨*proof*⟩

lemma *deflation-chain-min*:
assumes *chain*: *chain d*
assumes *defl*: ⋀*n*. *deflation* (*d n*)
shows *d m*·(*d n*·*x*) = *d* (*min m n*)·*x*
⟨*proof*⟩

lemma *lub-ID-take-lemma*:
assumes *chain t* **and** (⨆ *n*. *t n*) = *ID*
assumes ⋀*n*. *t n*·*x* = *t n*·*y* **shows** *x* = *y*
⟨*proof*⟩

lemma *lub-ID-reach*:
assumes *chain t* **and** (⨆ *n*. *t n*) = *ID*
shows (⨆ *n*. *t n*·*x*) = *x*
⟨*proof*⟩

lemma *lub-ID-take-induct*:
assumes *chain t* **and** (⨆ *n*. *t n*) = *ID*
assumes *adm P* **and** ⋀*n*. *P* (*t n*·*x*) **shows** *P x*
⟨*proof*⟩

26.3 Finiteness

Let a "decisive" function be a deflation that maps every input to either itself or bottom. Then if a domain's take functions are all decisive, then all values in the domain are finite.

definition
decisive :: $('a{::}pcpo \to 'a) \Rightarrow bool$
where
$decisive\ d \longleftrightarrow (\forall x.\ d{\cdot}x = x \vee d{\cdot}x = \bot)$

lemma *decisiveI*: $(\bigwedge x.\ d{\cdot}x = x \vee d{\cdot}x = \bot) \Longrightarrow decisive\ d$
⟨*proof*⟩

lemma *decisive-cases*:
assumes *decisive d* **obtains** $d{\cdot}x = x \mid d{\cdot}x = \bot$
⟨*proof*⟩

lemma *decisive-bottom*: *decisive* $\bot$
⟨*proof*⟩

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

lemma *decisive-ssum-map*:
assumes *f*: *decisive f*
assumes *g*: *decisive g*
shows *decisive* (*ssum-map*·*f*·*g*)
⟨*proof*⟩

lemma *decisive-sprod-map*:
assumes *f*: *decisive f*
assumes *g*: *decisive g*
shows *decisive* (*sprod-map*·*f*·*g*)
⟨*proof*⟩

lemma *decisive-abs-rep*:
fixes *abs rep*
assumes *iso*: *iso abs rep*
assumes *d*: *decisive d*
shows *decisive* (*abs oo d oo rep*)
⟨*proof*⟩

lemma *lub-ID-finite*:
assumes *chain*: *chain d*
assumes *lub*: $(\bigsqcup n.\ d\ n) = ID$
assumes *decisive*: $\bigwedge n.\ decisive\ (d\ n)$
shows $\exists n.\ d\ n{\cdot}x = x$
⟨*proof*⟩

lemma *lub-ID-finite-take-induct*:
assumes *chain d* **and** $(\bigsqcup n.\ d\ n) = ID$ **and** $\bigwedge n.\ decisive\ (d\ n)$
shows $(\bigwedge n.\ P\ (d\ n{\cdot}x)) \Longrightarrow P\ x$
⟨*proof*⟩

26.4 ML setup

⟨*ML*⟩

end

27 Representable: Representable Types

theory *Representable*
imports *Algebraic Universal Ssum Sprod One Fixrec Domain-Aux*
uses
(*Tools/repdef.ML*)
(*Tools/Domain/domain-isomorphism.ML*)
begin

27.1 Class of representable types

Overloaded embedding and projection functions between a representable type and the universal domain.

class *rep* = *bifinite* +
fixes *emb* :: $'a$::*pcpo* → *udom*
fixes *prj* :: *udom* → $'a$::*pcpo*
assumes *ep-pair-emb-prj*: *ep-pair emb prj*

interpretation *rep*!:
pcpo-ep-pair
emb :: $'a$::*rep* → *udom*
prj :: *udom* → $'a$::*rep*
⟨*proof*⟩

lemmas *emb-inverse* = *rep.e-inverse*
lemmas *emb-prj-below* = *rep.e-p-below*
lemmas *emb-eq-iff* = *rep.e-eq-iff*
lemmas *emb-strict* = *rep.e-strict*
lemmas *prj-strict* = *rep.p-strict*

27.2 Making *rep* the default class

From now on, free type variables are assumed to be in class *rep*, unless specified otherwise.

default-sort *rep*

27.3 Representations of types

A TypeRep is an algebraic deflation over the universe of values.

types *TypeRep* = *udom alg-defl*
translations (*type*) *TypeRep* ↽ (*type*) *udom alg-defl*

definition
Rep-of :: *'a*::*rep itself* ⇒ *TypeRep*
where
Rep-of TYPE(*'a*::*rep*) =
(⨆ *i*. *alg-defl-principal* (*Abs-fin-defl*
(*emb oo* (*approx i* :: *'a* → *'a*) *oo prj*)))

syntax -*REP* :: *type* ⇒ *TypeRep* ((*1REP*/(*1'*(-'))))
translations *REP*(*'t*) ⇌ *CONST Rep-of TYPE*(*'t*)

lemma *cast-REP*:
cast·*REP*(*'a*::*rep*) = (*emb*::*'a* → *udom*) *oo* (*prj*::*udom* → *'a*)
⟨*proof*⟩

lemma *emb-prj*: *emb*·((*prj*·*x*)::*'a*::*rep*) = *cast*·*REP*(*'a*)·*x*
⟨*proof*⟩

lemma *in-REP-iff*:
x ::: *REP*(*'a*::*rep*) ⟷ *emb*·((*prj*·*x*)::*'a*) = *x*
⟨*proof*⟩

lemma *prj-inverse*:
x ::: *REP*(*'a*::*rep*) ⟹ *emb*·((*prj*·*x*)::*'a*) = *x*
⟨*proof*⟩

lemma *emb-in-REP* [*simp*]:
emb·(*x*::*'a*::*rep*) ::: *REP*(*'a*)
⟨*proof*⟩

27.4 Coerce operator

definition *coerce* :: *'a* → *'b*
where *coerce* = *prj oo emb*

lemma *beta-coerce*: *coerce*·*x* = *prj*·(*emb*·*x*)
⟨*proof*⟩

lemma *prj-emb*: *prj*·(*emb*·*x*) = *coerce*·*x*
⟨*proof*⟩

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

lemma *coerce-eq-ID* [*simp*]: $(coerce :: 'a \to 'a) = ID$
⟨*proof*⟩

lemma *emb-coerce*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow emb{\cdot}((coerce{::}'a \to 'b){\cdot}x) = emb{\cdot}x$
⟨*proof*⟩

lemma *coerce-prj*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow (coerce{::}'b \to 'a){\cdot}(prj{\cdot}x) = prj{\cdot}x$
⟨*proof*⟩

lemma *coerce-coerce* [*simp*]:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow coerce{\cdot}((coerce{::}'a \to 'b){\cdot}x) = coerce{\cdot}x$
⟨*proof*⟩

lemma *coerce-inverse*:
$emb{\cdot}(x{::}'a) ::: REP('b) \Longrightarrow coerce{\cdot}(coerce{\cdot}x :: 'b) = x$
⟨*proof*⟩

lemma *coerce-type*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow emb{\cdot}((coerce{::}'a \to 'b){\cdot}x) ::: REP('a)$
⟨*proof*⟩

lemma *ep-pair-coerce*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow ep\text{-}pair\ (coerce{::}'a \to 'b)\ (coerce{::}'b \to 'a)$
⟨*proof*⟩

Isomorphism lemmas used internally by the domain package:

lemma *domain-abs-iso*:
fixes *abs* **and** *rep*
assumes *REP*: $REP('b) = REP('a)$
assumes *abs-def*: $abs \equiv (coerce :: 'a \to 'b)$
assumes *rep-def*: $rep \equiv (coerce :: 'b \to 'a)$
shows $rep{\cdot}(abs{\cdot}x) = x$
⟨*proof*⟩

lemma *domain-rep-iso*:
fixes *abs* **and** *rep*
assumes *REP*: $REP('b) = REP('a)$
assumes *abs-def*: $abs \equiv (coerce :: 'a \to 'b)$
assumes *rep-def*: $rep \equiv (coerce :: 'b \to 'a)$
shows $abs{\cdot}(rep{\cdot}x) = x$
⟨*proof*⟩

27.5 Proving a subtype is representable

Temporarily relax type constraints for *approx*, *emb*, and *prj*.

⟨*ML*⟩

definition
  repdef-approx ::
    ('*a*::*pcpo* ⇒ *udom*) ⇒ (*udom* ⇒ '*a*) ⇒ *udom alg-defl* ⇒ *nat* ⇒ '*a* → '*a*
where
  repdef-approx Rep Abs t = (λ*i*. Λ *x*. *Abs* (*cast*·(*approx i*·*t*)·(*Rep x*)))

lemma *typedef-rep-class*:
  fixes *Rep* :: '*a*::*pcpo* ⇒ *udom*
  fixes *Abs* :: *udom* ⇒ '*a*::*pcpo*
  fixes *t* :: *TypeRep*
  assumes *type*: *type-definition Rep Abs* {*x*. *x* ::: *t*}
  assumes *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
  assumes *emb*: *emb* ≡ (Λ *x*. *Rep x*)
  assumes *prj*: *prj* ≡ (Λ *x*. *Abs* (*cast*·*t*·*x*))
  assumes *approx*: (*approx* :: *nat* ⇒ '*a* → '*a*) ≡ *repdef-approx Rep Abs t*
  shows *OFCLASS*('*a*, *rep-class*)
⟨*proof*⟩

Restore original typing constraints

⟨*ML*⟩

lemma *typedef-REP*:
  fixes *Rep* :: '*a*::*rep* ⇒ *udom*
  fixes *Abs* :: *udom* ⇒ '*a*::*rep*
  fixes *t* :: *TypeRep*
  assumes *type*: *type-definition Rep Abs* {*x*. *x* ::: *t*}
  assumes *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
  assumes *emb*: *emb* ≡ (Λ *x*. *Rep x*)
  assumes *prj*: *prj* ≡ (Λ *x*. *Abs* (*cast*·*t*·*x*))
  shows *REP*('*a*) = *t*
⟨*proof*⟩

lemma *adm-mem-Collect-in-deflation*: *adm* (λ*x*. *x* ∈ {*x*. *x* ::: *A*})
⟨*proof*⟩

⟨*ML*⟩

27.6 Instances of class *rep*

27.6.1 Universal Domain

The Universal Domain itself is trivially representable.

instantiation *udom* :: *rep*
begin

definition *emb-udom-def* [*simp*]: *emb* = (*ID* :: *udom* → *udom*)
definition *prj-udom-def* [*simp*]: *prj* = (*ID* :: *udom* → *udom*)

instance
⟨*proof*⟩

end

27.6.2 Lifted types

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

definition *emb-lift-def*:
emb = *udom-emb oo* (*FLIFT x*. *Def* (*to-nat x*))

definition *prj-lift-def*:
prj = (*FLIFT n*. *if* (∃ *x*::′*a*::*countable*. *n* = *to-nat x*)
then Def (*THE x*::′*a*. *n* = *to-nat x*) *else* ⊥) *oo udom-prj*

instance
⟨*proof*⟩

end

27.6.3 Representable type constructors

Functions between representable types are representable.

instantiation *cfun* :: (*rep*, *rep*) *rep*
begin

definition *emb-cfun-def*: *emb* = *udom-emb oo cfun-map*·*prj*·*emb*
definition *prj-cfun-def*: *prj* = *cfun-map*·*emb*·*prj oo udom-prj*

instance
⟨*proof*⟩

end

Strict products of representable types are representable.

instantiation *sprod* :: (*rep*, *rep*) *rep*
begin

definition *emb-sprod-def*: *emb* = *udom-emb oo sprod-map*·*emb*·*emb*
definition *prj-sprod-def*: *prj* = *sprod-map*·*prj*·*prj oo udom-prj*

instance
⟨*proof*⟩

end

Strict sums of representable types are representable.

instantiation *ssum* :: (*rep*, *rep*) *rep*
begin

definition *emb-ssum-def*: *emb* = *udom-emb oo ssum-map·emb·emb*
definition *prj-ssum-def*: *prj* = *ssum-map·prj·prj oo udom-prj*

instance
⟨*proof*⟩

end

Up of a representable type is representable.

instantiation *u* :: (*rep*) *rep*
begin

definition *emb-u-def*: *emb* = *udom-emb oo u-map·emb*
definition *prj-u-def*: *prj* = *u-map·prj oo udom-prj*

instance
⟨*proof*⟩

end

Cartesian products of representable types are representable.

instantiation ∗ :: (*rep*, *rep*) *rep*
begin

definition *emb-cprod-def*: *emb* = *udom-emb oo cprod-map·emb·emb*
definition *prj-cprod-def*: *prj* = *cprod-map·prj·prj oo udom-prj*

instance
⟨*proof*⟩

end

27.7 Type combinators

definition
TypeRep-fun1 ::
((*udom* → *udom*) → (*'a* → *'a*))
⇒ (*TypeRep* → *TypeRep*)
where
TypeRep-fun1 f =
alg-defl.basis-fun (λ*a*.
alg-defl-principal (

Abs-fin-defl (*udom-emb oo f*·(*Rep-fin-defl a*) *oo udom-prj*)))

definition
TypeRep-fun2 ::
((*udom* → *udom*) → (*udom* → *udom*) → ('*a* → '*a*))
⇒ (*TypeRep* → *TypeRep* → *TypeRep*)
where
TypeRep-fun2 f =
alg-defl.basis-fun (λ*a*.
alg-defl.basis-fun (λ*b*.
alg-defl-principal (
Abs-fin-defl (*udom-emb oo*
f·(*Rep-fin-defl a*)·(*Rep-fin-defl b*) *oo udom-prj*))))

definition *cfun-defl* = *TypeRep-fun2 cfun-map*
definition *ssum-defl* = *TypeRep-fun2 ssum-map*
definition *sprod-defl* = *TypeRep-fun2 sprod-map*
definition *cprod-defl* = *TypeRep-fun2 cprod-map*
definition *u-defl* = *TypeRep-fun1 u-map*

lemma *Rep-fin-defl-mono*: *a* ⊑ *b* ⟹ *Rep-fin-defl a* ⊑ *Rep-fin-defl b*
⟨*proof*⟩

lemma *cast-TypeRep-fun1*:
assumes *f*: ⋀*a*. *finite-deflation a* ⟹ *finite-deflation* (*f*·*a*)
shows *cast*·(*TypeRep-fun1 f*·*A*) = *udom-emb oo f*·(*cast*·*A*) *oo udom-prj*
⟨*proof*⟩

lemma *cast-TypeRep-fun2*:
assumes *f*: ⋀*a b*. *finite-deflation a* ⟹ *finite-deflation b* ⟹
finite-deflation (*f*·*a*·*b*)
shows *cast*·(*TypeRep-fun2 f*·*A*·*B*) =
udom-emb oo f·(*cast*·*A*)·(*cast*·*B*) *oo udom-prj*
⟨*proof*⟩

lemma *cast-cfun-defl*:
cast·(*cfun-defl*·*A*·*B*) = *udom-emb oo cfun-map*·(*cast*·*A*)·(*cast*·*B*) *oo udom-prj*
⟨*proof*⟩

lemma *cast-ssum-defl*:
cast·(*ssum-defl*·*A*·*B*) = *udom-emb oo ssum-map*·(*cast*·*A*)·(*cast*·*B*) *oo udom-prj*
⟨*proof*⟩

lemma *cast-sprod-defl*:
cast·(*sprod-defl*·*A*·*B*) = *udom-emb oo sprod-map*·(*cast*·*A*)·(*cast*·*B*) *oo udom-prj*
⟨*proof*⟩

lemma *cast-cprod-defl*:
cast·(*cprod-defl*·*A*·*B*) = *udom-emb oo cprod-map*·(*cast*·*A*)·(*cast*·*B*) *oo udom-prj*

⟨*proof*⟩

lemma *cast-u-defl*:
cast·(*u-defl*·*A*) = *udom-emb oo u-map*·(*cast*·*A*) *oo udom-prj*
⟨*proof*⟩

REP of type constructor = type combinator

lemma *REP-cfun*: *REP*('*a* → '*b*) = *cfun-defl*·*REP*('*a*)·*REP*('*b*)
⟨*proof*⟩

lemma *REP-ssum*: *REP*('*a* ⊕ '*b*) = *ssum-defl*·*REP*('*a*)·*REP*('*b*)
⟨*proof*⟩

lemma *REP-sprod*: *REP*('*a* ⊗ '*b*) = *sprod-defl*·*REP*('*a*)·*REP*('*b*)
⟨*proof*⟩

lemma *REP-cprod*: *REP*('*a* × '*b*) = *cprod-defl*·*REP*('*a*)·*REP*('*b*)
⟨*proof*⟩

lemma *REP-up*: *REP*('*a* *u*) = *u-defl*·*REP*('*a*)
⟨*proof*⟩

lemmas *REP-simps* =
REP-cfun
REP-ssum
REP-sprod
REP-cprod
REP-up

27.8 Isomorphic deflations

definition
isodefl :: ('*a*::*rep* → '*a*) ⇒ *udom alg-defl* ⇒ *bool*
where
isodefl d t ⟷ *cast*·*t* = *emb oo d oo prj*

lemma *isodeflI*: (⋀*x*. *cast*·*t*·*x* = *emb*·(*d*·(*prj*·*x*))) ⟹ *isodefl d t*
⟨*proof*⟩

lemma *cast-isodefl*: *isodefl d t* ⟹ *cast*·*t* = (Λ *x*. *emb*·(*d*·(*prj*·*x*)))
⟨*proof*⟩

lemma *isodefl-strict*: *isodefl d t* ⟹ *d*·⊥ = ⊥
⟨*proof*⟩

lemma *isodefl-imp-deflation*:
fixes *d* :: '*a*::*rep* → '*a*
assumes *isodefl d t* **shows** *deflation d*
⟨*proof*⟩

lemma *isodefl-ID-REP*: *isodefl* (*ID* :: *'a* → *'a*) *REP*(*'a*)
⟨*proof*⟩

lemma *isodefl-REP-imp-ID*: *isodefl* (*d* :: *'a* → *'a*) *REP*(*'a*) ⟹ *d* = *ID*
⟨*proof*⟩

lemma *isodefl-bottom*: *isodefl* ⊥ ⊥
⟨*proof*⟩

lemma *adm-isodefl*:
cont f ⟹ *cont g* ⟹ *adm* (λ*x*. *isodefl* (*f x*) (*g x*))
⟨*proof*⟩

lemma *isodefl-lub*:
assumes *chain d* **and** *chain t*
assumes ⋀*i*. *isodefl* (*d i*) (*t i*)
shows *isodefl* (⨆ *i*. *d i*) (⨆ *i*. *t i*)
⟨*proof*⟩

lemma *isodefl-fix*:
assumes ⋀*d t*. *isodefl d t* ⟹ *isodefl* (*f*·*d*) (*g*·*t*)
shows *isodefl* (*fix*·*f*) (*fix*·*g*)
⟨*proof*⟩

lemma *isodefl-coerce*:
fixes *d* :: *'a* → *'a*
assumes *REP*: *REP*(*'b*) = *REP*(*'a*)
shows *isodefl d t* ⟹ *isodefl* (*coerce oo d oo coerce* :: *'b* → *'b*) *t*
⟨*proof*⟩

lemma *isodefl-abs-rep*:
fixes *abs* **and** *rep* **and** *d*
assumes *REP*: *REP*(*'b*) = *REP*(*'a*)
assumes *abs-def*: *abs* ≡ (*coerce* :: *'a* → *'b*)
assumes *rep-def*: *rep* ≡ (*coerce* :: *'b* → *'a*)
shows *isodefl d t* ⟹ *isodefl* (*abs oo d oo rep*) *t*
⟨*proof*⟩

lemma *isodefl-cfun*:
isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
isodefl (*cfun-map*·*d1*·*d2*) (*cfun-defl*·*t1*·*t2*)
⟨*proof*⟩

lemma *isodefl-ssum*:
isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
isodefl (*ssum-map*·*d1*·*d2*) (*ssum-defl*·*t1*·*t2*)
⟨*proof*⟩

lemma *isodefl-sprod*:
isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
isodefl (*sprod-map*·*d1*·*d2*) (*sprod-defl*·*t1*·*t2*)
⟨*proof*⟩

lemma *isodefl-cprod*:
isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
isodefl (*cprod-map*·*d1*·*d2*) (*cprod-defl*·*t1*·*t2*)
⟨*proof*⟩

lemma *isodefl-u*:
isodefl d t $\Longrightarrow$ *isodefl* (*u-map*·*d*) (*u-defl*·*t*)
⟨*proof*⟩

27.9 Constructing Domain Isomorphisms

⟨*ML*⟩

end

28 Domain: Domain package

theory *Domain*
imports *Ssum Sprod Up One Tr Fixrec Representable*
uses
(*Tools/cont-consts.ML*)
(*Tools/cont-proc.ML*)
(*Tools/Domain/domain-constructors.ML*)
(*Tools/Domain/domain-library.ML*)
(*Tools/Domain/domain-axioms.ML*)
(*Tools/Domain/domain-theorems.ML*)
(*Tools/Domain/domain-extender.ML*)
begin

default-sort *pcpo*

28.1 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*

28.2 Combinators for building copy functions

lemmas *domain-map-stricts* =
ssum-map-strict sprod-map-strict u-map-strict

lemmas *domain-map-simps* =
ssum-map-sinl ssum-map-sinr sprod-map-spair u-map-up

28.3 Installing the domain package

lemmas *con-strict-rules* =

sinl-strict sinr-strict spair-strict1 spair-strict2

lemmas *con-defin-rules* =
sinl-defined sinr-defined spair-defined up-defined ONE-defined

lemmas *con-defined-iff-rules* =
sinl-defined-iff sinr-defined-iff spair-strict-iff up-defined ONE-defined

lemmas *con-below-iff-rules* =
sinl-below sinr-below sinl-below-sinr sinr-below-sinl con-defined-iff-rules

lemmas *con-eq-iff-rules* =
sinl-eq sinr-eq sinl-eq-sinr sinr-eq-sinl con-defined-iff-rules

lemmas *sel-strict-rules* =
cfcomp2 sscase1 sfst-strict ssnd-strict fup1

lemma *sel-app-extra-rules*:
$sscase{\cdot}ID{\cdot}\bot{\cdot}(sinr{\cdot}x) = \bot$
$sscase{\cdot}ID{\cdot}\bot{\cdot}(sinl{\cdot}x) = x$
$sscase{\cdot}\bot{\cdot}ID{\cdot}(sinl{\cdot}x) = \bot$
$sscase{\cdot}\bot{\cdot}ID{\cdot}(sinr{\cdot}x) = x$
$fup{\cdot}ID{\cdot}(up{\cdot}x) = x$
⟨*proof*⟩

lemmas *sel-app-rules* =
sel-strict-rules sel-app-extra-rules
ssnd-spair sfst-spair up-defined spair-defined

lemmas *sel-defined-iff-rules* =
cfcomp2 sfst-defined-iff ssnd-defined-iff

lemmas *take-con-rules* =
ssum-map-sinl' ssum-map-sinr' sprod-map-spair' u-map-up
deflation-strict deflation-ID ID1 cfcomp2

⟨*ML*⟩

end

29 UpperPD: Upper powerdomain

theory *UpperPD*
imports *CompactBasis*
begin

29.1 Basis preorder

definition
upper-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♯ *50*) **where**
upper-le = (λ*u v*. ∀ *y*∈*Rep-pd-basis v*. ∃ *x*∈*Rep-pd-basis u*. *x* ⊑ *y*)

lemma *upper-le-refl* [*simp*]: *t* ≤♯ *t*
⟨*proof*⟩

lemma *upper-le-trans*: ⟦*t* ≤♯ *u*; *u* ≤♯ *v*⟧ ⟹ *t* ≤♯ *v*
⟨*proof*⟩

interpretation *upper-le*: *preorder upper-le*
⟨*proof*⟩

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* ≤♯ *t*
⟨*proof*⟩

lemma *PDUnit-upper-mono*: *x* ⊑ *y* ⟹ *PDUnit x* ≤♯ *PDUnit y*
⟨*proof*⟩

lemma *PDPlus-upper-mono*: ⟦*s* ≤♯ *t*; *u* ≤♯ *v*⟧ ⟹ *PDPlus s u* ≤♯ *PDPlus t v*
⟨*proof*⟩

lemma *PDPlus-upper-le*: *PDPlus t u* ≤♯ *t*
⟨*proof*⟩

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* ≤♯ *PDUnit b*) = *a* ⊑ *b*
⟨*proof*⟩

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

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

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

lemma *pd-take-upper-chain*:
pd-take n t ≤♯ *pd-take* (*Suc n*) *t*
⟨*proof*⟩

lemma *pd-take-upper-le*: *pd-take i t* $\leq\sharp$ *t*
⟨*proof*⟩

lemma *pd-take-upper-mono*:
t $\leq\sharp$ *u* $\Longrightarrow$ *pd-take n t* $\leq\sharp$ *pd-take n u*
⟨*proof*⟩

29.2 Type definition

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

instantiation *upper-pd* :: (*profinite*) *below*
begin

definition
x $\sqsubseteq$ *y* $\longleftrightarrow$ *Rep-upper-pd x* $\subseteq$ *Rep-upper-pd y*

instance ⟨*proof*⟩
end

instance *upper-pd* :: (*profinite*) *po*
⟨*proof*⟩

instance *upper-pd* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-upper-pd-lub*:
chain Y $\Longrightarrow$ *Rep-upper-pd* ($\bigsqcup$ *i*. *Y i*) = ($\bigcup$ *i*. *Rep-upper-pd* (*Y i*))
⟨*proof*⟩

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

definition
upper-principal :: *'a pd-basis* $\Rightarrow$ *'a upper-pd* **where**
upper-principal t = *Abs-upper-pd* {*u*. *u* $\leq\sharp$ *t*}

lemma *Rep-upper-principal*:
Rep-upper-pd (*upper-principal t*) = {*u*. *u* $\leq\sharp$ *t*}
⟨*proof*⟩

interpretation *upper-pd*:
ideal-completion upper-le pd-take upper-principal Rep-upper-pd
⟨*proof*⟩

Upper powerdomain is pointed

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*

⟨*proof*⟩

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

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

Upper powerdomain is profinite

instantiation *upper-pd* :: (*profinite*) *profinite*
begin

definition
approx-upper-pd-def: *approx* = *upper-pd.completion-approx*

instance
⟨*proof*⟩

end

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

lemma *approx-upper-principal* [*simp*]:
approx n·(*upper-principal t*) = *upper-principal* (*pd-take n t*)
⟨*proof*⟩

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

29.3 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
  $\{x,xs\}\sharp == \{x\}\sharp +\sharp \{xs\}\sharp$
  $\{x\}\sharp == CONST\ upper\text{-}unit{\cdot}x$

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*⟩

interpretation *upper-add*!: *semilattice upper-add* ⟨*proof*⟩

lemmas *upper-plus-assoc* = *upper-add.assoc*
lemmas *upper-plus-commute* = *upper-add.commute*
lemmas *upper-plus-absorb* = *upper-add.idem*
lemmas *upper-plus-left-commute* = *upper-add.left-commute*
lemmas *upper-plus-left-absorb* = *upper-add.left-idem*

Useful for *simp add*: *upper-plus-ac*

lemmas *upper-plus-ac* =
  upper-plus-assoc upper-plus-commute upper-plus-left-commute

Useful for *simp only*: *upper-plus-aci*

lemmas *upper-plus-aci* =
  upper-plus-ac upper-plus-absorb upper-plus-left-absorb

lemma *upper-plus-below1*: $xs +\sharp ys \sqsubseteq xs$
⟨*proof*⟩

lemma *upper-plus-below2*: $xs +\sharp ys \sqsubseteq ys$
⟨*proof*⟩

lemma *upper-plus-greatest*: $[\![xs \sqsubseteq ys;\ xs \sqsubseteq zs]\!] \Longrightarrow xs \sqsubseteq ys +\sharp zs$
⟨*proof*⟩

lemma *upper-below-plus-iff*:
  $xs \sqsubseteq ys +\sharp zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
⟨*proof*⟩

lemma *upper-plus-below-unit-iff*:
$xs +\sharp ys \sqsubseteq \{z\}\sharp \longleftrightarrow xs \sqsubseteq \{z\}\sharp \vee ys \sqsubseteq \{z\}\sharp$
⟨*proof*⟩

lemma *upper-unit-below-iff* [*simp*]: $\{x\}\sharp \sqsubseteq \{y\}\sharp \longleftrightarrow x \sqsubseteq y$
⟨*proof*⟩

lemmas *upper-pd-below-simps* =
upper-unit-below-iff
upper-below-plus-iff
upper-plus-below-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*⟩

29.4 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*⟩

29.5 Monadic bind

definition
upper-bind-basis ::
$'a$ *pd-basis* ⇒ ($'a \to 'b$ *upper-pd*) → $'b$ *upper-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*:
class.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-bind-basis* $t{\cdot}f +\sharp$ *upper-bind-basis* $u{\cdot}f$)
⟨*proof*⟩

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

definition
upper-bind :: $'a$ *upper-pd* → ($'a \to 'b$ *upper-pd*) → $'b$ *upper-pd* **where**
upper-bind = *upper-pd.basis-fun upper-bind-basis*

lemma *upper-bind-principal* [*simp*]:
upper-bind·(*upper-principal t*) = *upper-bind-basis t*
⟨*proof*⟩

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

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

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

29.6 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-ID*: $upper\text{-}map{\cdot}ID = ID$
⟨*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*⟩

lemma *ep-pair-upper-map*: *ep-pair e p* ⟹ *ep-pair* (*upper-map*·*e*) (*upper-map*·*p*)
⟨*proof*⟩

lemma *deflation-upper-map*: *deflation d* ⟹ *deflation* (*upper-map*·*d*)
⟨*proof*⟩

end

30 LowerPD: Lower powerdomain

theory *LowerPD*
imports *CompactBasis*
begin

30.1 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-le*: *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*:

$(PDUnit\ a \leq\flat\ PDPlus\ t\ u) = (PDUnit\ a \leq\flat\ t \vee PDUnit\ a \leq\flat\ u)$
⟨*proof*⟩

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

lemma *lower-le-induct* [*induct set*: *lower-le*]:
assumes *le*: $t \leq\flat\ u$
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *2*: $\bigwedge t\ u\ a.\ P\ (PDUnit\ a)\ t \Longrightarrow P\ (PDUnit\ a)\ (PDPlus\ t\ u)$
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ v;\ P\ u\ v]\!] \Longrightarrow P\ (PDPlus\ t\ u)\ v$
shows $P\ t\ u$
⟨*proof*⟩

lemma *pd-take-lower-chain*:
$pd\text{-}take\ n\ t \leq\flat\ pd\text{-}take\ (Suc\ n)\ t$
⟨*proof*⟩

lemma *pd-take-lower-le*: $pd\text{-}take\ i\ t \leq\flat\ t$
⟨*proof*⟩

lemma *pd-take-lower-mono*:
$t \leq\flat\ u \Longrightarrow pd\text{-}take\ n\ t \leq\flat\ pd\text{-}take\ n\ u$
⟨*proof*⟩

30.2 Type definition

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

instantiation *lower-pd* :: (*profinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow Rep\text{-}lower\text{-}pd\ x \subseteq Rep\text{-}lower\text{-}pd\ y$

instance ⟨*proof*⟩
end

instance *lower-pd* :: (*profinite*) *po*
⟨*proof*⟩

instance *lower-pd* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-lower-pd-lub*:
$chain\ Y \Longrightarrow Rep\text{-}lower\text{-}pd\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ Rep\text{-}lower\text{-}pd\ (Y\ i))$
⟨*proof*⟩

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

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

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

interpretation *lower-pd*:
ideal-completion lower-le pd-take lower-principal Rep-lower-pd
⟨*proof*⟩

Lower powerdomain is pointed

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

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

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

Lower powerdomain is profinite

instantiation *lower-pd* :: (*profinite*) *profinite*
begin

definition
approx-lower-pd-def: *approx* = *lower-pd.completion-approx*

instance
⟨*proof*⟩

end

instance *lower-pd* :: (*bifinite*) *bifinite* ⟨*proof*⟩

lemma *approx-lower-principal* [*simp*]:
approx n·(*lower-principal t*) = *lower-principal* (*pd-take n t*)
⟨*proof*⟩

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

30.3 Monadic unit and plus

definition
lower-unit :: *'a → 'a lower-pd* **where**
lower-unit = compact-basis.basis-fun (*λa. 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*⟩

interpretation *lower-add*!: *semilattice lower-add* ⟨*proof*⟩

lemmas *lower-plus-assoc = lower-add.assoc*
lemmas *lower-plus-commute = lower-add.commute*
lemmas *lower-plus-absorb = lower-add.idem*
lemmas *lower-plus-left-commute = lower-add.left-commute*
lemmas *lower-plus-left-absorb = lower-add.left-idem*

Useful for *simp add*: *lower-plus-ac*

lemmas *lower-plus-ac* =
lower-plus-assoc lower-plus-commute lower-plus-left-commute

Useful for *simp only*: *lower-plus-aci*

lemmas *lower-plus-aci* =
lower-plus-ac lower-plus-absorb lower-plus-left-absorb

lemma *lower-plus-below1*: $xs \sqsubseteq xs +\flat ys$
⟨*proof*⟩

lemma *lower-plus-below2*: $ys \sqsubseteq xs +\flat ys$
⟨*proof*⟩

lemma *lower-plus-least*: $[\![xs \sqsubseteq zs; ys \sqsubseteq zs]\!] \Longrightarrow xs +\flat ys \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-plus-below-iff*:
$xs +\flat ys \sqsubseteq zs \longleftrightarrow xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-unit-below-plus-iff*:
$\{x\}\flat \sqsubseteq ys +\flat zs \longleftrightarrow \{x\}\flat \sqsubseteq ys \vee \{x\}\flat \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-unit-below-iff* [*simp*]: $\{x\}\flat \sqsubseteq \{y\}\flat \longleftrightarrow x \sqsubseteq y$
⟨*proof*⟩

lemmas *lower-pd-below-simps* =
lower-unit-below-iff
lower-plus-below-iff
lower-unit-below-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*⟧ ⟹ *compact* (*xs* +♭ *ys*)
⟨*proof*⟩

30.4 Induction rules

lemma *lower-pd-induct1*:
  assumes *P*: *adm P*
  assumes *unit*: ⋀*x*. *P* {*x*}♭
  assumes *insert*:
    ⋀*x ys*. ⟦*P* {*x*}♭; *P ys*⟧ ⟹ *P* ({*x*}♭ +♭ *ys*)
  shows *P* (*xs*::'*a lower-pd*)
⟨*proof*⟩

lemma *lower-pd-induct*:
  assumes *P*: *adm P*
  assumes *unit*: ⋀*x*. *P* {*x*}♭
  assumes *plus*: ⋀*xs ys*. ⟦*P xs*; *P ys*⟧ ⟹ *P* (*xs* +♭ *ys*)
  shows *P* (*xs*::'*a lower-pd*)
⟨*proof*⟩

30.5 Monadic bind

definition
  lower-bind-basis ::
  '*a pd-basis* ⇒ ('*a* → '*b lower-pd*) → '*b lower-pd* **where**
  lower-bind-basis = *fold-pd*
    (λ*a*. Λ *f*. *f*·(*Rep-compact-basis a*))
    (λ*x y*. Λ *f*. *x*·*f* +♭ *y*·*f*)

lemma *ACI-lower-bind*:
  class.ab-semigroup-idem-mult (λ*x y*. Λ *f*. *x*·*f* +♭ *y*·*f*)
⟨*proof*⟩

lemma *lower-bind-basis-simps* [*simp*]:
  lower-bind-basis (*PDUnit a*) =
    (Λ *f*. *f*·(*Rep-compact-basis a*))
  lower-bind-basis (*PDPlus t u*) =
    (Λ *f*. *lower-bind-basis t*·*f* +♭ *lower-bind-basis u*·*f*)
⟨*proof*⟩

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

definition
  lower-bind :: '*a lower-pd* → ('*a* → '*b lower-pd*) → '*b lower-pd* **where**
  lower-bind = *lower-pd.basis-fun lower-bind-basis*

lemma *lower-bind-principal* [*simp*]:
lower-bind·(*lower-principal t*) = *lower-bind-basis t*
⟨*proof*⟩

lemma *lower-bind-unit* [*simp*]:
lower-bind·{*x*}♭·*f* = *f*·*x*
⟨*proof*⟩

lemma *lower-bind-plus* [*simp*]:
lower-bind·(*xs* +♭ *ys*)·*f* = *lower-bind*·*xs*·*f* +♭ *lower-bind*·*ys*·*f*
⟨*proof*⟩

lemma *lower-bind-strict* [*simp*]: *lower-bind*·⊥·*f* = *f*·⊥
⟨*proof*⟩

30.6 Map and join

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

definition
lower-join :: '*a lower-pd lower-pd* → '*a lower-pd* **where**
lower-join = (Λ *xss*. *lower-bind*·*xss*·(Λ *xs*. *xs*))

lemma *lower-map-unit* [*simp*]:
lower-map·*f*·{*x*}♭ = {*f*·*x*}♭
⟨*proof*⟩

lemma *lower-map-plus* [*simp*]:
lower-map·*f*·(*xs* +♭ *ys*) = *lower-map*·*f*·*xs* +♭ *lower-map*·*f*·*ys*
⟨*proof*⟩

lemma *lower-join-unit* [*simp*]:
lower-join·{*xs*}♭ = *xs*
⟨*proof*⟩

lemma *lower-join-plus* [*simp*]:
lower-join·(*xss* +♭ *yss*) = *lower-join*·*xss* +♭ *lower-join*·*yss*
⟨*proof*⟩

lemma *lower-map-ident*: *lower-map*·(Λ *x*. *x*)·*xs* = *xs*
⟨*proof*⟩

lemma *lower-map-ID*: *lower-map*·*ID* = *ID*
⟨*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*⟩

lemma *ep-pair-lower-map*: *ep-pair e p ⟹ ep-pair (lower-map·e) (lower-map·p)*
⟨*proof*⟩

lemma *deflation-lower-map*: *deflation d ⟹ deflation (lower-map·d)*
⟨*proof*⟩

end

31 ConvexPD: Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

31.1 Basis preorder

definition
convex-le :: *'a pd-basis ⇒ 'a pd-basis ⇒ bool* (**infix** $\leq\natural$ *50*) **where**
convex-le = (λu 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*]: $PDUnit\ compact\text{-}bot \leq\natural t$
⟨*proof*⟩

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

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

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

lemma *convex-le-PDUnit-lemma1*:
$(PDUnit\ a \leq\natural t) = (\forall\, b{\in}Rep\text{-}pd\text{-}basis\ t.\ a \sqsubseteq b)$
⟨*proof*⟩

lemma *convex-le-PDUnit-PDPlus-iff* [*simp*]:
$(PDUnit\ a \leq\natural PDPlus\ t\ u) = (PDUnit\ a \leq\natural t \wedge PDUnit\ a \leq\natural u)$
⟨*proof*⟩

lemma *convex-le-PDUnit-lemma2*:
$(t \leq\natural PDUnit\ b) = (\forall\, a{\in}Rep\text{-}pd\text{-}basis\ t.\ a \sqsubseteq b)$
⟨*proof*⟩

lemma *convex-le-PDPlus-PDUnit-iff* [*simp*]:
$(PDPlus\ t\ u \leq\natural PDUnit\ a) = (t \leq\natural PDUnit\ a \wedge u \leq\natural PDUnit\ a)$
⟨*proof*⟩

lemma *convex-le-PDPlus-lemma*:
assumes z: $PDPlus\ t\ u \leq\natural z$
shows $\exists\, v\ w.\ z = PDPlus\ v\ w \wedge t \leq\natural v \wedge u \leq\natural w$
⟨*proof*⟩

lemma *convex-le-induct* [*induct set*: *convex-le*]:
assumes *le*: $t \leq\natural u$
assumes *2*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ u\ v]\!] \Longrightarrow P\ t\ v$
assumes *3*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *4*: $\bigwedge t\ u\ v\ w.\ [\![P\ t\ v;\ P\ u\ w]\!] \Longrightarrow P\ (PDPlus\ t\ u)\ (PDPlus\ v\ w)$
shows $P\ t\ u$
⟨*proof*⟩

lemma *pd-take-convex-chain*:
$pd\text{-}take\ n\ t \leq\natural pd\text{-}take\ (Suc\ n)\ t$
⟨*proof*⟩

lemma *pd-take-convex-le*: $pd\text{-}take\ i\ t \leq\natural t$
⟨*proof*⟩

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

31.2 Type definition

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

instantiation *convex-pd* :: (*profinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-convex-pd x* $\subseteq$ *Rep-convex-pd y*

instance ⟨*proof*⟩
end

instance *convex-pd* :: (*profinite*) *po*
⟨*proof*⟩

instance *convex-pd* :: (*profinite*) *cpo*
⟨*proof*⟩

lemma *Rep-convex-pd-lub*:
chain Y $\Longrightarrow$ *Rep-convex-pd* $(\bigsqcup i.\ Y\ i) = (\bigcup i.$ *Rep-convex-pd* $(Y\ i))$
⟨*proof*⟩

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

definition
convex-principal :: *'a pd-basis* ⇒ *'a convex-pd* **where**
convex-principal t = *Abs-convex-pd* $\{u.\ u \leq\natural t\}$

lemma *Rep-convex-principal*:
Rep-convex-pd (*convex-principal t*) = $\{u.\ u \leq\natural t\}$
⟨*proof*⟩

interpretation *convex-pd*:
ideal-completion convex-le pd-take convex-principal Rep-convex-pd
⟨*proof*⟩

Convex powerdomain is pointed

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

instance *convex-pd* :: (*bifinite*) *pcpo*

⟨*proof*⟩

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

Convex powerdomain is profinite

instantiation *convex-pd* :: (*profinite*) *profinite*
begin

definition
approx-convex-pd-def: *approx* = *convex-pd.completion-approx*

instance
⟨*proof*⟩

end

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

lemma *approx-convex-principal* [*simp*]:
approx n·(*convex-principal t*) = *convex-principal* (*pd-take n t*)
⟨*proof*⟩

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

31.3 Monadic unit and plus

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

definition
convex-plus :: *'a convex-pd* → *'a convex-pd* → *'a convex-pd* **where**
convex-plus = *convex-pd.basis-fun* (λ*t*. *convex-pd.basis-fun* (λ*u*.
convex-principal (*PDPlus t u*)))

abbreviation
convex-add :: *'a convex-pd* ⇒ *'a convex-pd* ⇒ *'a convex-pd*
(**infixl** +♮ *65*) **where**
xs +♮ *ys* == *convex-plus*·*xs*·*ys*

syntax
-*convex-pd* :: *args* ⇒ *'a convex-pd* ({-}♮)

translations
{*x*,*xs*}♮ == {*x*}♮ +♮ {*xs*}♮
{*x*}♮ == *CONST convex-unit*·*x*

lemma *convex-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♮ = *convex-principal* (*PDUnit a*)
⟨*proof*⟩

lemma *convex-plus-principal* [*simp*]:
convex-principal t +♮ *convex-principal u* = *convex-principal* (*PDPlus t u*)
⟨*proof*⟩

lemma *approx-convex-unit* [*simp*]:
approx n·{*x*}♮ = {*approx n*·*x*}♮
⟨*proof*⟩

lemma *approx-convex-plus* [*simp*]:
approx n·(*xs* +♮ *ys*) = *approx n*·*xs* +♮ *approx n*·*ys*
⟨*proof*⟩

interpretation *convex-add*!: *semilattice convex-add* ⟨*proof*⟩

lemmas *convex-plus-assoc* = *convex-add.assoc*
lemmas *convex-plus-commute* = *convex-add.commute*
lemmas *convex-plus-absorb* = *convex-add.idem*
lemmas *convex-plus-left-commute* = *convex-add.left-commute*
lemmas *convex-plus-left-absorb* = *convex-add.left-idem*

Useful for *simp add*: *convex-plus-ac*

lemmas *convex-plus-ac* =
convex-plus-assoc convex-plus-commute convex-plus-left-commute

Useful for *simp only*: *convex-plus-aci*

lemmas *convex-plus-aci* =
convex-plus-ac convex-plus-absorb convex-plus-left-absorb

lemma *convex-unit-below-plus-iff* [*simp*]:
{*x*}♮ ⊑ *ys* +♮ *zs* ⟷ {*x*}♮ ⊑ *ys* ∧ {*x*}♮ ⊑ *zs*
⟨*proof*⟩

lemma *convex-plus-below-unit-iff* [*simp*]:
xs +♮ *ys* ⊑ {*z*}♮ ⟷ *xs* ⊑ {*z*}♮ ∧ *ys* ⊑ {*z*}♮
⟨*proof*⟩

lemma *convex-unit-below-iff* [*simp*]: {*x*}♮ ⊑ {*y*}♮ ⟷ *x* ⊑ *y*
⟨*proof*⟩

lemma *convex-unit-eq-iff* [*simp*]: {*x*}♮ = {*y*}♮ ⟷ *x* = *y*
⟨*proof*⟩

lemma *convex-unit-strict* [*simp*]: {⊥}♮ = ⊥
⟨*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*⟩

31.4 Induction rules

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

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

31.5 Monadic bind

definition
convex-bind-basis ::
$'a$ *pd-basis* $\Rightarrow$ $('a \to 'b$ *convex-pd*$) \to 'b$ *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*:
class.ab-semigroup-idem-mult $(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\natural y{\cdot}f)$
⟨*proof*⟩

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

lemma *monofun-LAM*:
$[\![$*cont* f; *cont* g; $\bigwedge x.\ f\ x \sqsubseteq g\ x]\!] \Longrightarrow (\Lambda\ x.\ f\ x) \sqsubseteq (\Lambda\ x.\ g\ x)$

⟨*proof*⟩

lemma *convex-bind-basis-mono*:
t ≤♮ *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*⟩

31.6 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-ID*: *convex-map*·*ID* = *ID*
⟨*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*⟩

lemma *ep-pair-convex-map*:
ep-pair e p ⟹ *ep-pair* (*convex-map*·*e*) (*convex-map*·*p*)
⟨*proof*⟩

lemma *deflation-convex-map*: *deflation d* ⟹ *deflation* (*convex-map*·*d*)
⟨*proof*⟩

31.7 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: *t* ≤♮ *u* ⟹ *t* ≤♯ *u*
⟨*proof*⟩

definition
convex-to-upper :: *'a convex-pd* → *'a upper-pd* **where**
convex-to-upper = *convex-pd.basis-fun upper-principal*

lemma *convex-to-upper-principal* [*simp*]:
convex-to-upper·(*convex-principal t*) = *upper-principal t*
⟨*proof*⟩

lemma *convex-to-upper-unit* [*simp*]:
convex-to-upper·{*x*}♮ = {*x*}♯
⟨*proof*⟩

lemma *convex-to-upper-plus* [*simp*]:
convex-to-upper·(*xs* +♮ *ys*) = *convex-to-upper*·*xs* +♯ *convex-to-upper*·*ys*
⟨*proof*⟩

lemma *approx-convex-to-upper*:
approx i·(*convex-to-upper*·*xs*) = *convex-to-upper*·(*approx i*·*xs*)
⟨*proof*⟩

lemma *convex-to-upper-bind* [*simp*]:
convex-to-upper·(*convex-bind*·*xs*·*f*) =
upper-bind·(*convex-to-upper*·*xs*)·(*convex-to-upper oo f*)
⟨*proof*⟩

lemma *convex-to-upper-map* [*simp*]:
convex-to-upper·(*convex-map*·*f*·*xs*) = *upper-map*·*f*·(*convex-to-upper*·*xs*)
⟨*proof*⟩

lemma *convex-to-upper-join* [*simp*]:
convex-to-upper·(*convex-join*·*xss*) =
upper-bind·(*convex-to-upper*·*xss*)·*convex-to-upper*
⟨*proof*⟩

Convex to lower

lemma *convex-le-imp-lower-le*: *t* ≤♮ *u* ⟹ *t* ≤♭ *u*
⟨*proof*⟩

definition
convex-to-lower :: *'a convex-pd* → *'a lower-pd* **where**
convex-to-lower = *convex-pd.basis-fun lower-principal*

lemma *convex-to-lower-principal* [*simp*]:
convex-to-lower·(*convex-principal t*) = *lower-principal t*
⟨*proof*⟩

lemma *convex-to-lower-unit* [*simp*]:
convex-to-lower·{*x*}♮ = {*x*}♭
⟨*proof*⟩

lemma *convex-to-lower-plus* [*simp*]:
convex-to-lower·(*xs* +♮ *ys*) = *convex-to-lower*·*xs* +♭ *convex-to-lower*·*ys*
⟨*proof*⟩

lemma *approx-convex-to-lower*:
approx i·(*convex-to-lower*·*xs*) = *convex-to-lower*·(*approx i*·*xs*)
⟨*proof*⟩

lemma *convex-to-lower-bind* [*simp*]:
convex-to-lower·(*convex-bind*·*xs*·*f*) =
lower-bind·(*convex-to-lower*·*xs*)·(*convex-to-lower oo f*)
⟨*proof*⟩

lemma *convex-to-lower-map* [*simp*]:
convex-to-lower·(*convex-map*·*f*·*xs*) = *lower-map*·*f*·(*convex-to-lower*·*xs*)
⟨*proof*⟩

lemma *convex-to-lower-join* [*simp*]:
convex-to-lower·(*convex-join*·*xss*) =
lower-bind·(*convex-to-lower*·*xss*)·*convex-to-lower*
⟨*proof*⟩

Ordering property

lemma *convex-pd-below-iff*:
(*xs* ⊑ *ys*) =
(*convex-to-upper*·*xs* ⊑ *convex-to-upper*·*ys* ∧
convex-to-lower·*xs* ⊑ *convex-to-lower*·*ys*)
⟨*proof*⟩

lemmas *convex-plus-below-plus-iff* =
convex-pd-below-iff [**where** *xs*=*xs* +♮ *ys* **and** *ys*=*zs* +♮ *ws*, *standard*]

lemmas *convex-pd-below-simps* =
convex-unit-below-plus-iff
convex-plus-below-unit-iff
convex-plus-below-plus-iff
convex-unit-below-iff
convex-to-upper-unit
convex-to-upper-plus
convex-to-lower-unit
convex-to-lower-plus
upper-pd-below-simps
lower-pd-below-simps

end

32 Powerdomains: Powerdomains

theory *Powerdomains*
imports *Representable ConvexPD*
begin

32.1 Powerdomains are representable

Upper powerdomain of a representable type is representable.

instantiation *upper-pd* :: (*rep*) *rep*
begin

definition *emb-upper-pd-def*: *emb* = *udom-emb oo upper-map·emb*
definition *prj-upper-pd-def*: *prj* = *upper-map·prj oo udom-prj*

instance
⟨*proof*⟩

end

Lower powerdomain of a representable type is representable.

instantiation *lower-pd* :: (*rep*) *rep*
begin

definition *emb-lower-pd-def*: *emb* = *udom-emb oo lower-map·emb*
definition *prj-lower-pd-def*: *prj* = *lower-map·prj oo udom-prj*

instance
⟨*proof*⟩

end

Convex powerdomain of a representable type is representable.

instantiation *convex-pd* :: (*rep*) *rep*
begin

definition *emb-convex-pd-def*: *emb* = *udom-emb oo convex-map·emb*
definition *prj-convex-pd-def*: *prj* = *convex-map·prj oo udom-prj*

instance
⟨*proof*⟩

end

32.2 Finite deflation lemmas

TODO: move these lemmas somewhere else

lemma *finite-compact-range-imp-finite-range*:
fixes *d* :: *'a*::*profinite* → *'b*::*cpo*
assumes *finite* ((λ*x*. *d*·*x*) ‘ {*x*. *compact x*})
shows *finite* (*range* (λ*x*. *d*·*x*))
⟨*proof*⟩

lemma *finite-deflation-upper-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*upper-map*·*d*)
⟨*proof*⟩

lemma *finite-deflation-lower-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*lower-map*·*d*)
⟨*proof*⟩

lemma *finite-deflation-convex-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*convex-map*·*d*)
⟨*proof*⟩

32.3 Deflation combinators

definition *upper-defl* = *TypeRep-fun1 upper-map*
definition *lower-defl* = *TypeRep-fun1 lower-map*
definition *convex-defl* = *TypeRep-fun1 convex-map*

lemma *cast-upper-defl*:
cast·(*upper-defl*·*A*) = *udom-emb oo upper-map*·(*cast*·*A*) *oo udom-prj*
⟨*proof*⟩

lemma *cast-lower-defl*:
cast·(*lower-defl*·*A*) = *udom-emb oo lower-map*·(*cast*·*A*) *oo udom-prj*
⟨*proof*⟩

lemma *cast-convex-defl*:
cast·(*convex-defl*·*A*) = *udom-emb oo convex-map*·(*cast*·*A*) *oo udom-prj*
⟨*proof*⟩

lemma *REP-upper*: *REP*('*a upper-pd*) = *upper-defl*·*REP*('*a*)
⟨*proof*⟩

lemma *REP-lower*: *REP*('*a lower-pd*) = *lower-defl*·*REP*('*a*)
⟨*proof*⟩

lemma *REP-convex*: *REP*('*a convex-pd*) = *convex-defl*·*REP*('*a*)
⟨*proof*⟩

lemma *isodefl-upper*:
isodefl d t ⟹ *isodefl* (*upper-map*·*d*) (*upper-defl*·*t*)
⟨*proof*⟩

lemma *isodefl-lower*:
isodefl d t ⟹ *isodefl* (*lower-map*·*d*) (*lower-defl*·*t*)
⟨*proof*⟩

lemma *isodefl-convex*:
isodefl d t ⟹ *isodefl* (*convex-map*·*d*) (*convex-defl*·*t*)
⟨*proof*⟩

32.4 Domain package setup for powerdomains

⟨*ML*⟩

end

theory *HOLCF*
imports
Main
Domain
Powerdomains
begin

default-sort *pcpo*

⟨*ML*⟩

Legacy theorem names

lemmas *sq-ord-less-eq-trans* = *below-eq-trans*
lemmas *sq-ord-eq-less-trans* = *eq-below-trans*
lemmas *refl-less* = *below-refl*
lemmas *trans-less* = *below-trans*
lemmas *antisym-less* = *below-antisym*
lemmas *antisym-less-inverse* = *below-antisym-inverse*
lemmas *box-less* = *box-below*
lemmas *rev-trans-less* = *rev-below-trans*
lemmas *not-less2not-eq* = *not-below2not-eq*
lemmas *less-UU-iff* = *below-UU-iff*
lemmas *flat-less-iff* = *flat-below-iff*
lemmas *adm-less* = *adm-below*
lemmas *adm-not-less* = *adm-not-below*
lemmas *adm-compact-not-less* = *adm-compact-not-below*
lemmas *less-fun-def* = *below-fun-def*
lemmas *expand-fun-less* = *expand-fun-below*
lemmas *less-fun-ext* = *below-fun-ext*
lemmas *less-discr-def* = *below-discr-def*
lemmas *discr-less-eq* = *discr-below-eq*
lemmas *less-unit-def* = *below-unit-def*
lemmas *less-cprod-def* = *below-prod-def*
lemmas *prod-lessI* = *prod-belowI*
lemmas *Pair-less-iff* = *Pair-below-iff*
lemmas *fst-less-iff* = *fst-below-iff*
lemmas *snd-less-iff* = *snd-below-iff*
lemmas *expand-cfun-less* = *expand-cfun-below*
lemmas *less-cfun-ext* = *below-cfun-ext*
lemmas *injection-less* = *injection-below*
lemmas *approx-less* = *approx-below*
lemmas *profinite-less-ext* = *profinite-below-ext*
lemmas *less-up-def* = *below-up-def*
lemmas *not-Iup-less* = *not-Iup-below*

lemmas *Iup-less = Iup-below*
lemmas *up-less = up-below*
lemmas *Def-inject-less-eq = Def-below-Def*
lemmas *Def-less-is-eq = Def-below-iff*
lemmas *spair-less-iff = spair-below-iff*
lemmas *less-sprod = below-sprod*
lemmas *spair-less = spair-below*
lemmas *sfst-less-iff = sfst-below-iff*
lemmas *ssnd-less-iff = ssnd-below-iff*
lemmas *fix-least-less = fix-least-below*
lemmas *dist-less-one = dist-below-one*
lemmas *less-ONE = below-ONE*
lemmas *ONE-less-iff = ONE-below-iff*
lemmas *less-sinlD = below-sinlD*
lemmas *less-sinrD = below-sinrD*

end