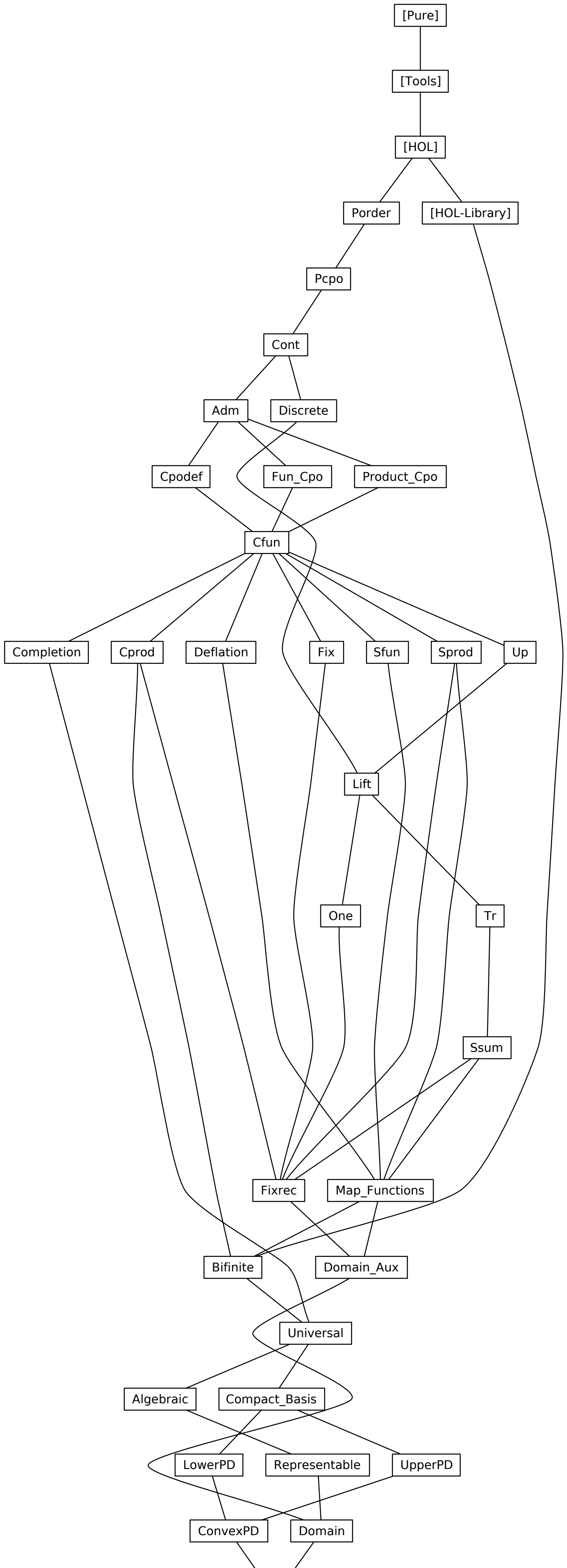
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

December 12, 2021

Contents


[Pure]
[Tools]
[HOL]
Porder
[HOL-Library]
Pcpo
Cont
Adm
Discrete
Cpodef
Fun_Cpo
Product_Cpo
Cfun
Completion
Cprod
Deflation
Fix
Sfun
Sprod
Up
Lift
One
Tr
Ssum
Fixrec
Map_Functions
Bifinite
Domain_Aux
Universal
Algebraic
Compact_Basis
LowerPD
Representable
UpperPD
ConvexPD
Domain

1 Partial orders

theory *Porder*
imports *Main*
begin

declare [[*typedef-overloaded*]]

1.1 Type class for partial orders

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

notation (*ASCII*)
below (**infix** << *50*)

notation
below (**infix** $\sqsubseteq$ *50*)

abbreviation *not-below* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\not\sqsubseteq$ *50*)
where *not-below* $x\ y \equiv \neg$ *below* $x\ y$

notation (*ASCII*)
not-below (**infix** ~<< *50*)

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

lemma *eq-below-trans*: $a = b \Longrightarrow b \sqsubseteq c \Longrightarrow a \sqsubseteq c$
⟨*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

lemma *eq-imp-below*: $x = y \Longrightarrow x \sqsubseteq y$
⟨*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*: $x \not\sqsubseteq y \Longrightarrow x \neq y$
⟨*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* :: $'a\ set \Rightarrow 'a \Rightarrow bool$ (**infix** $<|$ *55*)
where $S <| x \longleftrightarrow (\forall y \in S.\ y \sqsubseteq x)$

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

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

end

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

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

translations
$LUB\ x{:}A.\ t \rightleftharpoons CONST\ lub\ ((\lambda x.\ t)\ `\ A)$

context *po*
begin

abbreviation *Lub* (**binder** ⨆ *10*)
where $\bigsqcup n.\ t\ n \equiv lub\ (range\ t)$

notation (*ASCII*)
Lub (**binder** *LUB* *10*)

access to some definition as inference rule

lemma *is-lubD1*: $S <<| x \Longrightarrow S <| x$
⟨*proof*⟩

lemma *is-lubD2*: $[\![S <<| x;\ S <| u]\!] \Longrightarrow x \sqsubseteq u$
⟨*proof*⟩

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

lemma *is-lub-below-iff*: $S <<| x \Longrightarrow x \sqsubseteq u \longleftrightarrow S <| u$
⟨*proof*⟩

lubs are unique

lemma *is-lub-unique*: $S <<| x \Longrightarrow S <<| y \Longrightarrow x = y$
⟨*proof*⟩

technical lemmas about *lub* and $(<<|)$

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

lemma *lub-eqI*: $M <<| \; l \Longrightarrow lub\ M = l$
⟨*proof*⟩

lemma *is-lub-singleton* [*simp*]: $\{x\} <<| \; x$
⟨*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*: $S <| \; x \Longrightarrow x \in S \Longrightarrow S <<| \; x$
⟨*proof*⟩

lemma *lub-maximal*: $S <| \; x \Longrightarrow x \in S \Longrightarrow lub\ S = x$
⟨*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*: $chain\ Y \Longrightarrow i < j \Longrightarrow Y\ i \sqsubseteq Y\ j$
⟨*proof*⟩

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

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

technical lemmas about (least) upper bounds of chains

lemma *is-lub-rangeD1*: $range\ S <<| \; x \Longrightarrow S\ i \sqsubseteq x$
⟨*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 *is-lub-const*: $range\ (\lambda x.\ c) <<|\ c$
⟨*proof*⟩

lemma *lub-const* [*simp*]: $(\bigsqcup i.\ c) = c$
⟨*proof*⟩

1.5 Finite chains

definition *max-in-chain* :: $nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow bool$
where — finite chains, needed for monotony of continuous functions
$max\text{-}in\text{-}chain\ i\ C \longleftrightarrow (\forall j.\ i \le j \longrightarrow C\ i = C\ j)$

definition *finite-chain* :: $(nat \Rightarrow 'a) \Rightarrow bool$
where $finite\text{-}chain\ C = (chain\ C \wedge (\exists i.\ max\text{-}in\text{-}chain\ i\ C))$

results about finite chains

lemma *max-in-chainI*: $(\bigwedge j.\ i \le j \Longrightarrow Y\ i = Y\ j) \Longrightarrow max\text{-}in\text{-}chain\ i\ Y$
⟨*proof*⟩

lemma *max-in-chainD*: $max\text{-}in\text{-}chain\ i\ Y \Longrightarrow i \le j \Longrightarrow Y\ i = Y\ j$
⟨*proof*⟩

lemma *finite-chainI*: $chain\ C \Longrightarrow max\text{-}in\text{-}chain\ i\ C \Longrightarrow finite\text{-}chain\ C$
⟨*proof*⟩

lemma *finite-chainE*: $[\![finite\text{-}chain\ C;\ \bigwedge i.\ [\![chain\ C;\ max\text{-}in\text{-}chain\ i\ C]\!] \Longrightarrow R]\!] \Longrightarrow R$
⟨*proof*⟩

lemma *lub-finch1*: $chain\ C \Longrightarrow max\text{-}in\text{-}chain\ i\ C \Longrightarrow range\ C <<|\ C\ i$
⟨*proof*⟩

lemma *lub-finch2*: $finite\text{-}chain\ C \Longrightarrow range\ C <<|\ C\ (LEAST\ i.\ max\text{-}in\text{-}chain\ i\ C)$
⟨*proof*⟩

lemma *finch-imp-finite-range*: $finite\text{-}chain\ Y \Longrightarrow finite\ (range\ Y)$
⟨*proof*⟩

lemma *finite-range-has-max*:

fixes $f :: nat \Rightarrow 'a$
and $r :: 'a \Rightarrow 'a \Rightarrow bool$
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*: $chain\ Y \Longrightarrow finite\ (range\ Y) \Longrightarrow finite\text{-}chain\ Y$
⟨*proof*⟩

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

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

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

the maximal element in a chain is its lub

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

end

end

2 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-below-iff*: $chain\ S \Longrightarrow (\bigsqcup i.\ S\ i) \sqsubseteq x \longleftrightarrow (\forall i.\ S\ i \sqsubseteq x)$
⟨*proof*⟩

lemma *lub-below*: $[\![chain\ S;\ \bigwedge i.\ S\ i \sqsubseteq x]\!] \Longrightarrow (\bigsqcup i.\ S\ i) \sqsubseteq x$
⟨*proof*⟩

lemma *below-lub*: $[\![chain\ S;\ x \sqsubseteq S\ i]\!] \Longrightarrow x \sqsubseteq (\bigsqcup i.\ S\ i)$
⟨*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*: $chain\ Y \Longrightarrow max\text{-}in\text{-}chain\ i\ Y = ((\bigsqcup i.\ Y\ i) = Y\ i)$
⟨*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-eq*: $(\bigwedge i.\ X\ i = Y\ i) \Longrightarrow (\bigsqcup i.\ X\ i) = (\bigsqcup i.\ Y\ i)$
⟨*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 *bottom* :: $'a$ $(\bot)$
where $bottom = (THE\ x. \forall y. x \sqsubseteq y)$

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

end

Old "UU" syntax:

syntax *UU* :: *logic*
translations $UU \rightharpoonup CONST\ bottom$

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

⟨*ML*⟩

useful lemmas about $\bot$

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

lemma *eq-bottom-iff*: $x = \bot \longleftrightarrow x \sqsubseteq \bot$
⟨*proof*⟩

lemma *bottomI*: $x \sqsubseteq \bot \Longrightarrow x = \bot$
⟨*proof*⟩

lemma *lub-eq-bottom-iff*: $chain\ Y \Longrightarrow (\bigsqcup i. Y\ i) = \bot \longleftrightarrow (\forall i. Y\ i = \bot)$
⟨*proof*⟩

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

subclass *chfin*
⟨*proof*⟩

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

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

end

2.4 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 *chfin*
⟨*proof*⟩

end

end

3 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 $monofun\ f \longleftrightarrow (\forall\, x\ y.\ x \sqsubseteq y \longrightarrow f\ x \sqsubseteq f\ y)$

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*: $cont\ f \Longrightarrow chain\ Y \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i)$
  ⟨*proof*⟩

lemma *monofunI*: $(\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y) \Longrightarrow monofun\ f$
  ⟨*proof*⟩

lemma *monofunE*: $monofun\ f \Longrightarrow x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
  ⟨*proof*⟩

3.2 Equivalence of alternate definition

monotone functions map chains to chains

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

monotone functions map upper bound to upper bounds

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

a lemma about binary chains

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

continuity implies monotonicity

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

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

lemmas *ch2ch-cont* = *cont2mono* [*THEN ch2ch-monofun*] continuity implies preservation of lubs

lemma *cont2contlubE*: $\mathit{cont}\ f \Longrightarrow \mathit{chain}\ Y \Longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i))$
⟨*proof*⟩

lemma *contI2*:
fixes $f :: {}'a{::}\mathit{cpo} \Rightarrow {}'b{::}\mathit{cpo}$
assumes *mono*: *monofun f*
assumes *below*: $\bigwedge Y.\ [\![\mathit{chain}\ Y;\ \mathit{chain}\ (\lambda i.\ f\ (Y\ i))]\!] \Longrightarrow f\ (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ f\ (Y\ i))$
shows *cont f*
⟨*proof*⟩

3.3 Collection of continuity rules

named-theorems *cont2cont continuity intro rule*

3.4 Continuity of basic functions

The identity function is continuous

lemma *cont-id* [*simp*, *cont2cont*]: $\mathit{cont}\ (\lambda x.\ x)$
⟨*proof*⟩

constant functions are continuous

lemma *cont-const* [*simp*, *cont2cont*]: $\mathit{cont}\ (\lambda x.\ c)$
⟨*proof*⟩

application of functions is continuous

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

lemma *cont-compose*: $\mathit{cont}\ c \Longrightarrow \mathit{cont}\ (\lambda x.\ f\ x) \Longrightarrow \mathit{cont}\ (\lambda x.\ c\ (f\ x))$
⟨*proof*⟩

Least upper bounds preserve continuity

lemma *cont2cont-lub* [*simp*]:
assumes *chain*: $\bigwedge x.\ \mathit{chain}\ (\lambda i.\ F\ i\ x)$
and *cont*: $\bigwedge i.\ \mathit{cont}\ (\lambda x.\ F\ i\ x)$
shows $\mathit{cont}\ (\lambda x.\ \bigsqcup i.\ F\ i\ x)$
⟨*proof*⟩

if-then-else is continuous

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

3.5 Finite chains and flat pcpos

Monotone functions map finite chains to finite chains.

lemma *monofun-finch2finch*: *monofun f* $\Longrightarrow$ *finite-chain Y* $\Longrightarrow$ *finite-chain* ($\lambda n.$ f (Y n))
⟨*proof*⟩

The same holds for continuous functions.

lemma *cont-finch2finch*: *cont f* $\Longrightarrow$ *finite-chain Y* $\Longrightarrow$ *finite-chain* ($\lambda n.$ f (Y n))
⟨*proof*⟩

All monotone functions with chain-finite domain are continuous.

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

All strict functions with flat domain are continuous.

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

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

All functions with discrete domain are continuous.

lemma *cont-discrete-cpo* [*simp*, *cont2cont*]: *cont f*
for f :: *'a::discrete-cpo* $\Rightarrow$ *'b::cpo*
⟨*proof*⟩

end

4 Admissibility and compactness

theory *Adm*
imports *Cont*
begin

default-sort *cpo*

4.1 Definitions

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

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

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

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

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

4.2 Admissibility on chain-finite types

For chain-finite (easy) types every formula is admissible.

lemma *adm-chfin* [*simp*]: $adm\ P$
for $P :: {'a}{::}chfin \Rightarrow bool$
⟨*proof*⟩

4.3 Admissibility of special formulae and propagation

lemma *adm-const* [*simp*]: $adm\ (\lambda x.\ t)$
⟨*proof*⟩

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

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

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

Admissibility for disjunction is hard to prove. It requires 2 lemmas.

lemma *adm-disj-lemma1*:
assumes *adm*: $adm\ P$
assumes *chain*: $chain\ Y$
assumes *P*: $\forall\, i.\ \exists\, j{\geq}i.\ P\ (Y\ j)$
shows $P\ (\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *adm-disj-lemma2*: $\forall\, n{::}nat.\ P\ n \vee Q\ n \Longrightarrow (\forall\, i.\ \exists\, j{\geq}i.\ P\ j) \vee (\forall\, i.\ \exists\, j{\geq}i.\ Q\ j)$
⟨*proof*⟩

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

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

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

admissibility and continuity

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

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

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

lemma *adm-not-below* [*simp*]: $cont\ (\lambda x.\ t\ x) \Longrightarrow adm\ (\lambda x.\ t\ x \not\sqsubseteq u)$
⟨*proof*⟩

4.4 Compactness

definition $compact :: 'a{::}cpo \Rightarrow bool$
where $compact\ k = adm\ (\lambda x.\ k \not\sqsubseteq x)$

lemma *compactI*: $adm\ (\lambda x.\ k \not\sqsubseteq x) \Longrightarrow compact\ k$
⟨*proof*⟩

lemma *compactD*: $compact\ k \Longrightarrow adm\ (\lambda x.\ k \not\sqsubseteq x)$
⟨*proof*⟩

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

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

lemma *compact-below-lub-iff*: $compact\ x \Longrightarrow chain\ Y \Longrightarrow x \sqsubseteq (\bigsqcup i.\ Y\ i) \longleftrightarrow (\exists\, i.\ x \sqsubseteq Y\ i)$
⟨*proof*⟩

lemma *compact-chfin* [*simp*]: $compact\ x$
for $x :: 'a{::}chfin$
⟨*proof*⟩

lemma *compact-imp-max-in-chain*: $chain\ Y \Longrightarrow compact\ (\bigsqcup i.\ Y\ i) \Longrightarrow \exists\, i.\ max\text{-}in\text{-}chain$

i Y
⟨*proof*⟩

admissibility and compactness

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

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

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

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

Any upward-closed predicate is admissible.

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

lemmas *adm-lemmas* =
adm-const 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

end

5 Subtypes of pcpos

theory *Cpodef*
imports *Adm*
keywords *pcpodef cpodef* :: *thy-goal-defn*
begin

5.1 Proving a subtype is a partial order

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

⟨*ML*⟩

theorem *typedef-po*:
fixes *Abs* :: *'a::po* $\Rightarrow$ *'b::type*
assumes *type*: *type-definition Rep Abs A*

  and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
  shows $OFCLASS('b,\ po\text{-}class)$
  $\langle proof \rangle$

$\langle ML \rangle$

5.2 Proving a subtype is finite

lemma *typedef-finite-UNIV*:
  fixes $Abs :: 'a::type \Rightarrow 'b::type$
  assumes *type*: *type-definition Rep Abs A*
  shows $finite\ A \Longrightarrow finite\ (UNIV :: 'b\ set)$
$\langle proof \rangle$

5.3 Proving a subtype is chain-finite

lemma *ch2ch-Rep*:
  assumes *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
  shows $chain\ S \Longrightarrow chain\ (\lambda i.\ Rep\ (S\ i))$
  $\langle proof \rangle$

theorem *typedef-chfin*:
  fixes $Abs :: 'a::chfin \Rightarrow 'b::po$
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
  shows $OFCLASS('b,\ chfin\text{-}class)$
  $\langle proof \rangle$

5.4 Proving a subtype is complete

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

lemma *typedef-is-lubI*:
  assumes *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
  shows $range\ (\lambda i.\ Rep\ (S\ i)) <<|\ Rep\ x \Longrightarrow range\ S <<|\ x$
  $\langle proof \rangle$

lemma *Abs-inverse-lub-Rep*:
  fixes $Abs :: 'a::cpo \Rightarrow 'b::po$
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *adm*: $adm\ (\lambda x.\ x \in A)$
  shows $chain\ S \Longrightarrow Rep\ (Abs\ (\bigsqcup i.\ Rep\ (S\ i))) = (\bigsqcup i.\ Rep\ (S\ i))$
  $\langle proof \rangle$

theorem *typedef-is-lub*:
  fixes $Abs :: 'a::cpo \Rightarrow 'b::po$
  assumes *type*: *type-definition Rep Abs A*

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

lemmas *typedef-lub* = *typedef-is-lub* [*THEN lub-eqI*]

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

5.4.1 Continuity of *Rep* and *Abs*

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

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

5.5 Proving subtype elements are compact

theorem *typedef-compact*:
fixes $Abs :: 'a::cpo \Rightarrow 'b::cpo$
assumes *type*: *type-definition Rep Abs A*
and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
shows $compact\ (Rep\ k) \Longrightarrow compact\ k$
⟨*proof*⟩

5.6 Proving a subtype is pointed

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

theorem *typedef-pcpo-generic*:
  fixes *Abs* :: $'a$::*cpo* $\Rightarrow$ $'b$::*cpo*
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *z-in-A*: $z \in A$
    and *z-least*: $\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x$
  shows *OFCLASS*($'b$, *pcpo-class*)
  ⟨*proof*⟩

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

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

5.6.1 Strictness of *Rep* and *Abs*

For a sub-pcpo where $\perp$ 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*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *bottom-in-A*: $\perp \in A$
  shows $Abs\ \perp = \perp$
  ⟨*proof*⟩

theorem *typedef-Rep-strict*:
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *bottom-in-A*: $\perp \in A$
  shows $Rep\ \perp = \perp$
  ⟨*proof*⟩

theorem *typedef-Abs-bottom-iff*:
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *bottom-in-A*: $\perp \in A$
  shows $x \in A \Longrightarrow (Abs\ x = \perp) = (x = \perp)$
  ⟨*proof*⟩

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

5.7 Proving a subtype is flat

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

5.8 HOLCF type definition package

⟨*ML*⟩

end

6 Class instances for the full function space

theory *Fun-Cpo*
  imports *Adm*
begin

6.1 Full function space is a partial order

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

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

instance ⟨*proof*⟩
end

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

lemma *fun-below-iff*: $f \sqsubseteq g \longleftrightarrow (\forall x.\ f\ x \sqsubseteq g\ x)$
  ⟨*proof*⟩

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

lemma *fun-belowD*: $f \sqsubseteq g \Longrightarrow f\ x \sqsubseteq g\ x$

⟨*proof*⟩

6.2 Full function space is chain complete

Properties of chains of functions.

lemma *fun-chain-iff*: $chain\ S \longleftrightarrow (\forall x.\ chain\ (\lambda i.\ S\ i\ x))$
⟨*proof*⟩

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*⟩

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

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

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

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

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

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

6.3 Full function space is pointed

lemma *minimal-fun*: $(\lambda x.\ \bot) \sqsubseteq f$
⟨*proof*⟩

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

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

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

lemma *lambda-strict*: $(\lambda x.\ \bot) = \bot$
⟨*proof*⟩

6.4 Propagation of monotonicity and continuity

The lub of a chain of monotone functions is monotone.

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

The lub of a chain of continuous functions is continuous.

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

Function application preserves monotonicity and continuity.

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*⟩

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

Lambda abstraction preserves monotonicity and continuity. (Note $(\lambda x.\ \lambda y.\ f\ x\ y) = f$.)

lemma *mono2mono-lambda*: $(\bigwedge y.\ monofun\ (\lambda x.\ f\ x\ y)) \Longrightarrow monofun\ f$
⟨*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*: $(\bigwedge x.\ chain\ (\lambda i.\ S\ i\ x)) \Longrightarrow (\lambda x.\ \bigsqcup i.\ S\ i\ x) = (\bigsqcup i.\ (\lambda x.\ S\ i\ x))$
for $S :: nat \Rightarrow 'a::type \Rightarrow 'b::cpo$
⟨*proof*⟩

end

7 The cpo of cartesian products

theory *Product-Cpo*
imports *Adm*
begin

default-sort *cpo*

7.1 Unit type is a pcpo

instantiation *unit* :: *discrete-cpo*
begin

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

instance
⟨*proof*⟩

end

instance *unit* :: *pcpo*
⟨*proof*⟩

7.2 Product type is a partial order

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

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

instance ⟨*proof*⟩

end

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

7.3 Monotonicity of *Pair*, *fst*, *snd*

lemma *prod-belowI*: $fst\ p \sqsubseteq fst\ q \Longrightarrow snd\ p \sqsubseteq snd\ q \Longrightarrow p \sqsubseteq q$
⟨*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*: $x1 \sqsubseteq x2 \Longrightarrow y1 \sqsubseteq y2 \Longrightarrow (x1,\ y1) \sqsubseteq (x2,\ y2)$
⟨*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*: $x \sqsubseteq y \Longrightarrow \mathit{fst}\ x \sqsubseteq \mathit{fst}\ y$
⟨*proof*⟩

lemma *snd-monofun*: $x \sqsubseteq y \Longrightarrow \mathit{snd}\ x \sqsubseteq \mathit{snd}\ y$
⟨*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*: *chain Y*
obtains *A B*
where *chain A* **and** *chain B* **and** $Y = (\lambda i.\ (A\ i,\ B\ i))$
⟨*proof*⟩

7.4 Product type is a cpo

lemma *is-lub-Pair*: $\mathit{range}\ A <<|\ x \Longrightarrow \mathit{range}\ B <<|\ y \Longrightarrow \mathit{range}\ (\lambda i.\ (A\ i,\ B\ i)) <<|\ (x,\ y)$
⟨*proof*⟩

lemma *lub-Pair*: $\mathit{chain}\ A \Longrightarrow \mathit{chain}\ B \Longrightarrow (\bigsqcup i.\ (A\ i,\ B\ i)) = (\bigsqcup i.\ A\ i,\ \bigsqcup i.\ B\ i)$
for $A :: \mathit{nat} \Rightarrow {}'a::\mathit{cpo}$ **and** $B :: \mathit{nat} \Rightarrow {}'b::\mathit{cpo}$
⟨*proof*⟩

lemma *is-lub-prod*:
fixes $S :: \mathit{nat} \Rightarrow ({}'a::\mathit{cpo} \times {}'b::\mathit{cpo})$
assumes *chain S*
shows $\mathit{range}\ S <<|\ (\bigsqcup i.\ \mathit{fst}\ (S\ i),\ \bigsqcup i.\ \mathit{snd}\ (S\ i))$
⟨*proof*⟩

lemma *lub-prod*: $\mathit{chain}\ S \Longrightarrow (\bigsqcup i.\ S\ i) = (\bigsqcup i.\ \mathit{fst}\ (S\ i),\ \bigsqcup i.\ \mathit{snd}\ (S\ i))$
for $S :: \mathit{nat} \Rightarrow {}'a::\mathit{cpo} \times {}'b::\mathit{cpo}$
⟨*proof*⟩

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

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

7.5 Product type is pointed

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

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

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

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

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

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

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

lemma *split-strict* [*simp*]: $\mathit{case\text{-}prod}\ f\ \bot = f\ \bot\ \bot$
⟨*proof*⟩

7.6 Continuity of *Pair*, *fst*, *snd*

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

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

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

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

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

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

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

lemma *cont2cont-case-prod*:
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 $cont\ (\lambda x.\ case\ g\ x\ of\ (a,\ b) \Rightarrow f\ x\ a\ b)$
$\langle proof \rangle$

lemma *prod-contI*:
assumes $f1$: $\bigwedge y.\ cont\ (\lambda x.\ f\ (x,\ y))$
assumes $f2$: $\bigwedge x.\ cont\ (\lambda y.\ f\ (x,\ y))$
shows $cont\ f$
$\langle proof \rangle$

lemma *prod-cont-iff*: $cont\ f \longleftrightarrow (\forall\, y.\ cont\ (\lambda x.\ f\ (x,\ y))) \wedge (\forall\, x.\ cont\ (\lambda y.\ f\ (x,\ y)))$
$\langle proof \rangle$

lemma *cont2cont-case-prod'* [*simp*, *cont2cont*]:
assumes f: $cont\ (\lambda p.\ f\ (fst\ p)\ (fst\ (snd\ p))\ (snd\ (snd\ p)))$
assumes g: $cont\ (\lambda x.\ g\ x)$
shows $cont\ (\lambda x.\ case\text{-}prod\ (f\ x)\ (g\ x))$
$\langle proof \rangle$

The simple version (due to Joachim Breitner) is needed if either element type of the pair is not a cpo.

lemma *cont2cont-split-simple* [*simp*, *cont2cont*]:
assumes $\bigwedge a\ b.\ cont\ (\lambda x.\ f\ x\ a\ b)$
shows $cont\ (\lambda x.\ case\ p\ of\ (a,\ b) \Rightarrow f\ x\ a\ b)$
$\langle proof \rangle$

Admissibility of predicates on product types.

lemma *adm-case-prod* [*simp*]:
assumes $adm\ (\lambda x.\ P\ x\ (fst\ (f\ x))\ (snd\ (f\ x)))$
shows $adm\ (\lambda x.\ case\ f\ x\ of\ (a,\ b) \Rightarrow P\ x\ a\ b)$
$\langle proof \rangle$

7.7 Compactness and chain-finiteness

lemma *fst-below-iff*: $fst\ x \sqsubseteq y \longleftrightarrow x \sqsubseteq (y,\ snd\ x)$
for $x :: 'a \times 'b$
$\langle proof \rangle$

lemma *snd-below-iff*: $snd\ x \sqsubseteq y \longleftrightarrow x \sqsubseteq (fst\ x,\ y)$
for $x :: 'a \times 'b$
$\langle proof \rangle$

lemma *compact-fst*: $compact\ x \Longrightarrow compact\ (fst\ x)$
$\langle proof \rangle$

lemma *compact-snd*: *compact* $x \Longrightarrow$ *compact* (*snd* x)
⟨*proof*⟩

lemma *compact-Pair*: *compact* $x \Longrightarrow$ *compact* $y \Longrightarrow$ *compact* (x, y)
⟨*proof*⟩

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

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

end

8 The type of continuous functions

theory *Cfun*
imports *Cpodef Fun-Cpo Product-Cpo*
begin

default-sort *cpo*

8.1 Definition of continuous function type

definition *cfun* = $\{f::'a \Rightarrow 'b.\ cont\ f\}$

cpodef ('*a*, '*b*) *cfun* ((- →/ -) [*1*, *0*] *0*) = *cfun* :: ('*a* ⇒ '*b*) *set*
⟨*proof*⟩

type-notation (*ASCII*)
cfun (**infixr** −> *0*)

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

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

8.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: [*logic*, *logic*] ⇒ *logic*

⟨*ML*⟩

Syntax for nested abstractions

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

syntax
  -Lambda :: [*cargs*, *logic*] $\Rightarrow$ *logic* ((*3*Λ -./ -) [*1000*, *10*] *10*)

⟨*ML*⟩

Dummy patterns for continuous abstraction

translations
  Λ -. t $\rightharpoonup$ *CONST Abs-cfun* (λ-. t)

8.3 Continuous function space is pointed

lemma *bottom-cfun*: $\perp \in$ *cfun*
  ⟨*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 bottom-cfun*]

lemmas *Abs-cfun-strict* =
  typedef-Abs-strict [*OF type-definition-cfun below-cfun-def bottom-cfun*]

function application is strict in its first argument

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

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

for compatibility with old HOLCF-Version

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

8.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

8.4.1 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.

Update: The simproc now uses rule *Abs-cfun-inverse2* instead of *beta-cfun*, to avoid problems with eta-contraction.

⟨*ML*⟩

Eta-equality for continuous functions

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

Extensionality for continuous functions

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

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

Extensionality wrt. ordering for continuous functions

lemma *cfun-below-iff*: $f \sqsubseteq g \longleftrightarrow (\forall\, x.\ f\cdot x \sqsubseteq g\cdot x)$
⟨*proof*⟩

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

Congruence for continuous function application

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

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

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

8.5 Continuity of application

lemma *cont-Rep-cfun1*: *cont* $(\lambda f.\ f\cdot x)$
⟨*proof*⟩

lemma *cont-Rep-cfun2*: *cont* $(\lambda x.\ f\cdot x)$

⟨*proof*⟩

lemmas *monofun-Rep-cfun* = *cont-Rep-cfun* [*THEN cont2mono*]

lemmas *monofun-Rep-cfun1* = *cont-Rep-cfun1* [*THEN cont2mono*]
lemmas *monofun-Rep-cfun2* = *cont-Rep-cfun2* [*THEN cont2mono*]

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

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

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*]: $chain\ F \Longrightarrow chain\ Y \Longrightarrow chain\ (\lambda i.\ (F\ i)\cdot(Y\ i))$
⟨*proof*⟩

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

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

lemma *lub-APP*: $chain\ F \Longrightarrow chain\ Y \Longrightarrow (\bigsqcup i.\ F\ i\cdot(Y\ i)) = (\bigsqcup i.\ F\ i)\cdot(\bigsqcup i.\ Y\ i)$
⟨*proof*⟩

lemma *lub-LAM*:
assumes $\bigwedge x.\ chain\ (\lambda i.\ F\ i\ x)$

and $\bigwedge i.\ cont\ (\lambda x.\ F\ i\ x)$
shows $(\bigsqcup i.\ \Lambda\ x.\ F\ i\ x) = (\Lambda\ x.\ \bigsqcup i.\ F\ i\ x)$
$\langle proof \rangle$

lemmas *lub-distribs* = *lub-APP lub-LAM*

strictness

lemma *strictI*: $f \cdot x = \bot \Longrightarrow f \cdot \bot = \bot$
$\langle proof \rangle$

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

lemma *lub-cfun*: $chain\ F \Longrightarrow (\bigsqcup i.\ F\ i) = (\Lambda\ x.\ \bigsqcup i.\ F\ i \cdot x)$
$\langle proof \rangle$

8.6 Continuity simplification procedure

cont2cont lemma for *Rep-cfun*

lemma *cont2cont-APP* [*simp*, *cont2cont*]:
assumes *f*: $cont\ (\lambda x.\ f\ x)$
assumes *t*: $cont\ (\lambda x.\ t\ x)$
shows $cont\ (\lambda x.\ (f\ x) \cdot (t\ x))$
$\langle proof \rangle$

Two specific lemmas for the combination of LCF and HOL terms. These lemmas are needed in theories that use types like $'a \rightarrow 'b \Rightarrow 'c$.

lemma *cont-APP-app* [*simp*]: $cont\ f \Longrightarrow cont\ g \Longrightarrow cont\ (\lambda x.\ ((f\ x) \cdot (g\ x))\ s)$
$\langle proof \rangle$

lemma *cont-APP-app-app* [*simp*]: $cont\ f \Longrightarrow cont\ g \Longrightarrow cont\ (\lambda x.\ ((f\ x) \cdot (g\ x))\ s\ t)$
$\langle proof \rangle$

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

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

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 $cont\ (\lambda x.\ \Lambda\ y.\ f\ x\ y)$
$\langle proof \rangle$

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*⟩

8.7 Miscellaneous

Monotonicity of *Abs-cfun*

lemma *monofun-LAM*: $cont\ f \Longrightarrow cont\ g \Longrightarrow (\bigwedge x.\ f\ x \sqsubseteq g\ x) \Longrightarrow (\Lambda\ x.\ f\ x) \sqsubseteq (\Lambda\ x.\ g\ x)$
  ⟨*proof*⟩

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

lemma *chfin-Rep-cfunR*: $chain\ Y \Longrightarrow \forall s.\ \exists n.\ (LUB\ i.\ Y\ i)\cdot s = Y\ n\cdot s$
  for Y :: $nat \Rightarrow 'a::cpo \rightarrow 'b::chfin$
  ⟨*proof*⟩

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

8.8 Continuous injection-retraction pairs

Continuous retractions are strict.

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

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

lemma *injection-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*: $\forall x.\ f\cdot(g\cdot x) = x \Longrightarrow g\cdot z = \bot \Longrightarrow z = \bot$
  ⟨*proof*⟩

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

a result about functions with flat codomain

lemma *flat-eqI*: $x \sqsubseteq y \Longrightarrow x \neq \bot \Longrightarrow x = y$
  for $x\ y$ :: $'a::flat$

⟨*proof*⟩

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

8.9 Identity and composition

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

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

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

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

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

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

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

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

Show that interpretation of (pcpo, -→-) is a category.

- The class of objects is interpretation of syntactical class pcpo.
- The class of arrows between objects $'a$ and $'b$ is interpret. of $'a \rightarrow 'b$.
- 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*⟩

8.10 Strictified functions

default-sort *pcpo*

definition *seq* :: $'a \to 'b \to 'b$
  where $seq = (\Lambda\ x.\ if\ x = \bot\ then\ \bot\ else\ ID)$

lemma *cont2cont-if-bottom* [*cont2cont*, *simp*]:
  assumes *f*: $cont\ (\lambda x.\ f\ x)$
    and *g*: $cont\ (\lambda x.\ g\ x)$
  shows $cont\ (\lambda x.\ if\ f\ x = \bot\ then\ \bot\ else\ g\ x)$
⟨*proof*⟩

lemma *seq-conv-if*: $seq \cdot x = (if\ x = \bot\ then\ \bot\ else\ ID)$
  ⟨*proof*⟩

lemma *seq-simps* [*simp*]:
  $seq \cdot \bot = \bot$
  $seq \cdot x \cdot \bot = \bot$
  $x \neq \bot \Longrightarrow seq \cdot x = ID$
  ⟨*proof*⟩

definition *strictify* :: $('a \to 'b) \to 'a \to 'b$
  where $strictify = (\Lambda\ f\ x.\ seq \cdot x \cdot (f \cdot x))$

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*⟩

8.11 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*⟩

The simple version (suggested by Joachim Breitner) is needed if the type of the defined term is not a cpo.

lemma *cont2cont-Let-simple* [*simp*, *cont2cont*]:
  assumes $\bigwedge y.\ cont\ (\lambda x.\ g\ x\ y)$
  shows $cont\ (\lambda x.\ let\ y = t\ in\ g\ x\ y)$
  ⟨*proof*⟩

end

9 Continuous deflations and ep-pairs

theory *Deflation*
  imports *Cfun*
begin

default-sort *cpo*

9.1 Continuous deflations

locale *deflation* =
  fixes $d :: 'a \rightarrow 'a$
  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* ($\lambda d.$ *deflation* d)
⟨*proof*⟩

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

lemma *deflation-bottom*: *deflation* $\perp$
⟨*proof*⟩

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

9.2 Deflations with finite range

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

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

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

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

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

end

lemma *finite-deflation-intro*: *deflation* $d \Longrightarrow$ *finite* $\{x.\ d{\cdot}x = x\} \Longrightarrow$ *finite-deflation*

d
⟨*proof*⟩

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

lemma *finite-deflation-bottom*: *finite-deflation* $\perp$
⟨*proof*⟩

9.3 Continuous embedding-projection pairs

locale *ep-pair* =
fixes $e :: 'a \to 'b$ **and** $p :: 'b \to 'a$
assumes *e-inverse* [*simp*]: $\bigwedge x.\ p\cdot(e\cdot x) = x$
and *e-p-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*: $e\cdot(p\cdot x) = x \Longrightarrow e\cdot(p\cdot y) = y \Longrightarrow p\cdot x = p\cdot y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *p-inverse*: $(\exists\, x.\ y = e\cdot x) \longleftrightarrow e\cdot(p\cdot y) = y$
⟨*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*:
assumes *compact x*
shows *compact* $(e\cdot x)$
⟨*proof*⟩

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

Deflations from ep-pairs

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

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

shows *deflation* $(e\ oo\ d\ oo\ p)$
⟨*proof*⟩

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

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

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

end

9.4 Uniqueness of ep-pairs

lemma *ep-pair-unique-e-lemma*:
assumes *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* $\Longrightarrow$ *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* $\Longrightarrow$ *ep-pair e p2* $\Longrightarrow$ $p1 = p2$
⟨*proof*⟩

9.5 Composing ep-pairs

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

lemma *ep-pair-comp*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* $(e2\ oo\ e1)\ (p1\ oo\ p2)$

⟨*proof*⟩

locale *pcpo-ep-pair* = *ep-pair e p*
  for e :: $'a$::*pcpo* $\rightarrow$ $'b$::*pcpo*
  and p :: $'b$::*pcpo* $\rightarrow$ $'a$::*pcpo*
begin

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

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

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

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

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

end

end

10 The type of strict products

theory *Sprod*
  imports *Cfun*
begin

default-sort *pcpo*

10.1 Definition of strict product type

definition *sprod* = $\{p::'a \times 'b.\ p = \bot \vee (fst\ p \neq \bot \wedge snd\ p \neq \bot)\}$

pcpodef $('a, 'b)$ *sprod* ((- ⊗/ -) [*21*,*20*] *20*) = *sprod* :: $('a \times 'b)$ *set*
  ⟨*proof*⟩

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

type-notation (*ASCII*)
  sprod (**infixr** ** *20*)

10.2 Definitions of constants

definition *sfst* :: $('a\ {*}{*}\ 'b) \rightarrow 'a$

where $sfst = (\Lambda\ p.\ fst\ (Rep\text{-}sprod\ p))$

definition $ssnd :: ('a ** 'b) \to 'b$
where $ssnd = (\Lambda\ p.\ snd\ (Rep\text{-}sprod\ p))$

definition $spair :: 'a \to 'b \to ('a ** 'b)$
where $spair = (\Lambda\ a\ b.\ Abs\text{-}sprod\ (seq{\cdot}b{\cdot}a,\ seq{\cdot}a{\cdot}b))$

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

syntax *-stuple* :: [*logic*, *args*] ⇒ *logic* ((*1*'(:-,/ -:')))
translations
$(:x,\ y,\ z:) \rightleftharpoons (:x,\ (:y,\ z:):)$
$(:x,\ y:) \rightleftharpoons CONST\ spair{\cdot}x{\cdot}y$

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

10.3 Case analysis

lemma *spair-sprod*: $(seq{\cdot}b{\cdot}a,\ seq{\cdot}a{\cdot}b) \in sprod$
⟨*proof*⟩

lemma *Rep-sprod-spair*: $Rep\text{-}sprod\ (:a,\ b:) = (seq{\cdot}b{\cdot}a,\ seq{\cdot}a{\cdot}b)$
⟨*proof*⟩

lemmas *Rep-sprod-simps* =
Rep-sprod-inject [*symmetric*] *below-sprod-def*
prod-eq-iff below-prod-def
Rep-sprod-strict Rep-sprod-spair

lemma *sprodE* [*case-names bottom spair*, *cases type*: *sprod*]:
obtains $p = \bot \mid x\ y$ **where** $p = (:x,\ y:)$ **and** $x \neq \bot$ **and** $y \neq \bot$
⟨*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*⟩

10.4 Properties of *spair*

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

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

lemma *spair-bottom-iff* [*simp*]: $(:x,\ y:) = \bot \longleftrightarrow x = \bot \lor y = \bot$
⟨*proof*⟩

lemma *spair-below-iff*: $(:a,\ b:) \sqsubseteq (:c,\ d:) \longleftrightarrow a = \bot \lor b = \bot \lor (a \sqsubseteq c \land b \sqsubseteq d)$
⟨*proof*⟩

lemma *spair-eq-iff*: $(:a,\ b:) = (:c,\ d:) \longleftrightarrow a = c \land b = d \lor (a = \bot \lor b = \bot) \land (c = \bot \lor d = \bot)$
⟨*proof*⟩

lemma *spair-strict*: $x = \bot \lor y = \bot \Longrightarrow (:x,\ y:) = \bot$
⟨*proof*⟩

lemma *spair-strict-rev*: $(:x,\ y:) \neq \bot \Longrightarrow x \neq \bot \land y \neq \bot$
⟨*proof*⟩

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

lemma *spair-defined-rev*: $(:x,\ y:) = \bot \Longrightarrow x = \bot \lor y = \bot$
⟨*proof*⟩

lemma *spair-below*: $x \neq \bot \Longrightarrow y \neq \bot \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) \longleftrightarrow x \sqsubseteq a \land y \sqsubseteq b$
⟨*proof*⟩

lemma *spair-eq*: $x \neq \bot \Longrightarrow y \neq \bot \Longrightarrow (:x,\ y:) = (:a,\ b:) \longleftrightarrow x = a \land y = b$
⟨*proof*⟩

lemma *spair-inject*: $x \neq \bot \Longrightarrow y \neq \bot \Longrightarrow (:x,\ y:) = (:a,\ b:) \Longrightarrow x = a \land y = b$
⟨*proof*⟩

lemma *inst-sprod-pcpo2*: $\bot = (:\bot,\ \bot:)$
⟨*proof*⟩

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

10.5 Properties of *sfst* and *ssnd*

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

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

lemma *sfst-spair* [*simp*]: $y \neq \bot \Longrightarrow sfst\cdot(:x,\ y:) = x$
⟨*proof*⟩

lemma *ssnd-spair* [*simp*]: $x \neq \bot \Longrightarrow ssnd\cdot(:x,\ y:) = y$
⟨*proof*⟩

lemma *sfst-bottom-iff* [*simp*]: $sfst \cdot p = \bot \longleftrightarrow p = \bot$
⟨*proof*⟩

lemma *ssnd-bottom-iff* [*simp*]: $ssnd \cdot p = \bot \longleftrightarrow p = \bot$
⟨*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 *spair-sfst-ssnd*: $(:sfst \cdot p,\ ssnd \cdot p:) = p$
⟨*proof*⟩

lemma *below-sprod*: $x \sqsubseteq y \longleftrightarrow sfst \cdot x \sqsubseteq sfst \cdot y \wedge ssnd \cdot x \sqsubseteq ssnd \cdot y$
⟨*proof*⟩

lemma *eq-sprod*: $x = y \longleftrightarrow sfst \cdot x = sfst \cdot y \wedge ssnd \cdot x = ssnd \cdot y$
⟨*proof*⟩

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

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

10.6 Compactness

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

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

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

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

10.7 Properties of *ssplit*

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

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

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

10.8 Strict product preserves flatness

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

end

11 Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

11.1 Discrete cpo class instance

instantiation *discr* :: (*type*) *discrete-cpo*
begin

definition $((\sqsubseteq) :: 'a\ discr \Rightarrow 'a\ discr \Rightarrow bool) = (=)$

instance
⟨*proof*⟩

end

11.2 *undiscr*

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

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

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

end

12 The type of lifted values

theory *Up*
imports *Cfun*
begin

default-sort *cpo*

12.1 Definition of new type for lifting

datatype $'a\ u\ \ ((\text{-}_\bot)\ [1000]\ 999) = Ibottom \mid Iup\ 'a$

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

12.2 Ordering on lifted cpo

instantiation $u :: (cpo)\ below$
begin

definition *below-up-def*:
  $(\sqsubseteq) \equiv$
    $(\lambda x\ y.$
      $(case\ x\ of$
        $Ibottom \Rightarrow True$
      $\mid Iup\ a \Rightarrow (case\ y\ of\ Ibottom \Rightarrow False \mid Iup\ b \Rightarrow a \sqsubseteq b)))$

instance ⟨*proof*⟩

end

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

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

12.4 Lifted cpo is a cpo

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

lemma *up-chain-lemma*:
  assumes Y: $chain\ Y$
  obtains $\forall i.\ Y\ i = Ibottom$

| $A\ k$ **where** $\forall\, i.\ Iup\ (A\ i) = Y\ (i + k)$ **and** *chain* A **and** *range* Y <<| Iup $(\bigsqcup i.\ A\ i)$
⟨*proof*⟩

instance u :: (*cpo*) *cpo*
⟨*proof*⟩

12.5 Lifted cpo is pointed

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

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

translations
case l *of* $XCONST\ up{\cdot}x \Rightarrow t \rightleftharpoons CONST\ fup{\cdot}(\Lambda\ x.\ t){\cdot}l$
case l *of* $(XCONST\ up :: 'a){\cdot}x \Rightarrow t \rightharpoonup CONST\ fup{\cdot}(\Lambda\ x.\ t){\cdot}l$
$\Lambda(XCONST\ up{\cdot}x).\ t \rightleftharpoons CONST\ fup{\cdot}(\Lambda\ x.\ t)$

continuous versions of lemmas for $'a_{\bot}$

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

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

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

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

lemma *not-up-less-UU*: $up{\cdot}x \not\sqsubseteq \bot$
⟨*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*]: $[\![p = \bot \Longrightarrow Q; \bigwedge x.\ p = up{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

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

lifting preserves chain-finiteness

lemma *up-chain-cases*:
assumes Y: *chain* Y
obtains $\forall i.\ Y\ i = \bot$
| $A\ k$ **where** $\forall i.\ up{\cdot}(A\ i) = Y\ (i + k)$ **and** *chain* A **and** $(\bigsqcup i.\ Y\ i) = up{\cdot}(\bigsqcup i.\ A\ i)$
⟨*proof*⟩

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

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

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

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

properties of fup

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

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

⟨*proof*⟩

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

end

13 Lifting types of class type to flat pcpo's

theory *Lift*
imports *Discrete Up*
begin

default-sort *type*

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

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

definition
Def :: $'a \Rightarrow 'a\ lift$ **where**
$Def\ x = Abs\text{-}lift\ (up \cdot (Discr\ x))$

13.1 Lift as a datatype

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

old-rep-datatype $\bot :: 'a\ lift\ Def$
⟨*proof*⟩

$\bot$ and *Def*

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*⟩

13.3 Continuity of *case-lift*

lemma *case-lift-eq*: $case\text{-}lift\ \bot\ f\ x = fup\cdot(\Lambda\ y.\ f\ (undiscr\ y))\cdot(Rep\text{-}lift\ x)$
⟨*proof*⟩

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

13.4 Further operations

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

translations
$\Lambda(XCONST\ Def\ x).\ t \Rightarrow CONST\ flift1\ (\lambda x.\ t)$
$\Lambda(CONST\ Def\ x).\ FLIFT\ y.\ t \Leftarrow FLIFT\ x\ y.\ t$
$\Lambda(CONST\ Def\ x).\ t \Leftarrow FLIFT\ x.\ t$

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

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

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

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

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

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

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

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

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

end

14 The type of lifted booleans

theory *Tr*
imports *Lift*
begin

14.1 Type definition and constructors

type-synonym *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*, *cases type*: *tr*]:
$[\![p = \bot \Longrightarrow Q;\ p = TT \Longrightarrow Q;\ p = FF \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

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

distinctness for type *tr*

lemma *dist-below-tr* [*simp*]:
$TT \not\sqsubseteq \bot\ FF \not\sqsubseteq \bot\ TT \not\sqsubseteq FF\ FF \not\sqsubseteq TT$
⟨*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*]: $x \not\sqsubseteq TT \longleftrightarrow x = FF$
⟨*proof*⟩

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

14.2 Case analysis

default-sort *pcpo*

definition *tr-case* :: $'a \to 'a \to tr \to 'a$
where *tr-case* = (Λ *t e* (*Def b*). *if b then t else e*)

abbreviation *cifte-syn* :: [*tr*, *'c*, *'c*] $\Rightarrow$ *'c* ((*If* (-)/ *then* (-)/ *else* (-)) [*0*, *0*, *60*] *60*)
where *If b then e1 else e2* $\equiv$ *tr-case·e1·e2·b*

translations
Λ (*XCONST TT*). *t* $\rightleftharpoons$ *CONST tr-case·t·*$\bot$
Λ (*XCONST FF*). *t* $\rightleftharpoons$ *CONST tr-case·*$\bot$*·t*

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

14.3 Boolean connectives

definition *trand* :: $tr \to tr \to tr$
where *andalso-def*: *trand* = (Λ *x y*. *If x then y else FF*)

abbreviation *andalso-syn* :: $tr \Rightarrow tr \Rightarrow tr$ (- *andalso* - [*36*,*35*] *35*)
where *x andalso y* $\equiv$ *trand·x·y*

definition *tror* :: $tr \to tr \to tr$
where *orelse-def*: *tror* = (Λ *x y*. *If x then TT else y*)

abbreviation *orelse-syn* :: $tr \Rightarrow tr \Rightarrow tr$ (- *orelse* - [*31*,*30*] *30*)
where *x orelse y* $\equiv$ *tror·x·y*

definition *neg* :: $tr \rightarrow tr$
  where $neg = \mathit{flift2}\ Not$

definition *If2* :: $tr \Rightarrow {'c} \Rightarrow {'c} \Rightarrow {'c}$
  where $If2\ Q\ x\ y = (If\ Q\ then\ x\ else\ y)$

tactic for tr-thms with case split

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

lemmas about andalso, orelse, neg and if

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

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

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

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

lemma *split-If2*: $P\ (If2\ Q\ x\ y) \longleftrightarrow ((Q = \bot \longrightarrow P\ \bot) \wedge (Q = TT \longrightarrow P\ x) \wedge (Q = FF \longrightarrow P\ y))$
  ⟨*proof*⟩

⟨*ML*⟩

14.4 Rewriting of HOLCF operations to HOL functions

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

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

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

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

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

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

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

14.5 Compactness

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

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

end

15 The type of strict sums

theory *Ssum*
imports *Tr*
begin

default-sort *pcpo*

15.1 Definition of strict sum type

definition *ssum* =
$\{p :: tr \times ('a \times 'b).\ p = \bot \lor$
$(fst\ p = TT \land fst\ (snd\ p) \neq \bot \land snd\ (snd\ p) = \bot) \lor$
$(fst\ p = FF \land fst\ (snd\ p) = \bot \land snd\ (snd\ p) \neq \bot)\}$

pcpodef $('a, 'b)\ ssum\ \ ((\text{-} \oplus\text{/ -})\ [21, 20]\ 20) = ssum :: (tr \times 'a \times 'b)\ set$
⟨*proof*⟩

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

type-notation (*ASCII*)
ssum (**infixr** ++ *10*)

15.2 Definitions of constructors

definition *sinl* :: $'a \rightarrow ('a ++ 'b)$
  where *sinl* = $(\Lambda\ a.\ Abs\text{-}ssum\ (seq\cdot a\cdot TT,\ a,\ \bot))$

definition *sinr* :: $'b \rightarrow ('a ++ 'b)$
  where *sinr* = $(\Lambda\ b.\ Abs\text{-}ssum\ (seq\cdot b\cdot FF,\ \bot,\ b))$

lemma *sinl-ssum*: $(seq\cdot a\cdot TT,\ a,\ \bot) \in ssum$
  ⟨*proof*⟩

lemma *sinr-ssum*: $(seq\cdot b\cdot FF,\ \bot,\ b) \in ssum$
  ⟨*proof*⟩

lemma *Rep-ssum-sinl*: $Rep\text{-}ssum\ (sinl\cdot a) = (seq\cdot a\cdot TT,\ a,\ \bot)$
  ⟨*proof*⟩

lemma *Rep-ssum-sinr*: $Rep\text{-}ssum\ (sinr\cdot b) = (seq\cdot b\cdot FF,\ \bot,\ b)$
  ⟨*proof*⟩

lemmas *Rep-ssum-simps* =
  Rep-ssum-inject [*symmetric*] *below-ssum-def*
  prod-eq-iff below-prod-def
  Rep-ssum-strict Rep-ssum-sinl Rep-ssum-sinr

15.3 Properties of *sinl* and *sinr*

Ordering

lemma *sinl-below* [*simp*]: $sinl\cdot x \sqsubseteq sinl\cdot y \longleftrightarrow x \sqsubseteq y$
  ⟨*proof*⟩

lemma *sinr-below* [*simp*]: $sinr\cdot x \sqsubseteq sinr\cdot y \longleftrightarrow x \sqsubseteq y$
  ⟨*proof*⟩

lemma *sinl-below-sinr* [*simp*]: $sinl\cdot x \sqsubseteq sinr\cdot y \longleftrightarrow x = \bot$
  ⟨*proof*⟩

lemma *sinr-below-sinl* [*simp*]: $sinr\cdot x \sqsubseteq sinl\cdot y \longleftrightarrow x = \bot$
  ⟨*proof*⟩

Equality

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

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

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

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

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

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

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

Compactness

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

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

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

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

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

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

15.4 Case analysis

lemma *ssumE* [*case-names bottom sinl sinr*, *cases type*: *ssum*]:
obtains $p = \bot$
| x **where** $p = sinl \cdot x$ **and** $x \neq \bot$
| y **where** $p = sinr \cdot y$ **and** $y \neq \bot$
⟨*proof*⟩

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

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

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

lemma *below-sinrD*: $p \sqsubseteq sinr \cdot x \Longrightarrow \exists\, y.\ p = sinr \cdot y \wedge y \sqsubseteq x$
⟨*proof*⟩

15.5 Case analysis combinator

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

translations
$case\ s\ of\ XCONST\ sinl \cdot x \Rightarrow t1 \mid XCONST\ sinr \cdot y \Rightarrow t2 \rightleftharpoons CONST\ sscase \cdot (\Lambda\ x.\ t1) \cdot (\Lambda\ y.\ t2) \cdot s$
$case\ s\ of\ (XCONST\ sinl :: 'a) \cdot x \Rightarrow t1 \mid XCONST\ sinr \cdot y \Rightarrow t2 \rightharpoonup CONST\ sscase \cdot (\Lambda\ x.\ t1) \cdot (\Lambda\ y.\ t2) \cdot s$

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

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

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

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

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

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

15.6 Strict sum preserves flatness

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

end

16 The Strict Function Type

theory *Sfun*
imports *Cfun*
begin

pcpodef ('*a*, '*b*) *sfun* (**infixr** →! *0*) = {*f* :: '*a* → '*b*. *f*·⊥ = ⊥}
⟨*proof*⟩

type-notation (*ASCII*)
sfun (**infixr** −>! *0*)

TODO: Define nice syntax for abstraction, application.

definition *sfun-abs* :: ('*a* → '*b*) → ('*a* →! '*b*)
where *sfun-abs* = (Λ *f*. *Abs-sfun* (*strictify*·*f*))

definition *sfun-rep* :: ('*a* →! '*b*) → '*a* → '*b*
where *sfun-rep* = (Λ *f*. *Rep-sfun f*)

lemma *sfun-rep-beta*: *sfun-rep*·*f* = *Rep-sfun f*
⟨*proof*⟩

lemma *sfun-rep-strict1* [*simp*]: *sfun-rep*·⊥ = ⊥
⟨*proof*⟩

lemma *sfun-rep-strict2* [*simp*]: *sfun-rep*·*f*·⊥ = ⊥
⟨*proof*⟩

lemma *strictify-cancel*: *f*·⊥ = ⊥ ⟹ *strictify*·*f* = *f*
⟨*proof*⟩

lemma *sfun-abs-sfun-rep* [*simp*]: *sfun-abs*·(*sfun-rep*·*f*) = *f*
⟨*proof*⟩

lemma *sfun-rep-sfun-abs* [*simp*]: *sfun-rep*·(*sfun-abs*·*f*) = *strictify*·*f*
⟨*proof*⟩

lemma *sfun-eq-iff*: *f* = *g* ⟷ *sfun-rep*·*f* = *sfun-rep*·*g*

⟨*proof*⟩

lemma *sfun-below-iff*: $f \sqsubseteq g \longleftrightarrow sfun\text{-}rep\cdot f \sqsubseteq sfun\text{-}rep\cdot g$
⟨*proof*⟩

end

17 Map functions for various types

theory *Map-Functions*
imports *Deflation Sprod Ssum Sfun Up*
begin

17.1 Map operator for continuous function space

default-sort *cpo*

definition *cfun-map* :: $('b \to 'a) \to ('c \to 'd) \to ('a \to 'c) \to ('b \to 'd)$
where $cfun\text{-}map = (\Lambda\ a\ b\ f\ x.\ b\cdot(f\cdot(a\cdot x)))$

lemma *cfun-map-beta* [*simp*]: $cfun\text{-}map\cdot a\cdot b\cdot f\cdot x = b\cdot(f\cdot(a\cdot x))$
⟨*proof*⟩

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

lemma *cfun-map-map*: $cfun\text{-}map\cdot f1\cdot g1\cdot(cfun\text{-}map\cdot f2\cdot g2\cdot p) = cfun\text{-}map\cdot(\Lambda\ x.\ f2\cdot(f1\cdot x))\cdot(\Lambda\ x.\ g1\cdot(g2\cdot x))\cdot p$
⟨*proof*⟩

lemma *ep-pair-cfun-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* $(cfun\text{-}map\cdot p1\cdot e2)$ $(cfun\text{-}map\cdot e1\cdot p2)$
⟨*proof*⟩

lemma *deflation-cfun-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* $(cfun\text{-}map\cdot d1\cdot d2)$
⟨*proof*⟩

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

lemma *finite-deflation-cfun-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* $(cfun\text{-}map\cdot d1\cdot d2)$

⟨*proof*⟩

Finite deflations are compact elements of the function space

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

17.2 Map operator for product type

definition *prod-map* :: $('a \to 'b) \to ('c \to 'd) \to 'a \times 'c \to 'b \times 'd$
where *prod-map* = $(\Lambda\ f\ g\ p.\ (f\cdot(fst\ p),\ g\cdot(snd\ p)))$

lemma *prod-map-Pair* [*simp*]: *prod-map*$\cdot f\cdot g\cdot(x,\ y) = (f\cdot x,\ g\cdot y)$
⟨*proof*⟩

lemma *prod-map-ID*: *prod-map*$\cdot$*ID*$\cdot$*ID* = *ID*
⟨*proof*⟩

lemma *prod-map-map*: *prod-map*$\cdot f1\cdot g1\cdot($*prod-map*$\cdot f2\cdot g2\cdot p)$ = *prod-map*$\cdot(\Lambda\ x.\ f1\cdot(f2\cdot x))\cdot(\Lambda\ x.\ g1\cdot(g2\cdot x))\cdot p$
⟨*proof*⟩

lemma *ep-pair-prod-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*prod-map*$\cdot e1\cdot e2$) (*prod-map*$\cdot p1\cdot p2$)
⟨*proof*⟩

lemma *deflation-prod-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*prod-map*$\cdot d1\cdot d2$)
⟨*proof*⟩

lemma *finite-deflation-prod-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*prod-map*$\cdot d1\cdot d2$)
⟨*proof*⟩

17.3 Map function for lifted cpo

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

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

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

lemma *u-map-ID*: *u-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 *u-map-oo*: $u\text{-}map{\cdot}(f\ oo\ g) = u\text{-}map{\cdot}f\ oo\ u\text{-}map{\cdot}g$
⟨*proof*⟩

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

17.4 Map function for strict products

default-sort *pcpo*

definition $sprod\text{-}map :: ('a \to 'b) \to ('c \to 'd) \to 'a \otimes 'c \to 'b \otimes 'd$
where $sprod\text{-}map = (\Lambda\ f\ g.\ ssplit{\cdot}(\Lambda\ x\ y.\ (:f{\cdot}x,\ g{\cdot}y:)))$

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

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

lemma *sprod-map-spair'*: $f{\cdot}\bot = \bot \Longrightarrow g{\cdot}\bot = \bot \Longrightarrow sprod\text{-}map{\cdot}f{\cdot}g{\cdot}(:x,\ y:) = (:f{\cdot}x,\ g{\cdot}y:)$
⟨*proof*⟩

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

lemma *sprod-map-map*:
$[\![f1{\cdot}\bot = \bot;\ g1{\cdot}\bot = \bot]\!] \Longrightarrow$
$sprod\text{-}map{\cdot}f1{\cdot}g1{\cdot}(sprod\text{-}map{\cdot}f2{\cdot}g2{\cdot}p) =$
$sprod\text{-}map{\cdot}(\Lambda\ x.\ f1{\cdot}(f2{\cdot}x)){\cdot}(\Lambda\ x.\ g1{\cdot}(g2{\cdot}x)){\cdot}p$
⟨*proof*⟩

lemma *ep-pair-sprod-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows $ep\text{-}pair\ (sprod\text{-}map{\cdot}e1{\cdot}e2)\ (sprod\text{-}map{\cdot}p1{\cdot}p2)$
⟨*proof*⟩

lemma *deflation-sprod-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* $(sprod\text{-}map\cdot d1\cdot d2)$
⟨*proof*⟩

lemma *finite-deflation-sprod-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* $(sprod\text{-}map\cdot d1\cdot d2)$
⟨*proof*⟩

17.5 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*:
$[\![f1\cdot\bot = \bot;\ g1\cdot\bot = \bot]\!] \Longrightarrow$
$ssum\text{-}map\cdot f1\cdot g1\cdot(ssum\text{-}map\cdot f2\cdot g2\cdot p) =$
$ssum\text{-}map\cdot(\Lambda\ x.\ f1\cdot(f2\cdot x))\cdot(\Lambda\ x.\ g1\cdot(g2\cdot x))\cdot p$
⟨*proof*⟩

lemma *ep-pair-ssum-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* $(ssum\text{-}map\cdot e1\cdot e2)$ $(ssum\text{-}map\cdot p1\cdot p2)$
⟨*proof*⟩

lemma *deflation-ssum-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* $(ssum\text{-}map\cdot d1\cdot d2)$
⟨*proof*⟩

lemma *finite-deflation-ssum-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* $(ssum\text{-}map\cdot d1\cdot d2)$
⟨*proof*⟩

17.6 Map operator for strict function space

definition *sfun-map* :: $('b \rightarrow 'a) \rightarrow ('c \rightarrow 'd) \rightarrow ('a \rightarrow! 'c) \rightarrow ('b \rightarrow! 'd)$
where $sfun\text{-}map = (\Lambda\ a\ b.\ sfun\text{-}abs\ oo\ cfun\text{-}map\cdot a\cdot b\ oo\ sfun\text{-}rep)$

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

lemma *sfun-map-map*:
assumes $f2\cdot\bot = \bot$ **and** $g2\cdot\bot = \bot$
shows $sfun\text{-}map\cdot f1\cdot g1\cdot(sfun\text{-}map\cdot f2\cdot g2\cdot p) =$
$sfun\text{-}map\cdot(\Lambda\ x.\ f2\cdot(f1\cdot x))\cdot(\Lambda\ x.\ g1\cdot(g2\cdot x))\cdot p$
⟨*proof*⟩

lemma *ep-pair-sfun-map*:
assumes *1*: *ep-pair e1 p1*
assumes *2*: *ep-pair e2 p2*
shows *ep-pair* $(sfun\text{-}map\cdot p1\cdot e2)$ $(sfun\text{-}map\cdot e1\cdot p2)$
⟨*proof*⟩

lemma *deflation-sfun-map*:
assumes *1*: *deflation d1*
assumes *2*: *deflation d2*
shows *deflation* $(sfun\text{-}map\cdot d1\cdot d2)$
⟨*proof*⟩

lemma *finite-deflation-sfun-map*:
assumes *finite-deflation d1*
and *finite-deflation d2*
shows *finite-deflation* $(sfun\text{-}map\cdot d1\cdot d2)$
⟨*proof*⟩

end

18 The cpo of cartesian products

theory *Cprod*
imports *Cfun*
begin

default-sort *cpo*

18.1 Continuous case function for unit type

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

translations
  $\Lambda()$. *t* $\rightleftharpoons$ *CONST unit-when*$\cdot$*t*

lemma *unit-when* [*simp*]: *unit-when*$\cdot a \cdot u = a$
  ⟨*proof*⟩

18.2 Continuous version of split function

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

translations
  Λ(*CONST Pair x y*). *t* $\rightleftharpoons$ *CONST csplit*$\cdot$(Λ *x y*. *t*)

abbreviation *cfst* :: $'a \times 'b \to 'a$
  where *cfst* $\equiv$ *Abs-cfun fst*

abbreviation *csnd* :: $'a \times 'b \to 'b$
  where *csnd* $\equiv$ *Abs-cfun snd*

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 Profinite and bifinite cpos

theory *Bifinite*
  imports *Map-Functions Cprod Sprod Sfun Up HOL−Library.Countable*
begin

default-sort *cpo*

19.1 Chains of finite deflations

locale *approx-chain* =
  fixes *approx* :: $nat \Rightarrow 'a \to 'a$
  assumes *chain-approx* [*simp*]: *chain* (λi. *approx i*)
  assumes *lub-approx* [*simp*]: ($\bigsqcup i$. *approx i*) = *ID*
  assumes *finite-deflation-approx* [*simp*]: $\bigwedge i$. *finite-deflation* (*approx i*)

begin

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

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

lemma *approx-below*: *approx i*·*x* $\sqsubseteq$ *x*
⟨*proof*⟩

lemma *finite-range-approx*: *finite* (*range* (λx. *approx i*·*x*))
⟨*proof*⟩

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

lemma *compact-eq-approx*: *compact x* $\Longrightarrow$ $\exists$ *i*. *approx i*·*x* = *x*
⟨*proof*⟩

end

19.2 Omega-profinite and bifinite domains

class *bifinite* = *pcpo* +
  assumes *bifinite*: $\exists (a::nat \Rightarrow 'a \rightarrow 'a)$. *approx-chain a*

class *profinite* = *cpo* +
  assumes *profinite*: $\exists (a::nat \Rightarrow 'a_\bot \rightarrow 'a_\bot)$. *approx-chain a*

19.3 Building approx chains

lemma *approx-chain-iso*:
  assumes *a*: *approx-chain a*
  assumes [*simp*]: $\bigwedge x.\ f\cdot(g\cdot x) = x$
  assumes [*simp*]: $\bigwedge y.\ g\cdot(f\cdot y) = y$
  shows *approx-chain* (λi. *f oo a i oo g*)
⟨*proof*⟩

lemma *approx-chain-u-map*:
  assumes *approx-chain a*
  shows *approx-chain* (λi. *u-map*·(*a i*))
  ⟨*proof*⟩

lemma *approx-chain-sfun-map*:
  assumes *approx-chain a* **and** *approx-chain b*
  shows *approx-chain* (λi. *sfun-map*·(*a i*)·(*b i*))
  ⟨*proof*⟩

lemma *approx-chain-sprod-map*:

assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i.\ sprod\text{-}map\cdot(a\ i)\cdot(b\ i))$
⟨*proof*⟩

lemma *approx-chain-ssum-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i.\ ssum\text{-}map\cdot(a\ i)\cdot(b\ i))$
⟨*proof*⟩

lemma *approx-chain-cfun-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i.\ cfun\text{-}map\cdot(a\ i)\cdot(b\ i))$
⟨*proof*⟩

lemma *approx-chain-prod-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i.\ prod\text{-}map\cdot(a\ i)\cdot(b\ i))$
⟨*proof*⟩

Approx chains for countable discrete types.

definition *discr-approx* :: $nat \Rightarrow {}'a{::}countable\ discr\ u \rightarrow {}'a\ discr\ u$
where *discr-approx* $= (\lambda i.\ \Lambda(up\cdot x).\ if\ to\text{-}nat\ (undiscr\ x) < i\ then\ up\cdot x\ else\ \bot)$

lemma *chain-discr-approx* [*simp*]: *chain discr-approx*
⟨*proof*⟩

lemma *lub-discr-approx* [*simp*]: $(\bigsqcup i.\ discr\text{-}approx\ i) = ID$
⟨*proof*⟩

lemma *inj-on-undiscr* [*simp*]: *inj-on undiscr A*
⟨*proof*⟩

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

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

19.4 Class instance proofs

instance *bifinite* ⊆ *profinite*
⟨*proof*⟩

instance *u* :: (*profinite*) *bifinite*
⟨*proof*⟩

Types ${}'a \rightarrow {}'b$ and ${}'a_\bot \rightarrow! {}'b$ are isomorphic.

definition *encode-cfun* $= (\Lambda\ f.\ sfun\text{-}abs\cdot(fup\cdot f))$

definition *decode-cfun* = $(\Lambda\ g\ x.\ sfun\text{-}rep\cdot g\cdot(up\cdot x))$

lemma *decode-encode-cfun* [*simp*]: $decode\text{-}cfun\cdot(encode\text{-}cfun\cdot x) = x$
⟨*proof*⟩

lemma *encode-decode-cfun* [*simp*]: $encode\text{-}cfun\cdot(decode\text{-}cfun\cdot y) = y$
⟨*proof*⟩

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

Types $('a \times 'b)_\perp$ and $'a_\perp \otimes 'b_\perp$ are isomorphic.

definition *encode-prod-u* = $(\Lambda(up\cdot(x,\ y)).\ (:up\cdot x,\ up\cdot y:))$

definition *decode-prod-u* = $(\Lambda(:up\cdot x,\ up\cdot y:).\ up\cdot(x,\ y))$

lemma *decode-encode-prod-u* [*simp*]: $decode\text{-}prod\text{-}u\cdot(encode\text{-}prod\text{-}u\cdot x) = x$
⟨*proof*⟩

lemma *encode-decode-prod-u* [*simp*]: $encode\text{-}prod\text{-}u\cdot(decode\text{-}prod\text{-}u\cdot y) = y$
⟨*proof*⟩

instance *prod* :: (*profinite*, *profinite*) *profinite*
⟨*proof*⟩

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

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

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

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

lemma *approx-chain-unit*: *approx-chain* $(\perp :: nat \Rightarrow unit \rightarrow unit)$
⟨*proof*⟩

instance *unit* :: *bifinite*
⟨*proof*⟩

instance *discr* :: (*countable*) *profinite*
⟨*proof*⟩

instance *lift* :: (*countable*) *bifinite*
⟨*proof*⟩

end

20 Defining algebraic domains by ideal completion

theory *Completion*
imports *Cfun*
begin

20.1 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*: $[\![x \preceq y;\ y \preceq z]\!] \Longrightarrow x \preceq z$
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 $\bigwedge x\ y.\ [\![x \in A;\ y \in A]\!] \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
  assumes $\bigwedge x\ y.\ [\![x \preceq y;\ y \in A]\!] \Longrightarrow x \in A$
  shows *ideal A*
⟨*proof*⟩

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

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

lemma *idealD3*:
  $[\![ideal\ A;\ x \preceq y;\ y \in A]\!] \Longrightarrow x \in A$
⟨*proof*⟩

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

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

The set of ideals is a cpo

lemma *ideal-UN*:
  fixes A :: *nat* $\Rightarrow$ $'a$ *set*

assumes *ideal-A*: $\bigwedge i.\ ideal\ (A\ i)$
assumes *chain-A*: $\bigwedge i\ j.\ i \le j \Longrightarrow A\ i \subseteq A\ j$
shows *ideal* $(\bigcup i.\ A\ i)$
⟨*proof*⟩

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

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

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

end

interpretation *below*: $preorder\ below :: 'a{::}po \Rightarrow 'a \Rightarrow bool$
⟨*proof*⟩

20.2 Lemmas about least upper bounds

lemma *is-ub-thelub-ex*: $[\![\exists u.\ S <<|\ u;\ x \in S]\!] \Longrightarrow x \sqsubseteq lub\ S$
⟨*proof*⟩

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

20.3 Locale for ideal completion

hide-const (**open**) *Filter.principal*

locale *ideal-completion* = *preorder* +
fixes $principal :: 'a{::}type \Rightarrow 'b{::}cpo$
fixes $rep :: 'b{::}cpo \Rightarrow 'a{::}type\ set$
assumes *ideal-rep*: $\bigwedge x.\ ideal\ (rep\ x)$
assumes *rep-lub*: $\bigwedge Y.\ chain\ Y \Longrightarrow rep\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ rep\ (Y\ i))$

assumes *rep-principal*: $\bigwedge a.\ rep\ (principal\ a) = \{b.\ b \preceq a\}$
assumes *belowI*: $\bigwedge x\ y.\ rep\ x \subseteq rep\ y \Longrightarrow x \sqsubseteq y$
assumes *countable*: $\exists f{::}'a \Rightarrow nat.\ inj\ f$
begin

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 *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 *eq-iff*: $x = y \longleftrightarrow rep\ x = rep\ y$
⟨*proof*⟩

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

lemma *ch2ch-principal* [*simp*]:
$\forall i.\ Y\ i \preceq Y\ (Suc\ i) \Longrightarrow chain\ (\lambda i.\ principal\ (Y\ i))$
⟨*proof*⟩

20.3.1 Principal ideals approximate all elements

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

Construct a chain whose lub is the same as a given ideal

lemma *obtain-principal-chain*:
obtains Y **where** $\forall i.\ Y\ i \preceq Y\ (Suc\ i)$ **and** $x = (\bigsqcup i.\ principal\ (Y\ i))$
⟨*proof*⟩

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

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

20.4 Defining functions in terms of basis elements

definition
extension :: $('a::type \Rightarrow 'c::cpo) \Rightarrow 'b \rightarrow 'c$ **where**
extension = $(\lambda f.\ (\Lambda\ x.\ lub\ (f\ `\ rep\ x)))$

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

lemma *extension-beta*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $extension\ f \cdot x = lub\ (f\ `\ rep\ x)$
⟨*proof*⟩

lemma *extension-principal*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $extension\ f \cdot (principal\ a) = f\ a$
⟨*proof*⟩

lemma *extension-mono*:
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
assumes *g-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow g\ a \sqsubseteq g\ b$
assumes *below*: $\bigwedge a.\ f\ a \sqsubseteq g\ a$
shows $extension\ f \sqsubseteq extension\ g$
⟨*proof*⟩

lemma *cont-extension*:
assumes *f-mono*: $\bigwedge a\ b\ x.\ a \preceq b \Longrightarrow f\ x\ a \sqsubseteq f\ x\ b$
assumes *f-cont*: $\bigwedge a.\ cont\ (\lambda x.\ f\ x\ a)$
shows $cont\ (\lambda x.\ extension\ (\lambda a.\ f\ x\ a))$
⟨*proof*⟩

end

lemma (**in** *preorder*) *typedef-ideal-completion*:
fixes $Abs :: 'a\ set \Rightarrow 'b::cpo$
assumes *type*: $type\text{-}definition\ Rep\ Abs\ \{S.\ ideal\ S\}$
assumes *below*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow Rep\ x \subseteq Rep\ y$
assumes *principal*: $\bigwedge a.\ principal\ a = Abs\ \{b.\ b \preceq a\}$
assumes *countable*: $\exists f::'a \Rightarrow nat.\ inj\ f$
shows $ideal\text{-}completion\ r\ principal\ Rep$
⟨*proof*⟩

end

21 A universal bifinite domain

theory *Universal*
imports *Bifinite Completion HOL−Library.Nat-Bijection*
begin

no-notation *binomial* (**infixl** *choose 65*)

21.1 Basis for universal domain

21.1.1 Basis datatype

type-synonym *ubasis* = *nat*

definition
  $node :: nat \Rightarrow ubasis \Rightarrow ubasis\ set \Rightarrow ubasis$
where
  $node\ i\ a\ S = Suc\ (prod\text{-}encode\ (i,\ prod\text{-}encode\ (a,\ set\text{-}encode\ S)))$

lemma *node-not-0* [*simp*]: $node\ i\ a\ S \neq 0$
⟨*proof*⟩

lemma *node-gt-0* [*simp*]: $0 < node\ i\ a\ S$
⟨*proof*⟩

lemma *node-inject* [*simp*]:
  $[\![finite\ S;\ finite\ T]\!]$
    $\Longrightarrow node\ i\ a\ S = node\ j\ b\ T \longleftrightarrow i = j \wedge a = b \wedge S = T$
⟨*proof*⟩

lemma *node-gt0*: $i < node\ i\ a\ S$
⟨*proof*⟩

lemma *node-gt1*: $a < node\ i\ a\ S$
⟨*proof*⟩

lemma *nat-less-power2*: $n < 2\hat{}n$
  ⟨*proof*⟩

lemma *node-gt2*: $[\![finite\ S;\ b \in S]\!] \Longrightarrow b < node\ i\ a\ S$
⟨*proof*⟩

lemma *eq-prod-encode-pairI*:
  $[\![fst\ (prod\text{-}decode\ x) = a;\ snd\ (prod\text{-}decode\ x) = b]\!] \Longrightarrow x = prod\text{-}encode\ (a,\ b)$
⟨*proof*⟩

lemma *node-cases*:
  assumes *1*: $x = 0 \Longrightarrow P$
  assumes *2*: $\bigwedge i\ a\ S.\ [\![finite\ S;\ x = node\ i\ a\ S]\!] \Longrightarrow P$
  shows *P*

⟨*proof*⟩

lemma *node-induct*:
assumes *1*: $P\ 0$
assumes *2*: $\bigwedge i\ a\ S.\ [\![P\ a;\ finite\ S;\ \forall\, b{\in}S.\ P\ b]\!] \Longrightarrow P\ (node\ i\ a\ S)$
shows $P\ x$
⟨*proof*⟩

21.1.2 Basis ordering

inductive
ubasis-le :: $nat \Rightarrow nat \Rightarrow bool$
where
ubasis-le-refl: *ubasis-le a a*
| *ubasis-le-trans*:
$[\![ubasis\text{-}le\ a\ b;\ ubasis\text{-}le\ b\ c]\!] \Longrightarrow ubasis\text{-}le\ a\ c$
| *ubasis-le-lower*:
$finite\ S \Longrightarrow ubasis\text{-}le\ a\ (node\ i\ a\ S)$
| *ubasis-le-upper*:
$[\![finite\ S;\ b \in S;\ ubasis\text{-}le\ a\ b]\!] \Longrightarrow ubasis\text{-}le\ (node\ i\ a\ S)\ b$

lemma *ubasis-le-minimal*: *ubasis-le 0 x*
⟨*proof*⟩

interpretation *udom*: *preorder ubasis-le*
⟨*proof*⟩

21.1.3 Generic take function

function
ubasis-until :: $(ubasis \Rightarrow bool) \Rightarrow ubasis \Rightarrow ubasis$
where
$ubasis\text{-}until\ P\ 0 = 0$
| $finite\ S \Longrightarrow ubasis\text{-}until\ P\ (node\ i\ a\ S) =$
$(if\ P\ (node\ i\ a\ S)\ then\ node\ i\ a\ S\ else\ ubasis\text{-}until\ P\ a)$
⟨*proof*⟩

termination *ubasis-until*
⟨*proof*⟩

lemma *ubasis-until*: $P\ 0 \Longrightarrow P\ (ubasis\text{-}until\ P\ x)$
⟨*proof*⟩

lemma *ubasis-until'*: $0 < ubasis\text{-}until\ P\ x \Longrightarrow P\ (ubasis\text{-}until\ P\ x)$
⟨*proof*⟩

lemma *ubasis-until-same*: $P\ x \Longrightarrow ubasis\text{-}until\ P\ x = x$
⟨*proof*⟩

lemma *ubasis-until-idem*:

$P\ 0 \Longrightarrow$ *ubasis-until* P (*ubasis-until* $P\ x$) = *ubasis-until* $P\ x$
⟨*proof*⟩

lemma *ubasis-until-0*:
$\forall x.\ x \neq 0 \longrightarrow \neg\ P\ x \Longrightarrow$ *ubasis-until* $P\ x = 0$
⟨*proof*⟩

lemma *ubasis-until-less*: *ubasis-le* (*ubasis-until* $P\ x$) x
⟨*proof*⟩

lemma *ubasis-until-chain*:
assumes PQ: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows *ubasis-le* (*ubasis-until* $P\ x$) (*ubasis-until* $Q\ x$)
⟨*proof*⟩

lemma *ubasis-until-mono*:
assumes $\bigwedge i\ a\ S\ b.$ ⟦*finite* S; P (*node* $i\ a\ S$); $b \in S$; *ubasis-le* $a\ b$⟧ $\Longrightarrow P\ b$
shows *ubasis-le* $a\ b \Longrightarrow$ *ubasis-le* (*ubasis-until* $P\ a$) (*ubasis-until* $P\ b$)
⟨*proof*⟩

lemma *finite-range-ubasis-until*:
finite $\{x.\ P\ x\} \Longrightarrow$ *finite* (*range* (*ubasis-until* P))
⟨*proof*⟩

21.2 Defining the universal domain by ideal completion

typedef *udom* = $\{S.$ *udom.ideal* $S\}$
⟨*proof*⟩

instantiation *udom* :: *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-udom* $x \subseteq$ *Rep-udom* y

instance ⟨*proof*⟩
end

instance *udom* :: *po*
⟨*proof*⟩

instance *udom* :: *cpo*
⟨*proof*⟩

definition
udom-principal :: *nat* ⇒ *udom* **where**
udom-principal t = *Abs-udom* $\{u.$ *ubasis-le* $u\ t\}$

lemma *ubasis-countable*: $\exists f$::*ubasis* ⇒ *nat*. *inj* f

⟨*proof*⟩

interpretation *udom*:
ideal-completion ubasis-le udom-principal Rep-udom
⟨*proof*⟩

Universal domain is pointed

lemma *udom-minimal*: $udom\text{-}principal\ 0 \sqsubseteq x$
⟨*proof*⟩

instance *udom* :: *pcpo*
⟨*proof*⟩

lemma *inst-udom-pcpo*: $\bot = udom\text{-}principal\ 0$
⟨*proof*⟩

21.3 Compact bases of domains

typedef $'a\ compact\text{-}basis = \{x{::}'a{::}pcpo.\ compact\ x\}$
⟨*proof*⟩

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

lemma *Abs-compact-basis-inverse'* [*simp*]:
$compact\ x \Longrightarrow Rep\text{-}compact\text{-}basis\ (Abs\text{-}compact\text{-}basis\ x) = x$
⟨*proof*⟩

instantiation *compact-basis* :: (*pcpo*) *below*
begin

definition
compact-le-def:
$(\sqsubseteq) \equiv (\lambda x\ y.\ Rep\text{-}compact\text{-}basis\ x \sqsubseteq Rep\text{-}compact\text{-}basis\ y)$

instance ⟨*proof*⟩
end

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

definition
$approximants :: 'a \Rightarrow 'a\ compact\text{-}basis\ set$ **where**
$approximants = (\lambda x.\ \{a.\ Rep\text{-}compact\text{-}basis\ a \sqsubseteq x\})$

definition
$compact\text{-}bot :: 'a{::}pcpo\ compact\text{-}basis$ **where**
$compact\text{-}bot = Abs\text{-}compact\text{-}basis\ \bot$

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

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

21.4 Universality of *udom*

We use a locale to parameterize the construction over a chain of approx functions on the type to be embedded.

locale *bifinite-approx-chain* =
  approx-chain approx **for** *approx* :: *nat* ⇒ *'a::bifinite* → *'a*
begin

21.4.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
  fixes *A* :: *'a compact-basis set*
  shows ⟦*finite A*; *A* ≠ {}⟧ ⟹ ∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*
⟨*proof*⟩

definition
  choose :: *'a compact-basis set* ⇒ *'a compact-basis*
where
  choose A = (*SOME x*. *x* ∈ {*x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*})

lemma *choose-lemma*:
  ⟦*finite A*; *A* ≠ {}⟧ ⟹ *choose A* ∈ {*x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*}
⟨*proof*⟩

lemma *maximal-choose*:
  ⟦*finite A*; *y* ∈ *A*; *choose A* ⊑ *y*⟧ ⟹ *choose A* = *y*
⟨*proof*⟩

lemma *choose-in*: ⟦*finite A*; *A* ≠ {}⟧ ⟹ *choose A* ∈ *A*
⟨*proof*⟩

function
  choose-pos :: *'a compact-basis set* ⇒ *'a compact-basis* ⇒ *nat*
where
  choose-pos A x =
    (*if finite A* ∧ *x* ∈ *A* ∧ *x* ≠ *choose A*
      then Suc (*choose-pos* (*A* − {*choose A*}) *x*) *else 0*)
⟨*proof*⟩

termination *choose-pos*
⟨*proof*⟩

declare *choose-pos.simps* [*simp del*]

lemma *choose-pos-choose*: *finite* $A \Longrightarrow$ *choose-pos* A (*choose* A) $= 0$
⟨*proof*⟩

lemma *inj-on-choose-pos* [*OF refl*]:
⟦*card* $A = n$; *finite* A⟧ $\Longrightarrow$ *inj-on* (*choose-pos* A) A
⟨*proof*⟩

lemma *choose-pos-bounded* [*OF refl*]:
⟦*card* $A = n$; *finite* A; $x \in A$⟧ $\Longrightarrow$ *choose-pos* A $x < n$
⟨*proof*⟩

lemma *choose-pos-lessD*:
⟦*choose-pos* A $x <$ *choose-pos* A y; *finite* A; $x \in A$; $y \in A$⟧ $\Longrightarrow x \not\sqsubseteq y$
⟨*proof*⟩

21.4.2 Compact basis take function

primrec
cb-take :: *nat* $\Rightarrow$ *'a compact-basis* $\Rightarrow$ *'a compact-basis* **where**
cb-take 0 = (λx. *compact-bot*)
| *cb-take* (*Suc* n) = (λa. *Abs-compact-basis* (*approx* $n \cdot$(*Rep-compact-basis* a)))

declare *cb-take.simps* [*simp del*]

lemma *cb-take-zero* [*simp*]: *cb-take* 0 a = *compact-bot*
⟨*proof*⟩

lemma *Rep-cb-take*:
Rep-compact-basis (*cb-take* (*Suc* n) a) = *approx* $n \cdot$(*Rep-compact-basis* a)
⟨*proof*⟩

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

lemma *cb-take-covers*: $\exists n$. *cb-take* n $x = x$
⟨*proof*⟩

lemma *cb-take-less*: *cb-take* n $x \sqsubseteq x$
⟨*proof*⟩

lemma *cb-take-idem*: *cb-take* n (*cb-take* n x) = *cb-take* n x
⟨*proof*⟩

lemma *cb-take-mono*: $x \sqsubseteq y \Longrightarrow$ *cb-take* n $x \sqsubseteq$ *cb-take* n y
⟨*proof*⟩

lemma *cb-take-chain-le*: $m \leq n \Longrightarrow$ *cb-take* m $x \sqsubseteq$ *cb-take* n x
⟨*proof*⟩

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

21.4.3 Rank of basis elements

definition
  $rank$ :: $'a$ $compact\text{-}basis \Rightarrow nat$
where
  $rank\ x = (LEAST\ n.\ cb\text{-}take\ n\ x = x)$

lemma *compact-approx-rank*: $cb\text{-}take\ (rank\ x)\ x = x$
⟨*proof*⟩

lemma *rank-leD*: $rank\ x \le n \Longrightarrow cb\text{-}take\ n\ x = x$
⟨*proof*⟩

lemma *rank-leI*: $cb\text{-}take\ n\ x = x \Longrightarrow rank\ x \le n$
⟨*proof*⟩

lemma *rank-le-iff*: $rank\ x \le n \longleftrightarrow cb\text{-}take\ n\ x = x$
⟨*proof*⟩

lemma *rank-compact-bot* [*simp*]: $rank\ compact\text{-}bot = 0$
⟨*proof*⟩

lemma *rank-eq-0-iff* [*simp*]: $rank\ x = 0 \longleftrightarrow x = compact\text{-}bot$
⟨*proof*⟩

definition
  $rank\text{-}le$ :: $'a\ compact\text{-}basis \Rightarrow 'a\ compact\text{-}basis\ set$
where
  $rank\text{-}le\ x = \{y.\ rank\ y \le rank\ x\}$

definition
  $rank\text{-}lt$ :: $'a\ compact\text{-}basis \Rightarrow 'a\ compact\text{-}basis\ set$
where
  $rank\text{-}lt\ x = \{y.\ rank\ y < rank\ x\}$

definition
  $rank\text{-}eq$ :: $'a\ compact\text{-}basis \Rightarrow 'a\ compact\text{-}basis\ set$
where
  $rank\text{-}eq\ x = \{y.\ rank\ y = rank\ x\}$

lemma *rank-eq-cong*: $rank\ x = rank\ y \Longrightarrow rank\text{-}eq\ x = rank\text{-}eq\ y$
⟨*proof*⟩

lemma *rank-lt-cong*: $rank\ x = rank\ y \Longrightarrow rank\text{-}lt\ x = rank\text{-}lt\ y$
⟨*proof*⟩

lemma *rank-eq-subset*: $rank\text{-}eq\ x \subseteq rank\text{-}le\ x$
⟨*proof*⟩

lemma *rank-lt-subset*: $rank\text{-}lt\ x \subseteq rank\text{-}le\ x$
⟨*proof*⟩

lemma *finite-rank-le*: $finite\ (rank\text{-}le\ x)$
⟨*proof*⟩

lemma *finite-rank-eq*: $finite\ (rank\text{-}eq\ x)$
⟨*proof*⟩

lemma *finite-rank-lt*: $finite\ (rank\text{-}lt\ x)$
⟨*proof*⟩

lemma *rank-lt-Int-rank-eq*: $rank\text{-}lt\ x \cap rank\text{-}eq\ x = \{\}$
⟨*proof*⟩

lemma *rank-lt-Un-rank-eq*: $rank\text{-}lt\ x \cup rank\text{-}eq\ x = rank\text{-}le\ x$
⟨*proof*⟩

21.4.4 Sequencing basis elements

definition
  $place :: 'a\ compact\text{-}basis \Rightarrow nat$
where
  $place\ x = card\ (rank\text{-}lt\ x) + choose\text{-}pos\ (rank\text{-}eq\ x)\ x$

lemma *place-bounded*: $place\ x < card\ (rank\text{-}le\ x)$
⟨*proof*⟩

lemma *place-ge*: $card\ (rank\text{-}lt\ x) \leq place\ x$
⟨*proof*⟩

lemma *place-rank-mono*:
  fixes $x\ y :: 'a\ compact\text{-}basis$
  shows $rank\ x < rank\ y \Longrightarrow place\ x < place\ y$
⟨*proof*⟩

lemma *place-eqD*: $place\ x = place\ y \Longrightarrow x = y$
⟨*proof*⟩

lemma *inj-place*: $inj\ place$
⟨*proof*⟩

21.4.5 Embedding and projection on basis elements

definition
  $sub :: 'a\ compact\text{-}basis \Rightarrow 'a\ compact\text{-}basis$
where

$sub\ x = (case\ rank\ x\ of\ 0 \Rightarrow compact\text{-}bot \mid Suc\ k \Rightarrow cb\text{-}take\ k\ x)$

lemma *rank-sub-less*: $x \neq compact\text{-}bot \Longrightarrow rank\ (sub\ x) < rank\ x$
⟨*proof*⟩

lemma *place-sub-less*: $x \neq compact\text{-}bot \Longrightarrow place\ (sub\ x) < place\ x$
⟨*proof*⟩

lemma *sub-below*: $sub\ x \sqsubseteq x$
⟨*proof*⟩

lemma *rank-less-imp-below-sub*: $[\![x \sqsubseteq y;\ rank\ x < rank\ y]\!] \Longrightarrow x \sqsubseteq sub\ y$
⟨*proof*⟩

function *basis-emb* :: $'a\ compact\text{-}basis \Rightarrow ubasis$
where $basis\text{-}emb\ x = (if\ x = compact\text{-}bot\ then\ 0\ else$
$node\ (place\ x)\ (basis\text{-}emb\ (sub\ x))$
$(basis\text{-}emb\ `\ \{y.\ place\ y < place\ x \wedge x \sqsubseteq y\}))$
⟨*proof*⟩

termination *basis-emb*
⟨*proof*⟩

declare *basis-emb.simps* [*simp del*]

lemma *basis-emb-compact-bot* [*simp*]:
$basis\text{-}emb\ compact\text{-}bot = 0$
⟨*proof*⟩

lemma *basis-emb-rec*:
$basis\text{-}emb\ x = node\ (place\ x)\ (basis\text{-}emb\ (sub\ x))\ (basis\text{-}emb\ `\ \{y.\ place\ y < place\ x \wedge x \sqsubseteq y\})$
if $x \neq compact\text{-}bot$
⟨*proof*⟩

lemma *basis-emb-eq-0-iff* [*simp*]:
$basis\text{-}emb\ x = 0 \longleftrightarrow x = compact\text{-}bot$
⟨*proof*⟩

lemma *fin1*: $finite\ \{y.\ place\ y < place\ x \wedge x \sqsubseteq y\}$
⟨*proof*⟩

lemma *fin2*: $finite\ (basis\text{-}emb\ `\ \{y.\ place\ y < place\ x \wedge x \sqsubseteq y\})$
⟨*proof*⟩

lemma *rank-place-mono*:
$[\![place\ x < place\ y;\ x \sqsubseteq y]\!] \Longrightarrow rank\ x < rank\ y$
⟨*proof*⟩

lemma *basis-emb-mono*:
$x \sqsubseteq y \Longrightarrow \mathit{ubasis\text{-}le}\ (\mathit{basis\text{-}emb}\ x)\ (\mathit{basis\text{-}emb}\ y)$
⟨*proof*⟩

lemma *inj-basis-emb*: *inj basis-emb*
⟨*proof*⟩

definition
$\mathit{basis\text{-}prj} :: \mathit{ubasis} \Rightarrow {'a}\ \mathit{compact\text{-}basis}$
where
$\mathit{basis\text{-}prj}\ x = \mathit{inv}\ \mathit{basis\text{-}emb}$
$(\mathit{ubasis\text{-}until}\ (\lambda x.\ x \in \mathit{range}\ (\mathit{basis\text{-}emb} :: {'a}\ \mathit{compact\text{-}basis} \Rightarrow \mathit{ubasis}))\ x)$

lemma *basis-prj-basis-emb*: $\bigwedge x.\ \mathit{basis\text{-}prj}\ (\mathit{basis\text{-}emb}\ x) = x$
⟨*proof*⟩

lemma *basis-prj-node*:
$[\![\mathit{finite}\ S;\ \mathit{node}\ i\ a\ S \notin \mathit{range}\ (\mathit{basis\text{-}emb} :: {'a}\ \mathit{compact\text{-}basis} \Rightarrow \mathit{nat})]\!]$
$\Longrightarrow \mathit{basis\text{-}prj}\ (\mathit{node}\ i\ a\ S) = (\mathit{basis\text{-}prj}\ a :: {'a}\ \mathit{compact\text{-}basis})$
⟨*proof*⟩

lemma *basis-prj-0*: $\mathit{basis\text{-}prj}\ 0 = \mathit{compact\text{-}bot}$
⟨*proof*⟩

lemma *node-eq-basis-emb-iff*:
$\mathit{finite}\ S \Longrightarrow \mathit{node}\ i\ a\ S = \mathit{basis\text{-}emb}\ x \longleftrightarrow$
$x \neq \mathit{compact\text{-}bot} \wedge i = \mathit{place}\ x \wedge a = \mathit{basis\text{-}emb}\ (\mathit{sub}\ x) \wedge$
$S = \mathit{basis\text{-}emb}\ `\ \{y.\ \mathit{place}\ y < \mathit{place}\ x \wedge x \sqsubseteq y\}$
⟨*proof*⟩

lemma *basis-prj-mono*: $\mathit{ubasis\text{-}le}\ a\ b \Longrightarrow \mathit{basis\text{-}prj}\ a \sqsubseteq \mathit{basis\text{-}prj}\ b$
⟨*proof*⟩

lemma *basis-emb-prj-less*: $\mathit{ubasis\text{-}le}\ (\mathit{basis\text{-}emb}\ (\mathit{basis\text{-}prj}\ x))\ x$
⟨*proof*⟩

lemma *ideal-completion*:
$\mathit{ideal\text{-}completion}\ \mathit{below}\ \mathit{Rep\text{-}compact\text{-}basis}\ (\mathit{approximants} :: {'a} \Rightarrow \text{-})$
⟨*proof*⟩

end

interpretation *compact-basis*:
$\mathit{ideal\text{-}completion}\ \mathit{below}\ \mathit{Rep\text{-}compact\text{-}basis}$
$\mathit{approximants} :: {'a}::\mathit{bifinite} \Rightarrow {'a}\ \mathit{compact\text{-}basis}\ \mathit{set}$
⟨*proof*⟩

21.4.6 EP-pair from any bifinite domain into *udom*

context *bifinite-approx-chain* **begin**

definition
$udom\text{-}emb :: {'a} \rightarrow udom$
where
$udom\text{-}emb = compact\text{-}basis.extension\ (\lambda x.\ udom\text{-}principal\ (basis\text{-}emb\ x))$

definition
$udom\text{-}prj :: udom \rightarrow {'a}$
where
$udom\text{-}prj = udom.extension\ (\lambda x.\ Rep\text{-}compact\text{-}basis\ (basis\text{-}prj\ x))$

lemma *udom-emb-principal*:
$udom\text{-}emb\cdot(Rep\text{-}compact\text{-}basis\ x) = udom\text{-}principal\ (basis\text{-}emb\ x)$
$\langle proof \rangle$

lemma *udom-prj-principal*:
$udom\text{-}prj\cdot(udom\text{-}principal\ x) = Rep\text{-}compact\text{-}basis\ (basis\text{-}prj\ x)$
$\langle proof \rangle$

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
$\langle proof \rangle$

end

abbreviation $udom\text{-}emb \equiv bifinite\text{-}approx\text{-}chain.udom\text{-}emb$
abbreviation $udom\text{-}prj \equiv bifinite\text{-}approx\text{-}chain.udom\text{-}prj$

lemmas *ep-pair-udom* =
bifinite-approx-chain.ep-pair-udom [*unfolded bifinite-approx-chain-def*]

21.5 Chain of approx functions for type *udom*

definition
$udom\text{-}approx :: nat \Rightarrow udom \rightarrow udom$
where
$udom\text{-}approx\ i =$
$udom.extension\ (\lambda x.\ udom\text{-}principal\ (ubasis\text{-}until\ (\lambda y.\ y \leq i)\ x))$

lemma *udom-approx-mono*:
$ubasis\text{-}le\ a\ b \Longrightarrow$
$udom\text{-}principal\ (ubasis\text{-}until\ (\lambda y.\ y \leq i)\ a) \sqsubseteq$
$udom\text{-}principal\ (ubasis\text{-}until\ (\lambda y.\ y \leq i)\ b)$
$\langle proof \rangle$

lemma *adm-mem-finite*: $[\![cont\ f;\ finite\ S]\!] \Longrightarrow adm\ (\lambda x.\ f\ x \in S)$
$\langle proof \rangle$

lemma *udom-approx-principal*:
  $udom\text{-}approx\ i \cdot (udom\text{-}principal\ x) =$
    $udom\text{-}principal\ (ubasis\text{-}until\ (\lambda y.\ y \leq i)\ x)$
$\langle proof \rangle$

lemma *finite-deflation-udom-approx*: $finite\text{-}deflation\ (udom\text{-}approx\ i)$
$\langle proof \rangle$

interpretation *udom-approx*: $finite\text{-}deflation\ udom\text{-}approx\ i$
$\langle proof \rangle$

lemma *chain-udom-approx* [*simp*]: $chain\ (\lambda i.\ udom\text{-}approx\ i)$
$\langle proof \rangle$

lemma *lub-udom-approx* [*simp*]: $(\bigsqcup i.\ udom\text{-}approx\ i) = ID$
$\langle proof \rangle$

lemma *udom-approx* [*simp*]: $approx\text{-}chain\ udom\text{-}approx$
$\langle proof \rangle$

instance *udom* :: *bifinite*
  $\langle proof \rangle$

hide-const (**open**) *node*

notation *binomial* (**infixl** *choose 65*)

end

22 Algebraic deflations

theory *Algebraic*
imports *Universal Map-Functions*
begin

default-sort *bifinite*

22.1 Type constructor for finite deflations

typedef $'a\ fin\text{-}defl = \{d{::}'a \to 'a.\ finite\text{-}deflation\ d\}$
$\langle proof \rangle$

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

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

instance $\langle proof \rangle$

end

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

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

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

lemma *Rep-fin-defl-mono*: $a \sqsubseteq b \Longrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ a \sqsubseteq \mathit{Rep\text{-}fin\text{-}defl}\ b$
⟨*proof*⟩

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

lemma (**in** *finite-deflation*) *compact-belowI*:
assumes $\bigwedge x.\ \mathit{compact}\ x \Longrightarrow d{\cdot}x = x \Longrightarrow f{\cdot}x = x$ **shows** $d \sqsubseteq f$
⟨*proof*⟩

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

22.2 Defining algebraic deflations by ideal completion

typedef $'a\ \mathit{defl} = \{S{::}'a\ \mathit{fin\text{-}defl}\ \mathit{set}.\ \mathit{below.ideal}\ S\}$
⟨*proof*⟩

instantiation *defl* :: (*bifinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow \mathit{Rep\text{-}defl}\ x \subseteq \mathit{Rep\text{-}defl}\ y$

instance ⟨*proof*⟩
end

instance *defl* :: (*bifinite*) *po*
⟨*proof*⟩

instance *defl* :: (*bifinite*) *cpo*
⟨*proof*⟩

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

lemma *fin-defl-countable*: $\exists f{::}'a\ \mathit{fin\text{-}defl} \Rightarrow \mathit{nat}.\ \mathit{inj}\ f$
⟨*proof*⟩

interpretation *defl*: *ideal-completion below defl-principal Rep-defl*
⟨*proof*⟩

Algebraic deflations are pointed

lemma *defl-minimal*: $\mathit{defl\text{-}principal}\ (\mathit{Abs\text{-}fin\text{-}defl}\ \bot) \sqsubseteq x$
⟨*proof*⟩

instance *defl* :: (*bifinite*) *pcpo*
⟨*proof*⟩

lemma *inst-defl-pcpo*: $\bot = \mathit{defl\text{-}principal}\ (\mathit{Abs\text{-}fin\text{-}defl}\ \bot)$
⟨*proof*⟩

22.3 Applying algebraic deflations

definition
$\mathit{cast} :: 'a\ \mathit{defl} \rightarrow 'a \rightarrow 'a$
where
$\mathit{cast} = \mathit{defl.extension}\ \mathit{Rep\text{-}fin\text{-}defl}$

lemma *cast-defl-principal*:
$\mathit{cast}\cdot(\mathit{defl\text{-}principal}\ a) = \mathit{Rep\text{-}fin\text{-}defl}\ a$
⟨*proof*⟩

lemma *deflation-cast*: $\mathit{deflation}\ (\mathit{cast}\cdot d)$
⟨*proof*⟩

lemma *finite-deflation-cast*:
$\mathit{compact}\ d \Longrightarrow \mathit{finite\text{-}deflation}\ (\mathit{cast}\cdot d)$
⟨*proof*⟩

interpretation *cast*: *deflation cast$\cdot$d*
$\langle$*proof*$\rangle$

declare *cast.idem* [*simp*]

lemma *compact-cast* [*simp*]: *compact d* $\Longrightarrow$ *compact* (*cast$\cdot$d*)
$\langle$*proof*$\rangle$

lemma *cast-below-cast*: *cast$\cdot$A* $\sqsubseteq$ *cast$\cdot$B* $\longleftrightarrow$ *A* $\sqsubseteq$ *B*
$\langle$*proof*$\rangle$

lemma *compact-cast-iff*: *compact* (*cast$\cdot$d*) $\longleftrightarrow$ *compact d*
$\langle$*proof*$\rangle$

lemma *cast-below-imp-below*: *cast$\cdot$A* $\sqsubseteq$ *cast$\cdot$B* $\Longrightarrow$ *A* $\sqsubseteq$ *B*
$\langle$*proof*$\rangle$

lemma *cast-eq-imp-eq*: *cast$\cdot$A* = *cast$\cdot$B* $\Longrightarrow$ *A* = *B*
$\langle$*proof*$\rangle$

lemma *cast-strict1* [*simp*]: *cast*$\cdot\bot = \bot$
$\langle$*proof*$\rangle$

lemma *cast-strict2* [*simp*]: *cast$\cdot$A*$\cdot\bot = \bot$
$\langle$*proof*$\rangle$

22.4 Deflation combinators

definition
  defl-fun1 e p f =
    defl.extension (λ*a*.
      defl-principal (*Abs-fin-defl*
        (*e oo f*$\cdot$(*Rep-fin-defl a*) *oo p*)))

definition
  defl-fun2 e p f =
    defl.extension (λ*a*.
      defl.extension (λ*b*.
        defl-principal (*Abs-fin-defl*
          (*e oo f*$\cdot$(*Rep-fin-defl a*)$\cdot$(*Rep-fin-defl b*) *oo p*))))

lemma *cast-defl-fun1*:
  assumes *ep*: *ep-pair e p*
  assumes *f*: $\bigwedge$*a*. *finite-deflation a* $\Longrightarrow$ *finite-deflation* (*f$\cdot$a*)
  shows *cast*$\cdot$(*defl-fun1 e p f$\cdot$A*) = *e oo f*$\cdot$(*cast$\cdot$A*) *oo p*
$\langle$*proof*$\rangle$

lemma *cast-defl-fun2*:

assumes *ep*: *ep-pair e p*
assumes *f*: $\bigwedge a\ b.$ *finite-deflation a* $\Longrightarrow$ *finite-deflation b* $\Longrightarrow$ *finite-deflation* $(f\cdot a\cdot b)$
shows *cast*$\cdot$(*defl-fun2 e p f*$\cdot A\cdot B$) = *e oo f*$\cdot$(*cast*$\cdot A$)$\cdot$(*cast*$\cdot B$) *oo p*
⟨*proof*⟩

end

23 Representable domains

theory *Representable*
imports *Algebraic Map-Functions HOL−Library.Countable*
begin

default-sort *cpo*

23.1 Class of representable domains

We define a "domain" as a pcpo that is isomorphic to some algebraic deflation over the universal domain; this is equivalent to being omega-bifinite.

A predomain is a cpo that, when lifted, becomes a domain. Predomains are represented by deflations over a lifted universal domain type.

class *predomain-syn* = *cpo* +
fixes *liftemb* :: $'a_\bot \to udom_\bot$
fixes *liftprj* :: $udom_\bot \to 'a_\bot$
fixes *liftdefl* :: *'a itself* ⇒ *udom u defl*

class *predomain* = *predomain-syn* +
assumes *predomain-ep*: *ep-pair liftemb liftprj*
assumes *cast-liftdefl*: *cast*$\cdot$(*liftdefl TYPE('a)*) = *liftemb oo liftprj*

syntax *-LIFTDEFL* :: *type* ⇒ *logic* ((*1LIFTDEFL*/(*1*'(-'))))
translations *LIFTDEFL('t)* ⇌ *CONST liftdefl TYPE('t)*

definition *liftdefl-of* :: *udom defl* → *udom u defl*
where *liftdefl-of* = *defl-fun1 ID ID u-map*

lemma *cast-liftdefl-of*: *cast*$\cdot$(*liftdefl-of*$\cdot t$) = *u-map*$\cdot$(*cast*$\cdot t$)
⟨*proof*⟩

class *domain* = *predomain-syn* + *pcpo* +
fixes *emb* :: *'a* → *udom*
fixes *prj* :: *udom* → *'a*
fixes *defl* :: *'a itself* ⇒ *udom defl*
assumes *ep-pair-emb-prj*: *ep-pair emb prj*
assumes *cast-DEFL*: *cast*$\cdot$(*defl TYPE('a)*) = *emb oo prj*
assumes *liftemb-eq*: *liftemb* = *u-map*$\cdot$*emb*
assumes *liftprj-eq*: *liftprj* = *u-map*$\cdot$*prj*

assumes *liftdefl-eq*: *liftdefl TYPE('a)* = *liftdefl-of*·(*defl TYPE('a)*)

syntax *-DEFL* :: *type* ⇒ *logic* ((*1DEFL*/(*1'*(-'))))
translations *DEFL('t)* ⇌ *CONST defl TYPE('t)*

instance *domain* ⊆ *predomain*
⟨*proof*⟩

Constants *liftemb* and *liftprj* imply class predomain.

⟨*ML*⟩

interpretation *predomain*: *pcpo-ep-pair liftemb liftprj*
⟨*proof*⟩

interpretation *domain*: *pcpo-ep-pair emb prj*
⟨*proof*⟩

lemmas *emb-inverse* = *domain.e-inverse*
lemmas *emb-prj-below* = *domain.e-p-below*
lemmas *emb-eq-iff* = *domain.e-eq-iff*
lemmas *emb-strict* = *domain.e-strict*
lemmas *prj-strict* = *domain.p-strict*

23.2 Domains are bifinite

lemma *approx-chain-ep-cast*:
assumes *ep*: *ep-pair* (*e*::'*a*::*pcpo* → '*b*::*bifinite*) (*p*::'*b* → '*a*)
assumes *cast-t*: *cast*·*t* = *e oo p*
shows ∃(*a*::*nat* ⇒ '*a*::*pcpo* → '*a*). *approx-chain a*
⟨*proof*⟩

instance *domain* ⊆ *bifinite*
⟨*proof*⟩

instance *predomain* ⊆ *profinite*
⟨*proof*⟩

23.3 Universal domain ep-pairs

definition *u-emb* = *udom-emb* (λ*i*. *u-map*·(*udom-approx i*))
definition *u-prj* = *udom-prj* (λ*i*. *u-map*·(*udom-approx i*))

definition *prod-emb* = *udom-emb* (λ*i*. *prod-map*·(*udom-approx i*)·(*udom-approx i*))
definition *prod-prj* = *udom-prj* (λ*i*. *prod-map*·(*udom-approx i*)·(*udom-approx i*))

definition *sprod-emb* = *udom-emb* (λ*i*. *sprod-map*·(*udom-approx i*)·(*udom-approx i*))
definition *sprod-prj* = *udom-prj* (λ*i*. *sprod-map*·(*udom-approx i*)·(*udom-approx i*))

definition *ssum-emb* = *udom-emb* ($\lambda i.$ *ssum-map*$\cdot$(*udom-approx* i)$\cdot$(*udom-approx* i))
definition *ssum-prj* = *udom-prj* ($\lambda i.$ *ssum-map*$\cdot$(*udom-approx* i)$\cdot$(*udom-approx* i))

definition *sfun-emb* = *udom-emb* ($\lambda i.$ *sfun-map*$\cdot$(*udom-approx* i)$\cdot$(*udom-approx* i))
definition *sfun-prj* = *udom-prj* ($\lambda i.$ *sfun-map*$\cdot$(*udom-approx* i)$\cdot$(*udom-approx* i))

lemma *ep-pair-u*: *ep-pair u-emb u-prj*
⟨*proof*⟩

lemma *ep-pair-prod*: *ep-pair prod-emb prod-prj*
⟨*proof*⟩

lemma *ep-pair-sprod*: *ep-pair sprod-emb sprod-prj*
⟨*proof*⟩

lemma *ep-pair-ssum*: *ep-pair ssum-emb ssum-prj*
⟨*proof*⟩

lemma *ep-pair-sfun*: *ep-pair sfun-emb sfun-prj*
⟨*proof*⟩

23.4 Type combinators

definition *u-defl* :: *udom defl* → *udom defl*
where *u-defl* = *defl-fun1 u-emb u-prj u-map*

definition *prod-defl* :: *udom defl* → *udom defl* → *udom defl*
where *prod-defl* = *defl-fun2 prod-emb prod-prj prod-map*

definition *sprod-defl* :: *udom defl* → *udom defl* → *udom defl*
where *sprod-defl* = *defl-fun2 sprod-emb sprod-prj sprod-map*

definition *ssum-defl* :: *udom defl* → *udom defl* → *udom defl*
where *ssum-defl* = *defl-fun2 ssum-emb ssum-prj ssum-map*

definition *sfun-defl* :: *udom defl* → *udom defl* → *udom defl*
where *sfun-defl* = *defl-fun2 sfun-emb sfun-prj sfun-map*

lemma *cast-u-defl*:
cast$\cdot$(*u-defl*$\cdot$$A$) = *u-emb oo u-map*$\cdot$(*cast*$\cdot$$A$) *oo u-prj*
⟨*proof*⟩

lemma *cast-prod-defl*:
cast$\cdot$(*prod-defl*$\cdot$$A$$\cdot$$B$) =
prod-emb oo prod-map$\cdot$(*cast*$\cdot$$A$)$\cdot$(*cast*$\cdot$$B$) *oo prod-prj*
⟨*proof*⟩

lemma *cast-sprod-defl*:
  $cast \cdot (sprod\text{-}defl \cdot A \cdot B) =$
    $sprod\text{-}emb\ oo\ sprod\text{-}map \cdot (cast \cdot A) \cdot (cast \cdot B)\ oo\ sprod\text{-}prj$
⟨*proof*⟩

lemma *cast-ssum-defl*:
  $cast \cdot (ssum\text{-}defl \cdot A \cdot B) =$
    $ssum\text{-}emb\ oo\ ssum\text{-}map \cdot (cast \cdot A) \cdot (cast \cdot B)\ oo\ ssum\text{-}prj$
⟨*proof*⟩

lemma *cast-sfun-defl*:
  $cast \cdot (sfun\text{-}defl \cdot A \cdot B) =$
    $sfun\text{-}emb\ oo\ sfun\text{-}map \cdot (cast \cdot A) \cdot (cast \cdot B)\ oo\ sfun\text{-}prj$
⟨*proof*⟩

Special deflation combinator for unpointed types.

definition $u\text{-}liftdefl :: udom\ u\ defl \rightarrow udom\ defl$
  where $u\text{-}liftdefl = defl\text{-}fun1\ u\text{-}emb\ u\text{-}prj\ ID$

lemma *cast-u-liftdefl*:
  $cast \cdot (u\text{-}liftdefl \cdot A) = u\text{-}emb\ oo\ cast \cdot A\ oo\ u\text{-}prj$
⟨*proof*⟩

lemma *u-liftdefl-liftdefl-of*:
  $u\text{-}liftdefl \cdot (liftdefl\text{-}of \cdot A) = u\text{-}defl \cdot A$
⟨*proof*⟩

23.5 Class instance proofs

23.5.1 Universal domain

instantiation $udom :: domain$
begin

definition [*simp*]:
  $emb = (ID :: udom \rightarrow udom)$

definition [*simp*]:
  $prj = (ID :: udom \rightarrow udom)$

definition
  $defl\ (t::udom\ itself) = (\bigsqcup i.\ defl\text{-}principal\ (Abs\text{-}fin\text{-}defl\ (udom\text{-}approx\ i)))$

definition
  $(liftemb :: udom\ u \rightarrow udom\ u) = u\text{-}map \cdot emb$

definition
  $(liftprj :: udom\ u \rightarrow udom\ u) = u\text{-}map \cdot prj$

definition

liftdefl $(t::udom\ itself) = liftdefl\text{-}of \cdot DEFL(udom)$

instance ⟨*proof*⟩

end

23.5.2 Lifted cpo

instantiation $u :: (predomain)\ domain$
begin

definition
$emb = u\text{-}emb\ oo\ liftemb$

definition
$prj = liftprj\ oo\ u\text{-}prj$

definition
$defl\ (t::'a\ u\ itself) = u\text{-}liftdefl \cdot LIFTDEFL('a)$

definition
$(liftemb :: 'a\ u\ u \to udom\ u) = u\text{-}map \cdot emb$

definition
$(liftprj :: udom\ u \to 'a\ u\ u) = u\text{-}map \cdot prj$

definition
$liftdefl\ (t::'a\ u\ itself) = liftdefl\text{-}of \cdot DEFL('a\ u)$

instance ⟨*proof*⟩

end

lemma *DEFL-u*: $DEFL('a::predomain\ u) = u\text{-}liftdefl \cdot LIFTDEFL('a)$
⟨*proof*⟩

23.5.3 Strict function space

instantiation $sfun :: (domain, domain)\ domain$
begin

definition
$emb = sfun\text{-}emb\ oo\ sfun\text{-}map \cdot prj \cdot emb$

definition
$prj = sfun\text{-}map \cdot emb \cdot prj\ oo\ sfun\text{-}prj$

definition
$defl\ (t::('a \to!\ 'b)\ itself) = sfun\text{-}defl \cdot DEFL('a) \cdot DEFL('b)$

definition
  $(liftemb :: ('a \rightarrow! 'b)\ u \rightarrow udom\ u) = u\text{-}map\cdot emb$

definition
  $(liftprj :: udom\ u \rightarrow ('a \rightarrow! 'b)\ u) = u\text{-}map\cdot prj$

definition
  $liftdefl\ (t::('a \rightarrow! 'b)\ itself) = liftdefl\text{-}of\cdot DEFL('a \rightarrow! 'b)$

instance ⟨*proof*⟩

end

lemma *DEFL-sfun*:
  $DEFL('a::domain \rightarrow! 'b::domain) = sfun\text{-}defl\cdot DEFL('a)\cdot DEFL('b)$
⟨*proof*⟩

23.5.4 Continuous function space

instantiation *cfun* :: (*predomain*, *domain*) *domain*
begin

definition
  $emb = emb\ oo\ encode\text{-}cfun$

definition
  $prj = decode\text{-}cfun\ oo\ prj$

definition
  $defl\ (t::('a \rightarrow 'b)\ itself) = DEFL('a\ u \rightarrow! 'b)$

definition
  $(liftemb :: ('a \rightarrow 'b)\ u \rightarrow udom\ u) = u\text{-}map\cdot emb$

definition
  $(liftprj :: udom\ u \rightarrow ('a \rightarrow 'b)\ u) = u\text{-}map\cdot prj$

definition
  $liftdefl\ (t::('a \rightarrow 'b)\ itself) = liftdefl\text{-}of\cdot DEFL('a \rightarrow 'b)$

instance ⟨*proof*⟩

end

lemma *DEFL-cfun*:
  $DEFL('a::predomain \rightarrow 'b::domain) = DEFL('a\ u \rightarrow! 'b)$
⟨*proof*⟩

23.5.5 Strict product

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

definition
emb = *sprod-emb oo sprod-map·emb·emb*

definition
prj = *sprod-map·prj·prj oo sprod-prj*

definition
defl (*t*::(′*a* ⊗ ′*b*) *itself*) = *sprod-defl·DEFL*(′*a*)·*DEFL*(′*b*)

definition
(*liftemb* :: (′*a* ⊗ ′*b*) *u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → (′*a* ⊗ ′*b*) *u*) = *u-map·prj*

definition
liftdefl (*t*::(′*a* ⊗ ′*b*) *itself*) = *liftdefl-of·DEFL*(′*a* ⊗ ′*b*)

instance ⟨*proof*⟩

end

lemma *DEFL-sprod*:
DEFL(′*a*::*domain* ⊗ ′*b*::*domain*) = *sprod-defl·DEFL*(′*a*)·*DEFL*(′*b*)
⟨*proof*⟩

23.5.6 Cartesian product

definition *prod-liftdefl* :: *udom u defl* → *udom u defl* → *udom u defl*
where *prod-liftdefl* = *defl-fun2* (*u-map·prod-emb oo decode-prod-u*)
(*encode-prod-u oo u-map·prod-prj*) *sprod-map*

lemma *cast-prod-liftdefl*:
cast·(*prod-liftdefl·a·b*) =
(*u-map·prod-emb oo decode-prod-u*) *oo sprod-map*·(*cast·a*)·(*cast·b*) *oo*
(*encode-prod-u oo u-map·prod-prj*)
⟨*proof*⟩

instantiation *prod* :: (*predomain*, *predomain*) *predomain*
begin

definition
liftemb = (*u-map·prod-emb oo decode-prod-u*) *oo*
(*sprod-map·liftemb·liftemb oo encode-prod-u*)

definition
$liftprj = (decode\text{-}prod\text{-}u\ oo\ sprod\text{-}map{\cdot}liftprj{\cdot}liftprj)\ oo$
$(encode\text{-}prod\text{-}u\ oo\ u\text{-}map{\cdot}prod\text{-}prj)$

definition
$liftdefl\ (t::('a \times 'b)\ itself) = prod\text{-}liftdefl{\cdot}LIFTDEFL('a){\cdot}LIFTDEFL('b)$

instance ⟨*proof*⟩

end

instantiation *prod* :: (*domain*, *domain*) *domain*
begin

definition
$emb = prod\text{-}emb\ oo\ prod\text{-}map{\cdot}emb{\cdot}emb$

definition
$prj = prod\text{-}map{\cdot}prj{\cdot}prj\ oo\ prod\text{-}prj$

definition
$defl\ (t::('a \times 'b)\ itself) = prod\text{-}defl{\cdot}DEFL('a){\cdot}DEFL('b)$

instance ⟨*proof*⟩

end

lemma *DEFL-prod*:
$DEFL('a::domain \times 'b::domain) = prod\text{-}defl{\cdot}DEFL('a){\cdot}DEFL('b)$
⟨*proof*⟩

lemma *LIFTDEFL-prod*:
$LIFTDEFL('a::predomain \times 'b::predomain) =$
$prod\text{-}liftdefl{\cdot}LIFTDEFL('a){\cdot}LIFTDEFL('b)$
⟨*proof*⟩

23.5.7 Unit type

instantiation *unit* :: *domain*
begin

definition
$emb = (\bot :: unit \to udom)$

definition
$prj = (\bot :: udom \to unit)$

definition
$defl\ (t::unit\ itself) = \bot$

definition
($liftemb$ :: $unit\ u \rightarrow udom\ u$) = $u\text{-}map\cdot emb$

definition
($liftprj$ :: $udom\ u \rightarrow unit\ u$) = $u\text{-}map\cdot prj$

definition
$liftdefl\ (t::unit\ itself) = liftdefl\text{-}of\cdot DEFL(unit)$

instance ⟨*proof*⟩

end

23.5.8 Discrete cpo

instantiation $discr$ :: ($countable$) $predomain$
begin

definition
$(liftemb :: 'a\ discr\ u \rightarrow udom\ u) = strictify\cdot up\ oo\ udom\text{-}emb\ discr\text{-}approx$

definition
$(liftprj :: udom\ u \rightarrow 'a\ discr\ u) = udom\text{-}prj\ discr\text{-}approx\ oo\ fup\cdot ID$

definition
$liftdefl\ (t::'a\ discr\ itself) =$
$(\bigsqcup i.\ defl\text{-}principal\ (Abs\text{-}fin\text{-}defl\ (liftemb\ oo\ discr\text{-}approx\ i\ oo\ (liftprj::udom\ u \rightarrow 'a\ discr\ u))))$

instance ⟨*proof*⟩

end

23.5.9 Strict sum

instantiation $ssum$ :: ($domain$, $domain$) $domain$
begin

definition
$emb = ssum\text{-}emb\ oo\ ssum\text{-}map\cdot emb\cdot emb$

definition
$prj = ssum\text{-}map\cdot prj\cdot prj\ oo\ ssum\text{-}prj$

definition
$defl\ (t::('a \oplus 'b)\ itself) = ssum\text{-}defl\cdot DEFL('a)\cdot DEFL('b)$

definition
$(liftemb :: ('a \oplus 'b)\ u \rightarrow udom\ u) = u\text{-}map\cdot emb$

definition
$(liftprj :: udom\ u \to ('a \oplus 'b)\ u) = u\text{-}map \cdot prj$

definition
$liftdefl\ (t::('a \oplus 'b)\ itself) = liftdefl\text{-}of \cdot DEFL('a \oplus 'b)$

instance ⟨*proof*⟩

end

lemma *DEFL-ssum*:
$DEFL('a::domain \oplus 'b::domain) = ssum\text{-}defl \cdot DEFL('a) \cdot DEFL('b)$
⟨*proof*⟩

23.5.10 Lifted HOL type

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

definition
$emb = emb\ oo\ (\Lambda\ x.\ Rep\text{-}lift\ x)$

definition
$prj = (\Lambda\ y.\ Abs\text{-}lift\ y)\ oo\ prj$

definition
$defl\ (t::'a\ lift\ itself) = DEFL('a\ discr\ u)$

definition
$(liftemb :: 'a\ lift\ u \to udom\ u) = u\text{-}map \cdot emb$

definition
$(liftprj :: udom\ u \to 'a\ lift\ u) = u\text{-}map \cdot prj$

definition
$liftdefl\ (t::'a\ lift\ itself) = liftdefl\text{-}of \cdot DEFL('a\ lift)$

instance ⟨*proof*⟩

end

end

24 The unit domain

theory *One*
imports *Lift*
begin

type-synonym *one* = *unit lift*

translations
($type$) *one* $\leftharpoondown$ ($type$) *unit lift*

definition *ONE* :: *one*
where $ONE \equiv Def$ ()

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*]: $P\ \bot \Longrightarrow P\ ONE \Longrightarrow P\ x$
⟨*proof*⟩

lemma *dist-below-one* [*simp*]: $ONE \not\sqsubseteq \bot$
⟨*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-case* :: $'a::pcpo \rightarrow one \rightarrow 'a$
where $one\text{-}case = (\Lambda\ a\ x.\ seq{\cdot}x{\cdot}a)$

translations
$case\ x\ of\ XCONST\ ONE \Rightarrow t \rightleftharpoons CONST\ one\text{-}case{\cdot}t{\cdot}x$
$case\ x\ of\ XCONST\ ONE :: 'a \Rightarrow t \rightharpoonup CONST\ one\text{-}case{\cdot}t{\cdot}x$
$\Lambda\ (XCONST\ ONE).\ t \rightleftharpoons CONST\ one\text{-}case{\cdot}t$

lemma *one-case1* [*simp*]: (*case* $\bot$ *of ONE* $\Rightarrow$ *t*) = $\bot$
  ⟨*proof*⟩

lemma *one-case2* [*simp*]: (*case ONE of ONE* $\Rightarrow$ *t*) = *t*
  ⟨*proof*⟩

lemma *one-case3* [*simp*]: (*case x of ONE* $\Rightarrow$ *ONE*) = *x*
  ⟨*proof*⟩

end

25 Fixed point operator and admissibility

theory *Fix*
  imports *Cfun*
begin

default-sort *pcpo*

25.1 Iteration

primrec *iterate* :: *nat* $\Rightarrow$ (*'a::cpo* $\rightarrow$ *'a*) $\rightarrow$ (*'a* $\rightarrow$ *'a*)
  where
    iterate 0 = (Λ *F x. x*)
  | *iterate* (*Suc n*) = (Λ *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* (λ*i. iterate i*$\cdot$*F*$\cdot$$\bot$)
  ⟨*proof*⟩

25.2 Least fixed point operator

definition *fix* :: (*'a* $\rightarrow$ *'a*) $\rightarrow$ *'a*

where $fix = (\Lambda\ F.\ \bigsqcup i.\ iterate\ i \cdot F \cdot \bot)$

Binder syntax for *fix*

abbreviation $fix\text{-}syn :: ('a \Rightarrow 'a) \Rightarrow 'a$ (**binder** μ *10*)
where $fix\text{-}syn\ (\lambda x.\ f\ x) \equiv fix \cdot (\Lambda\ x.\ f\ x)$

notation (*ASCII*)
fix-syn (**binder** *FIX* *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*: $f = fix \cdot F \Longrightarrow f = F \cdot f$
⟨*proof*⟩

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

strictness of *fix*

lemma *fix-bottom-iff*: $fix \cdot F = \bot \longleftrightarrow F \cdot \bot = \bot$

$\langle proof \rangle$

lemma *fix-strict*: $F{\cdot}\bot = \bot \Longrightarrow fix{\cdot}F = \bot$
$\langle proof \rangle$

lemma *fix-defined*: $F{\cdot}\bot \neq \bot \Longrightarrow fix{\cdot}F \neq \bot$
$\langle proof \rangle$

fix applied to identity and constant functions

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

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

25.3 Fixed point induction

lemma *fix-ind*: $adm\ P \Longrightarrow P\ \bot \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow P\ (F{\cdot}x)) \Longrightarrow P\ (fix{\cdot}F)$
$\langle proof \rangle$

lemma *cont-fix-ind*: $cont\ F \Longrightarrow adm\ P \Longrightarrow P\ \bot \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow P\ (F\ x)) \Longrightarrow P\ (fix{\cdot}(Abs\text{-}cfun\ F))$
$\langle proof \rangle$

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

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

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

lemma *cont-parallel-fix-ind*:
assumes $cont\ F$ **and** $cont\ G$
assumes $adm\ (\lambda x.\ P\ (fst\ x)\ (snd\ x))$
assumes $P\ \bot\ \bot$
assumes $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (F\ x)\ (G\ y)$
shows $P\ (fix{\cdot}(Abs\text{-}cfun\ F))\ (fix{\cdot}(Abs\text{-}cfun\ G))$
$\langle proof \rangle$

25.4 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*:
$fix{\cdot}(F{::}'a \times 'b \rightarrow 'a \times 'b) =$
$(\mu\ x.\ fst\ (F{\cdot}(x,\ \mu\ y.\ snd\ (F{\cdot}(x,\ y)))),$
$\mu\ y.\ snd\ (F{\cdot}(\mu\ x.\ fst\ (F{\cdot}(x,\ \mu\ y.\ snd\ (F{\cdot}(x,\ y)))),\ y)))$
(**is** $fix{\cdot}F = (?x,\ ?y)$)
⟨*proof*⟩

end

26 Package for defining recursive functions in HOLCF

theory *Fixrec*
imports *Cprod Sprod Ssum Up One Tr Fix*
keywords *fixrec* :: *thy-defn*
begin

26.1 Pattern-match monad

default-sort *cpo*

pcpodef $'a\ match = UNIV{::}(one\ {+}{+}\ 'a\ u)\ set$
⟨*proof*⟩

definition
fail :: *'a match* **where**
$fail = Abs\text{-}match\ (sinl{\cdot}ONE)$

definition
succeed :: $'a \rightarrow 'a\ match$ **where**
$succeed = (\Lambda\ x.\ Abs\text{-}match\ (sinr{\cdot}(up{\cdot}x)))$

lemma *matchE* [*case-names bottom fail succeed*, *cases type*: *match*]:
$[\![p = \bot \Longrightarrow Q;\ p = fail \Longrightarrow Q;\ \bigwedge x.\ p = succeed{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

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

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

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

lemma *succeed-neq-fail* [*simp*]:

$succeed \cdot x \neq fail$ $fail \neq succeed \cdot x$
⟨*proof*⟩

26.1.1 Run operator

definition
$run :: 'a\ match \to 'a::pcpo$ **where**
$run = (\Lambda\ m.\ sscase \cdot \bot \cdot (fup \cdot ID) \cdot (Rep\text{-}match\ m))$

rewrite rules for run

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

lemma *run-fail* [*simp*]: $run \cdot fail = \bot$
⟨*proof*⟩

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

26.1.2 Monad plus operator

definition
$mplus :: 'a\ match \to 'a\ match \to 'a\ match$ **where**
$mplus = (\Lambda\ m1\ m2.\ sscase \cdot (\Lambda\ \text{-}.\ m2) \cdot (\Lambda\ \text{-}.\ m1) \cdot (Rep\text{-}match\ m1))$

abbreviation
$mplus\text{-}syn :: ['a\ match, 'a\ match] \Rightarrow 'a\ match$ (**infixr** +++ *65*) **where**
$m1 +\!+\!+ m2 == mplus \cdot m1 \cdot m2$

rewrite rules for mplus

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

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

lemma *mplus-succeed* [*simp*]: $succeed \cdot x +\!+\!+ m = succeed \cdot x$
⟨*proof*⟩

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

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

26.2 Match functions for built-in types

default-sort *pcpo*

definition
$match\text{-}bottom :: 'a \to 'c\ match \to 'c\ match$
where
$match\text{-}bottom = (\Lambda\ x\ k.\ seq \cdot x \cdot fail)$

definition
$match\text{-}Pair :: 'a::cpo \times 'b::cpo \to ('a \to 'b \to 'c\ match) \to 'c\ match$
where
$match\text{-}Pair = (\Lambda\ x\ k.\ csplit \cdot k \cdot x)$

definition
$match\text{-}spair :: 'a \otimes 'b \to ('a \to 'b \to 'c\ match) \to 'c\ match$
where
$match\text{-}spair = (\Lambda\ x\ k.\ ssplit \cdot k \cdot x)$

definition
$match\text{-}sinl :: 'a \oplus 'b \to ('a \to 'c\ match) \to 'c\ match$
where
$match\text{-}sinl = (\Lambda\ x\ k.\ sscase \cdot k \cdot (\Lambda\ b.\ fail) \cdot x)$

definition
$match\text{-}sinr :: 'a \oplus 'b \to ('b \to 'c\ match) \to 'c\ match$
where
$match\text{-}sinr = (\Lambda\ x\ k.\ sscase \cdot (\Lambda\ a.\ fail) \cdot k \cdot x)$

definition
$match\text{-}up :: 'a::cpo\ u \to ('a \to 'c\ match) \to 'c\ match$
where
$match\text{-}up = (\Lambda\ x\ k.\ fup \cdot k \cdot x)$

definition
$match\text{-}ONE :: one \to 'c\ match \to 'c\ match$
where
$match\text{-}ONE = (\Lambda\ ONE\ k.\ k)$

definition
$match\text{-}TT :: tr \to 'c\ match \to 'c\ match$
where
$match\text{-}TT = (\Lambda\ x\ k.\ If\ x\ then\ k\ else\ fail)$

definition
$match\text{-}FF :: tr \to 'c\ match \to 'c\ match$
where
$match\text{-}FF = (\Lambda\ x\ k.\ If\ x\ then\ fail\ else\ k)$

lemma *match-bottom-simps* [*simp*]:
$match\text{-}bottom \cdot x \cdot k = (if\ x = \bot\ then\ \bot\ else\ fail)$
$\langle proof \rangle$

lemma *match-Pair-simps* [*simp*]:
$match\text{-}Pair\cdot(x,\ y)\cdot k = k\cdot x\cdot y$
⟨*proof*⟩

lemma *match-spair-simps* [*simp*]:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow match\text{-}spair\cdot(:x,\ y:)\cdot k = k\cdot x\cdot y$
$match\text{-}spair\cdot\bot\cdot k = \bot$
⟨*proof*⟩

lemma *match-sinl-simps* [*simp*]:
$x \neq \bot \Longrightarrow match\text{-}sinl\cdot(sinl\cdot x)\cdot k = k\cdot x$
$y \neq \bot \Longrightarrow match\text{-}sinl\cdot(sinr\cdot y)\cdot k = fail$
$match\text{-}sinl\cdot\bot\cdot k = \bot$
⟨*proof*⟩

lemma *match-sinr-simps* [*simp*]:
$x \neq \bot \Longrightarrow match\text{-}sinr\cdot(sinl\cdot x)\cdot k = fail$
$y \neq \bot \Longrightarrow match\text{-}sinr\cdot(sinr\cdot y)\cdot k = k\cdot y$
$match\text{-}sinr\cdot\bot\cdot k = \bot$
⟨*proof*⟩

lemma *match-up-simps* [*simp*]:
$match\text{-}up\cdot(up\cdot x)\cdot k = k\cdot x$
$match\text{-}up\cdot\bot\cdot k = \bot$
⟨*proof*⟩

lemma *match-ONE-simps* [*simp*]:
$match\text{-}ONE\cdot ONE\cdot k = k$
$match\text{-}ONE\cdot\bot\cdot k = \bot$
⟨*proof*⟩

lemma *match-TT-simps* [*simp*]:
$match\text{-}TT\cdot TT\cdot k = k$
$match\text{-}TT\cdot FF\cdot k = fail$
$match\text{-}TT\cdot\bot\cdot k = \bot$
⟨*proof*⟩

lemma *match-FF-simps* [*simp*]:
$match\text{-}FF\cdot FF\cdot k = k$
$match\text{-}FF\cdot TT\cdot k = fail$
$match\text{-}FF\cdot\bot\cdot k = \bot$
⟨*proof*⟩

26.3 Mutual recursion

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

lemma *Pair-equalI*: $[\![x \equiv fst\ p;\ y \equiv snd\ p]\!] \Longrightarrow (x,\ y) \equiv p$
⟨*proof*⟩

lemma *Pair-eqD1*: $(x, y) = (x', y') \Longrightarrow x = x'$
$\langle proof \rangle$

lemma *Pair-eqD2*: $(x, y) = (x', y') \Longrightarrow y = y'$
$\langle proof \rangle$

lemma *def-cont-fix-eq*:
$[\![f \equiv fix\cdot(Abs\text{-}cfun\ F);\ cont\ F]\!] \Longrightarrow f = F\ f$
$\langle proof \rangle$

lemma *def-cont-fix-ind*:
$[\![f \equiv fix\cdot(Abs\text{-}cfun\ F);\ cont\ F;\ adm\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F\ x)]\!] \Longrightarrow P\ f$
$\langle proof \rangle$

lemma for proving rewrite rules

lemma *ssubst-lhs*: $[\![t = s;\ P\ s = Q]\!] \Longrightarrow P\ t = Q$
$\langle proof \rangle$

26.4 Initializing the fixrec package

$\langle ML \rangle$

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

end

27 Domain package support

theory *Domain-Aux*
imports *Map-Functions Fixrec*
begin

27.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
fixes $abs :: 'a \rightarrow 'b$
fixes $rep :: 'b \rightarrow 'a$
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*
$\langle proof \rangle$

lemma *abs-below*: $(abs\cdot x \sqsubseteq abs\cdot y) = (x \sqsubseteq y)$
$\langle proof \rangle$

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-bottom-iff*: $(abs{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

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

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

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

lemma *iso-swap*: $(x = abs{\cdot}y) = (rep{\cdot}x = y)$
$\langle proof \rangle$

end

27.2 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*: $\bigwedge x.\ rep{\cdot}(abs{\cdot}x) = x$
assumes *rep-iso*: $\bigwedge y.\ abs{\cdot}(rep{\cdot}y) = y$
shows $deflation\ d \Longrightarrow deflation\ (abs\ oo\ d\ oo\ rep)$
$\langle proof \rangle$

lemma *deflation-chain-min*:
assumes *chain*: *chain d*
assumes *defl*: $\bigwedge n.\ deflation\ (d\ n)$
shows $d\ m{\cdot}(d\ n{\cdot}x) = d\ (min\ m\ n){\cdot}x$
$\langle proof \rangle$

lemma *lub-ID-take-lemma*:
assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
assumes $\bigwedge n.\ t\ n{\cdot}x = t\ n{\cdot}y$ **shows** $x = y$
$\langle proof \rangle$

lemma *lub-ID-reach*:
assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
shows $(\bigsqcup n.\ t\ n{\cdot}x) = x$
$\langle proof \rangle$

lemma *lub-ID-take-induct*:
assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
assumes *adm P* **and** $\bigwedge n.\ P\ (t\ n{\cdot}x)$ **shows** $P\ x$
$\langle proof \rangle$

27.3 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
$decisive :: ('a{::}pcpo \rightarrow 'a) \Rightarrow bool$
where
$decisive\ d \longleftrightarrow (\forall x.\ d{\cdot}x = x \vee d{\cdot}x = \bot)$

lemma *decisiveI*: $(\bigwedge x.\ d{\cdot}x = x \lor d{\cdot}x = \bot) \Longrightarrow decisive\ d$
⟨*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 $decisive\ (ssum\text{-}map{\cdot}f{\cdot}g)$
⟨*proof*⟩

lemma *decisive-sprod-map*:
assumes f: $decisive\ f$
assumes g: $decisive\ g$
shows $decisive\ (sprod\text{-}map{\cdot}f{\cdot}g)$
⟨*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*⟩

27.4 Proofs about constructor functions

Lemmas for proving nchotomy rule:

lemma *ex-one-bottom-iff*:
$(\exists x.\ P\ x \land x \neq \bot) = P\ ONE$

⟨*proof*⟩

lemma *ex-up-bottom-iff*:
$(\exists x.\ P\ x \wedge x \neq \bot) = (\exists x.\ P\ (up{\cdot}x))$
⟨*proof*⟩

lemma *ex-sprod-bottom-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-bottom-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-bottom-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-bottom-iffs* =
ex-ssum-bottom-iff
ex-sprod-up-bottom-iff
ex-sprod-bottom-iff
ex-up-bottom-iff
ex-one-bottom-iff

Rules for turning nchotomy into exhaust:

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*

Rules for proving constructor properties

lemmas *con-strict-rules* =

sinl-strict sinr-strict spair-strict1 spair-strict2

lemmas *con-bottom-iff-rules* =
sinl-bottom-iff sinr-bottom-iff spair-bottom-iff up-defined ONE-defined

lemmas *con-below-iff-rules* =
sinl-below sinr-below sinl-below-sinr sinr-below-sinl con-bottom-iff-rules

lemmas *con-eq-iff-rules* =
sinl-eq sinr-eq sinl-eq-sinr sinr-eq-sinl con-bottom-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-bottom-iff-rules* =
cfcomp2 sfst-bottom-iff ssnd-bottom-iff

lemmas *take-con-rules* =
ssum-map-sinl′ ssum-map-sinr′ sprod-map-spair′ u-map-up
deflation-strict deflation-ID ID1 cfcomp2

27.5 ML setup

named-theorems *domain-deflation theorems like deflation a ==> deflation* (*foo-map*$*a*)
and *domain-map-ID theorems like foo-map*$*ID* = *ID*

⟨*ML*⟩

end

28 Domain package

theory *Domain*
imports *Representable Domain-Aux*
keywords
lazy unsafe **and**
domaindef domain :: *thy-defn* **and**

domain-isomorphism :: *thy-decl*
begin

default-sort *domain*

28.1 Representations of types

lemma *emb-prj*: $emb \cdot ((prj \cdot x) :: 'a) = cast \cdot DEFL('a) \cdot x$
⟨*proof*⟩

lemma *emb-prj-emb*:
fixes $x :: 'a$
assumes $DEFL('a) \sqsubseteq DEFL('b)$
shows $emb \cdot (prj \cdot (emb \cdot x) :: 'b) = emb \cdot x$
⟨*proof*⟩

lemma *prj-emb-prj*:
assumes $DEFL('a) \sqsubseteq DEFL('b)$
shows $prj \cdot (emb \cdot (prj \cdot x :: 'b)) = (prj \cdot x :: 'a)$
⟨*proof*⟩

Isomorphism lemmas used internally by the domain package:

lemma *domain-abs-iso*:
fixes *abs* **and** *rep*
assumes *DEFL*: $DEFL('b) = DEFL('a)$
assumes *abs-def*: $(abs :: 'a \to 'b) \equiv prj\ oo\ emb$
assumes *rep-def*: $(rep :: 'b \to 'a) \equiv prj\ oo\ emb$
shows $rep \cdot (abs \cdot x) = x$
⟨*proof*⟩

lemma *domain-rep-iso*:
fixes *abs* **and** *rep*
assumes *DEFL*: $DEFL('b) = DEFL('a)$
assumes *abs-def*: $(abs :: 'a \to 'b) \equiv prj\ oo\ emb$
assumes *rep-def*: $(rep :: 'b \to 'a) \equiv prj\ oo\ emb$
shows $abs \cdot (rep \cdot x) = x$
⟨*proof*⟩

28.2 Deflations as sets

definition *defl-set* :: $'a::bifinite\ defl \Rightarrow 'a\ set$
where $defl\text{-}set\ A = \{x.\ cast \cdot A \cdot x = x\}$

lemma *adm-defl-set*: $adm\ (\lambda x.\ x \in defl\text{-}set\ A)$
⟨*proof*⟩

lemma *defl-set-bottom*: $\bot \in defl\text{-}set\ A$
⟨*proof*⟩

lemma *defl-set-cast* [*simp*]: $cast{\cdot}A{\cdot}x \in$ *defl-set* A
⟨*proof*⟩

lemma *defl-set-subset-iff*: *defl-set* $A \subseteq$ *defl-set* $B \longleftrightarrow A \sqsubseteq B$
⟨*proof*⟩

28.3 Proving a subtype is representable

Temporarily relax type constraints.

⟨*ML*⟩

lemma *typedef-domain-class*:
  fixes Rep :: $'a$::*pcpo* $\Rightarrow$ *udom*
  fixes Abs :: *udom* $\Rightarrow$ $'a$::*pcpo*
  fixes t :: *udom defl*
  assumes *type*: *type-definition Rep Abs* (*defl-set* t)
  assumes *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
  assumes *emb*: $emb \equiv (\Lambda\ x.\ Rep\ x)$
  assumes *prj*: $prj \equiv (\Lambda\ x.\ Abs\ (cast{\cdot}t{\cdot}x))$
  assumes *defl*: $defl \equiv (\lambda\ a{::}'a\ itself.\ t)$
  assumes *liftemb*: $(liftemb :: 'a\ u \rightarrow udom\ u) \equiv u\text{-}map{\cdot}emb$
  assumes *liftprj*: $(liftprj :: udom\ u \rightarrow 'a\ u) \equiv u\text{-}map{\cdot}prj$
  assumes *liftdefl*: $(liftdefl :: 'a\ itself \Rightarrow \text{-}) \equiv (\lambda t.\ liftdefl\text{-}of{\cdot}DEFL('a))$
  shows $OFCLASS('a,\ domain\text{-}class)$
⟨*proof*⟩

lemma *typedef-DEFL*:
  assumes $defl \equiv (\lambda a{::}'a{::}pcpo\ itself.\ t)$
  shows $DEFL('a{::}pcpo) = t$
⟨*proof*⟩

Restore original typing constraints.

⟨*ML*⟩

28.4 Isomorphic deflations

definition *isodefl* :: $('a \rightarrow 'a) \Rightarrow udom\ defl \Rightarrow bool$
  where $isodefl\ d\ t \longleftrightarrow cast{\cdot}t = emb\ oo\ d\ oo\ prj$

definition *isodefl'* :: $('a{::}predomain \rightarrow 'a) \Rightarrow udom\ u\ defl \Rightarrow bool$
  where $isodefl'\ d\ t \longleftrightarrow cast{\cdot}t = liftemb\ oo\ u\text{-}map{\cdot}d\ oo\ liftprj$

lemma *isodeflI*: $(\bigwedge x.\ cast{\cdot}t{\cdot}x = emb{\cdot}(d{\cdot}(prj{\cdot}x))) \Longrightarrow isodefl\ d\ t$
⟨*proof*⟩

lemma *cast-isodefl*: $isodefl\ d\ t \Longrightarrow cast{\cdot}t = (\Lambda\ x.\ emb{\cdot}(d{\cdot}(prj{\cdot}x)))$
⟨*proof*⟩

lemma *isodefl-strict*: $isodefl\ d\ t \Longrightarrow d{\cdot}\bot = \bot$

⟨*proof*⟩

lemma *isodefl-imp-deflation*:
fixes $d :: 'a \to 'a$
assumes *isodefl d t* **shows** *deflation d*
⟨*proof*⟩

lemma *isodefl-ID-DEFL*: *isodefl* $(ID :: 'a \to 'a)$ $DEFL('a)$
⟨*proof*⟩

lemma *isodefl-LIFTDEFL*:
isodefl' $(ID :: 'a \to 'a)$ $LIFTDEFL('a::predomain)$
⟨*proof*⟩

lemma *isodefl-DEFL-imp-ID*: *isodefl* $(d :: 'a \to 'a)$ $DEFL('a) \Longrightarrow d = ID$
⟨*proof*⟩

lemma *isodefl-bottom*: *isodefl* $\perp$ $\perp$
⟨*proof*⟩

lemma *adm-isodefl*:
cont f $\Longrightarrow$ *cont g* $\Longrightarrow$ *adm* $(\lambda x.\ isodefl\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *isodefl-lub*:
assumes *chain d* **and** *chain t*
assumes $\bigwedge i.\ isodefl\ (d\ i)\ (t\ i)$
shows *isodefl* $(\bigsqcup i.\ d\ i)$ $(\bigsqcup i.\ t\ i)$
⟨*proof*⟩

lemma *isodefl-fix*:
assumes $\bigwedge d\ t.\ isodefl\ d\ t \Longrightarrow isodefl\ (f \cdot d)\ (g \cdot t)$
shows *isodefl* $(fix \cdot f)$ $(fix \cdot g)$
⟨*proof*⟩

lemma *isodefl-abs-rep*:
fixes *abs* **and** *rep* **and** *d*
assumes *DEFL*: $DEFL('b) = DEFL('a)$
assumes *abs-def*: $(abs :: 'a \to 'b) \equiv prj\ oo\ emb$
assumes *rep-def*: $(rep :: 'b \to 'a) \equiv prj\ oo\ emb$
shows *isodefl d t* $\Longrightarrow$ *isodefl* $(abs\ oo\ d\ oo\ rep)$ t
⟨*proof*⟩

lemma *isodefl'-liftdefl-of*: *isodefl d t* $\Longrightarrow$ *isodefl'* d $(liftdefl\text{-}of \cdot t)$
⟨*proof*⟩

lemma *isodefl-sfun*:
isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
isodefl $(sfun\text{-}map \cdot d1 \cdot d2)$ $(sfun\text{-}defl \cdot t1 \cdot t2)$

⟨*proof*⟩

lemma *isodefl-ssum*:
  isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
    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-prod*:
  isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
    isodefl (*prod-map*·*d1*·*d2*) (*prod-defl*·*t1*·*t2*)
⟨*proof*⟩

lemma *isodefl-u*:
  isodefl d t $\Longrightarrow$ *isodefl* (*u-map*·*d*) (*u-defl*·*t*)
⟨*proof*⟩

lemma *isodefl-u-liftdefl*:
  isodefl' d t $\Longrightarrow$ *isodefl* (*u-map*·*d*) (*u-liftdefl*·*t*)
⟨*proof*⟩

lemma *encode-prod-u-map*:
  encode-prod-u·(*u-map*·(*prod-map*·*f*·*g*)·(*decode-prod-u*·*x*))
    = *sprod-map*·(*u-map*·*f*)·(*u-map*·*g*)·*x*
⟨*proof*⟩

lemma *isodefl-prod-u*:
  assumes *isodefl' d1 t1* **and** *isodefl' d2 t2*
  shows *isodefl'* (*prod-map*·*d1*·*d2*) (*prod-liftdefl*·*t1*·*t2*)
⟨*proof*⟩

lemma *encode-cfun-map*:
  encode-cfun·(*cfun-map*·*f*·*g*·(*decode-cfun*·*x*))
    = *sfun-map*·(*u-map*·*f*)·*g*·*x*
⟨*proof*⟩

lemma *isodefl-cfun*:
  assumes *isodefl* (*u-map*·*d1*) *t1* **and** *isodefl d2 t2*
  shows *isodefl* (*cfun-map*·*d1*·*d2*) (*sfun-defl*·*t1*·*t2*)
⟨*proof*⟩

28.5 Setting up the domain package

named-theorems *domain-defl-simps theorems like DEFL('a t) = t-defl$DEFL('a)*
 and *domain-isodefl theorems like isodefl d t ==> isodefl (foo-map$d) (foo-defl$t)*

⟨*ML*⟩

lemmas [*domain-defl-simps*] =
DEFL-cfun DEFL-sfun DEFL-ssum DEFL-sprod DEFL-prod DEFL-u
liftdefl-eq LIFTDEFL-prod u-liftdefl-liftdefl-of

lemmas [*domain-map-ID*] =
cfun-map-ID sfun-map-ID ssum-map-ID sprod-map-ID prod-map-ID u-map-ID

lemmas [*domain-isodefl*] =
isodefl-u isodefl-sfun isodefl-ssum isodefl-sprod
isodefl-cfun isodefl-prod isodefl-prod-u isodefl′-liftdefl-of
isodefl-u-liftdefl

lemmas [*domain-deflation*] =
deflation-cfun-map deflation-sfun-map deflation-ssum-map
deflation-sprod-map deflation-prod-map deflation-u-map

⟨*ML*⟩

end

29 A compact basis for powerdomains

theory *Compact-Basis*
imports *Universal*
begin

default-sort *bifinite*

29.1 A compact basis for powerdomains

definition *pd-basis* = $\{S::'a\ compact\text{-}basis\ set.\ finite\ S \wedge S \neq \{\}\}$

typedef *′a pd-basis* = *pd-basis* :: *′a compact-basis set set*
⟨*proof*⟩

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

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

The powerdomain basis type is countable.

lemma *pd-basis-countable*: $\exists f::'a\ pd\text{-}basis \Rightarrow nat.\ inj\ f$
⟨*proof*⟩

29.2 Unit and plus constructors

definition
PDUnit :: *'a compact-basis* ⇒ *'a pd-basis* **where**
PDUnit = (*λx. 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*⟩

29.3 Fold operator

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

lemma *fold-pd-PDUnit*:

assumes *semilattice f*
shows *fold-pd g f* (*PDUnit x*) = *g x*
⟨*proof*⟩

lemma *fold-pd-PDPlus*:
assumes *semilattice f*
shows *fold-pd g f* (*PDPlus t u*) = *f* (*fold-pd g f t*) (*fold-pd g f u*)
⟨*proof*⟩

end

30 Upper powerdomain

theory *UpperPD*
imports *Compact-Basis*
begin

30.1 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*:

$(PDPlus\ t\ u \leq\sharp\ PDUnit\ a) = (t \leq\sharp\ PDUnit\ a \vee u \leq\sharp\ PDUnit\ a)$
⟨*proof*⟩

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

lemma *upper-le-induct* [*induct set*: *upper-le*]:
assumes *le*: $t \leq\sharp\ u$
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *2*: $\bigwedge t\ u\ a.\ P\ t\ (PDUnit\ a) \Longrightarrow P\ (PDPlus\ t\ u)\ (PDUnit\ a)$
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ t\ v]\!] \Longrightarrow P\ t\ (PDPlus\ u\ v)$
shows $P\ t\ u$
⟨*proof*⟩

30.2 Type definition

typedef *'a upper-pd* ((′(-′)♯)) =
{*S*::*'a pd-basis set*. *upper-le.ideal S*}
⟨*proof*⟩

instantiation *upper-pd* :: (*bifinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-upper-pd x* $\subseteq$ *Rep-upper-pd y*

instance ⟨*proof*⟩
end

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

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

definition
upper-principal :: *'a pd-basis* ⇒ *'a upper-pd* **where**
upper-principal t = *Abs-upper-pd* $\{u.\ u \leq\sharp\ t\}$

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

Upper powerdomain is pointed

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

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

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

30.3 Monadic unit and plus

definition
upper-unit :: $'a \to 'a$ *upper-pd* **where**
upper-unit = *compact-basis.extension* (λa. *upper-principal* (*PDUnit a*))

definition
upper-plus :: $'a$ *upper-pd* $\to$ $'a$ *upper-pd* $\to$ $'a$ *upper-pd* **where**
upper-plus = *upper-pd.extension* (λt. *upper-pd.extension* (λu.
upper-principal (*PDPlus t u*)))

abbreviation
upper-add :: $'a$ *upper-pd* $\Rightarrow$ $'a$ *upper-pd* $\Rightarrow$ $'a$ *upper-pd*
(**infixl** $\cup\sharp$ *65*) **where**
xs $\cup\sharp$ *ys* == *upper-plus·xs·ys*

syntax
-upper-pd :: *args* $\Rightarrow$ *logic* ({-}$\sharp$)

translations
$\{x,xs\}\sharp$ == $\{x\}\sharp$ $\cup\sharp$ $\{xs\}\sharp$
$\{x\}\sharp$ == *CONST upper-unit·x*

lemma *upper-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}$\sharp$ = *upper-principal* (*PDUnit a*)
⟨*proof*⟩

lemma *upper-plus-principal* [*simp*]:
upper-principal t $\cup\sharp$ *upper-principal u* = *upper-principal* (*PDPlus t u*)
⟨*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*: $xs \cup\sharp\ ys \sqsubseteq xs$
⟨*proof*⟩

lemma *upper-plus-below2*: $xs \cup\sharp\ ys \sqsubseteq ys$
⟨*proof*⟩

lemma *upper-plus-greatest*: $[\![xs \sqsubseteq ys;\ xs \sqsubseteq zs]\!] \Longrightarrow xs \sqsubseteq ys \cup\sharp\ zs$
⟨*proof*⟩

lemma *upper-below-plus-iff* [*simp*]:
$xs \sqsubseteq ys \cup\sharp\ zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
⟨*proof*⟩

lemma *upper-plus-below-unit-iff* [*simp*]:
$xs \cup\sharp\ ys \sqsubseteq \{z\}\sharp \longleftrightarrow xs \sqsubseteq \{z\}\sharp \vee ys \sqsubseteq \{z\}\sharp$
⟨*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*]: $\bot \cup\sharp\ ys = \bot$
⟨*proof*⟩

lemma *upper-plus-strict2* [*simp*]: $xs \cup\sharp\ \bot = \bot$
⟨*proof*⟩

lemma *upper-unit-bottom-iff* [*simp*]: $\{x\}\sharp = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *upper-plus-bottom-iff* [*simp*]:
$xs \cup\sharp\ ys = \bot \longleftrightarrow xs = \bot \vee ys = \bot$
⟨*proof*⟩

lemma *compact-upper-unit*: *compact* $x \Longrightarrow$ *compact* $\{x\}\sharp$
⟨*proof*⟩

lemma *compact-upper-unit-iff* [*simp*]: *compact* $\{x\}\sharp \longleftrightarrow$ *compact* x
$\langle proof \rangle$

lemma *compact-upper-plus* [*simp*]:
$[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* (*xs* $\cup\sharp$ *ys*)
$\langle proof \rangle$

30.4 Induction rules

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

lemma *upper-pd-induct*
[*case-names adm upper-unit upper-plus*, *induct type*: *upper-pd*]:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs \cup\sharp\ ys)$
shows *P* (*xs*::'*a upper-pd*)
$\langle proof \rangle$

30.5 Monadic bind

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

lemma *ACI-upper-bind*:
semilattice ($\lambda x\ y.\ \Lambda\ f.\ x\cdot f \cup\sharp\ y\cdot f$)
$\langle proof \rangle$

lemma *upper-bind-basis-simps* [*simp*]:
upper-bind-basis (*PDUnit a*) =
($\Lambda\ f.\ f\cdot$(*Rep-compact-basis a*))
upper-bind-basis (*PDPlus t u*) =
($\Lambda\ f.$ *upper-bind-basis t*$\cdot f \cup\sharp$ *upper-bind-basis u*$\cdot f$)
$\langle proof \rangle$

lemma *upper-bind-basis-mono*:
$t \leq\sharp\ u \Longrightarrow$ *upper-bind-basis t* $\sqsubseteq$ *upper-bind-basis u*
$\langle proof \rangle$

definition
upper-bind :: '*a upper-pd* $\rightarrow$ ('*a* $\rightarrow$ '*b upper-pd*) $\rightarrow$ '*b upper-pd* **where**

upper-bind = *upper-pd.extension upper-bind-basis*

syntax
-upper-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
((*3*⋃♯-∈-./ -) [*0*, *0*, *10*] *10*)

translations
⋃♯*x*∈*xs*. *e* == *CONST upper-bind*·*xs*·(Λ *x*. *e*)

lemma *upper-bind-principal* [*simp*]:
upper-bind·(*upper-principal t*) = *upper-bind-basis t*
⟨*proof*⟩

lemma *upper-bind-unit* [*simp*]:
upper-bind·{*x*}♯·*f* = *f*·*x*
⟨*proof*⟩

lemma *upper-bind-plus* [*simp*]:
upper-bind·(*xs* ∪♯ *ys*)·*f* = *upper-bind*·*xs*·*f* ∪♯ *upper-bind*·*ys*·*f*
⟨*proof*⟩

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

lemma *upper-bind-bind*:
upper-bind·(*upper-bind*·*xs*·*f*)·*g* = *upper-bind*·*xs*·(Λ *x*. *upper-bind*·(*f*·*x*)·*g*)
⟨*proof*⟩

30.6 Map

definition
upper-map :: (′*a* → ′*b*) → ′*a upper-pd* → ′*b upper-pd* **where**
upper-map = (Λ *f xs*. *upper-bind*·*xs*·(Λ *x*. {*f*·*x*}♯))

lemma *upper-map-unit* [*simp*]:
upper-map·*f*·{*x*}♯ = {*f*·*x*}♯
⟨*proof*⟩

lemma *upper-map-plus* [*simp*]:
upper-map·*f*·(*xs* ∪♯ *ys*) = *upper-map*·*f*·*xs* ∪♯ *upper-map*·*f*·*ys*
⟨*proof*⟩

lemma *upper-map-bottom* [*simp*]: *upper-map*·*f*·⊥ = {*f*·⊥}♯
⟨*proof*⟩

lemma *upper-map-ident*: *upper-map*·(Λ *x*. *x*)·*xs* = *xs*
⟨*proof*⟩

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

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

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

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

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

30.7 Upper powerdomain is bifinite

lemma *approx-chain-upper-map*:
assumes *approx-chain a*
shows *approx-chain* $(\lambda i.\ upper\text{-}map \cdot (a\ i))$
⟨*proof*⟩

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

30.8 Join

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

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

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

lemma *upper-join-bottom* [*simp*]: $upper\text{-}join \cdot \bot = \bot$

⟨*proof*⟩

lemma *upper-join-map-unit*:
upper-join·(*upper-map*·*upper-unit*·*xs*) = *xs*
⟨*proof*⟩

lemma *upper-join-map-join*:
upper-join·(*upper-map*·*upper-join*·*xsss*) = *upper-join*·(*upper-join*·*xsss*)
⟨*proof*⟩

lemma *upper-join-map-map*:
upper-join·(*upper-map*·(*upper-map*·*f*)·*xss*) =
upper-map·*f*·(*upper-join*·*xss*)
⟨*proof*⟩

end

31 Lower powerdomain

theory *LowerPD*
imports *Compact-Basis*
begin

31.1 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*]:
$(PDUnit\ a \leq\flat\ PDUnit\ b) = (a \sqsubseteq b)$
⟨*proof*⟩

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

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

lemma *lower-le-induct* [*induct set*: *lower-le*]:
assumes *le*: $t \leq\flat\ u$
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *2*: $\bigwedge t\ u\ a.\ P\ (PDUnit\ a)\ t \Longrightarrow P\ (PDUnit\ a)\ (PDPlus\ t\ u)$
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ v;\ P\ u\ v]\!] \Longrightarrow P\ (PDPlus\ t\ u)\ v$
shows $P\ t\ u$
⟨*proof*⟩

31.2 Type definition

typedef *′a lower-pd* $(('(-')\flat)) =$
$\{S::'a\ pd\text{-}basis\ set.\ lower\text{-}le.ideal\ S\}$
⟨*proof*⟩

instantiation *lower-pd* :: (*bifinite*) *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* :: (*bifinite*) *po*
⟨*proof*⟩

instance *lower-pd* :: (*bifinite*) *cpo*
⟨*proof*⟩

definition
lower-principal :: *′a pd-basis* ⇒ *′a lower-pd* **where**
$lower\text{-}principal\ t = Abs\text{-}lower\text{-}pd\ \{u.\ u \leq\flat\ t\}$

interpretation *lower-pd*:
ideal-completion lower-le lower-principal Rep-lower-pd
⟨*proof*⟩

Lower powerdomain is pointed

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

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

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

31.3 Monadic unit and plus

definition
lower-unit :: *'a* $\to$ *'a lower-pd* **where**
lower-unit = *compact-basis.extension* (λ*a*. *lower-principal* (*PDUnit a*))

definition
lower-plus :: *'a lower-pd* $\to$ *'a lower-pd* $\to$ *'a lower-pd* **where**
lower-plus = *lower-pd.extension* (λ*t*. *lower-pd.extension* (λ*u*.
lower-principal (*PDPlus t u*)))

abbreviation
lower-add :: *'a lower-pd* $\Rightarrow$ *'a lower-pd* $\Rightarrow$ *'a lower-pd*
(**infixl** $\cup\flat$ *65*) **where**
xs $\cup\flat$ *ys* == *lower-plus*·*xs*·*ys*

syntax
-lower-pd :: *args* $\Rightarrow$ *logic* ({-}$\flat$)

translations
{*x*,*xs*}$\flat$ == {*x*}$\flat$ $\cup\flat$ {*xs*}$\flat$
{*x*}$\flat$ == *CONST lower-unit*·*x*

lemma *lower-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}$\flat$ = *lower-principal* (*PDUnit a*)
⟨*proof*⟩

lemma *lower-plus-principal* [*simp*]:
lower-principal t $\cup\flat$ *lower-principal u* = *lower-principal* (*PDPlus t u*)
⟨*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*: $xs \sqsubseteq xs \cup\flat\ ys$
⟨*proof*⟩

lemma *lower-plus-below2*: $ys \sqsubseteq xs \cup\flat\ ys$
⟨*proof*⟩

lemma *lower-plus-least*: $[\![xs \sqsubseteq zs;\ ys \sqsubseteq zs]\!] \Longrightarrow xs \cup\flat\ ys \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-plus-below-iff* [*simp*]:
$xs \cup\flat\ ys \sqsubseteq zs \longleftrightarrow xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
⟨*proof*⟩

lemma *lower-unit-below-plus-iff* [*simp*]:
$\{x\}\flat \sqsubseteq ys \cup\flat\ zs \longleftrightarrow \{x\}\flat \sqsubseteq ys \vee \{x\}\flat \sqsubseteq zs$
⟨*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-bottom-iff* [*simp*]: $\{x\}\flat = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *lower-plus-bottom-iff* [*simp*]:
$xs \cup\flat\ ys = \bot \longleftrightarrow xs = \bot \wedge ys = \bot$
⟨*proof*⟩

lemma *lower-plus-strict1* [*simp*]: $\bot \cup\flat\ ys = ys$
⟨*proof*⟩

lemma *lower-plus-strict2* [*simp*]: $xs \cup\flat\ \bot = xs$

⟨*proof*⟩

lemma *compact-lower-unit*: *compact* $x \Longrightarrow$ *compact* $\{x\}\flat$
⟨*proof*⟩

lemma *compact-lower-unit-iff* [*simp*]: *compact* $\{x\}\flat \longleftrightarrow$ *compact* x
⟨*proof*⟩

lemma *compact-lower-plus* [*simp*]:
$[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* $(xs \cup\flat\ ys)$
⟨*proof*⟩

31.4 Induction rules

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

lemma *lower-pd-induct*
[*case-names adm lower-unit lower-plus*, *induct type*: *lower-pd*]:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\flat$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs \cup\flat\ ys)$
shows $P\ (xs{::}'a\ lower\text{-}pd)$
⟨*proof*⟩

31.5 Monadic bind

definition
lower-bind-basis ::
$'a\ pd\text{-}basis \Rightarrow ('a \rightarrow\ 'b\ lower\text{-}pd) \rightarrow\ 'b\ lower\text{-}pd$ **where**
lower-bind-basis = *fold-pd*
$(\lambda a.\ \Lambda\ f.\ f\cdot(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x\cdot f \cup\flat\ y\cdot f)$

lemma *ACI-lower-bind*:
semilattice $(\lambda x\ y.\ \Lambda\ f.\ x\cdot f \cup\flat\ y\cdot f)$
⟨*proof*⟩

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

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

definition
lower-bind :: $'a$ *lower-pd* $\rightarrow$ ($'a \rightarrow$ $'b$ *lower-pd*) $\rightarrow$ $'b$ *lower-pd* **where**
lower-bind = *lower-pd.extension lower-bind-basis*

syntax
-lower-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
((*3*⋃♭-∈-./ -) [*0*, *0*, *10*] *10*)

translations
⋃♭*x*∈*xs*. *e* == *CONST lower-bind*·*xs*·(Λ *x*. *e*)

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*⟩

lemma *lower-bind-bind*:
lower-bind·(*lower-bind*·*xs*·*f*)·*g* = *lower-bind*·*xs*·(Λ *x*. *lower-bind*·(*f*·*x*)·*g*)
⟨*proof*⟩

31.6 Map

definition
lower-map :: ($'a \rightarrow$ $'b$) $\rightarrow$ $'a$ *lower-pd* $\rightarrow$ $'b$ *lower-pd* **where**
lower-map = (Λ *f xs*. *lower-bind*·*xs*·(Λ *x*. {*f*·*x*}♭))

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-map-bottom* [*simp*]: *lower-map*·*f*·⊥ = {*f*·⊥}♭

⟨*proof*⟩

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

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

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

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

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

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

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

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

31.7 Lower powerdomain is bifinite

lemma *approx-chain-lower-map*:
assumes *approx-chain* a
shows *approx-chain* $(\lambda i.\ \mathit{lower\text{-}map}\cdot(a\ i))$
⟨*proof*⟩

instance *lower-pd* :: (*bifinite*) *bifinite*
⟨*proof*⟩

31.8 Join

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

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

lemma *lower-join-plus* [*simp*]:
lower-join·(*xss* $\cup\flat$ *yss*) = *lower-join*·*xss* $\cup\flat$ *lower-join*·*yss*
⟨*proof*⟩

lemma *lower-join-bottom* [*simp*]: *lower-join*·$\bot$ = $\bot$
⟨*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*⟩

end

32 Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

32.1 Basis preorder

definition
convex-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** $\leq\natural$ *50*) **where**
convex-le = (λ*u v*. *u* $\leq\sharp$ *v* $\wedge$ *u* $\leq\flat$ *v*)

lemma *convex-le-refl* [*simp*]: *t* $\leq\natural$ *t*
⟨*proof*⟩

lemma *convex-le-trans*: ⟦*t* $\leq\natural$ *u*; *u* $\leq\natural$ *v*⟧ $\Longrightarrow$ *t* $\leq\natural$ *v*
⟨*proof*⟩

interpretation *convex-le*: *preorder convex-le*
⟨*proof*⟩

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\natural$ *t*
⟨*proof*⟩

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

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

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

lemma *convex-le-PDUnit-lemma1*:
$(PDUnit\ a \leq\natural\ t) = (\forall\, b{\in}Rep\text{-}pd\text{-}basis\ t.\ a \sqsubseteq b)$
⟨*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*⟩

32.2 Type definition

typedef *'a convex-pd* $(('(\text{-}')\natural)) =$
$\{S::'a\ pd\text{-}basis\ set.\ convex\text{-}le.ideal\ S\}$
⟨*proof*⟩

instantiation *convex-pd* :: (*bifinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow Rep\text{-}convex\text{-}pd\ x \subseteq Rep\text{-}convex\text{-}pd\ y$

instance ⟨*proof*⟩
end

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

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

definition
convex-principal :: $'a$ *pd-basis* ⇒ $'a$ *convex-pd* **where**
convex-principal t = *Abs-convex-pd* $\{u.\ u \leq\natural\ t\}$

interpretation *convex-pd*:
ideal-completion convex-le convex-principal Rep-convex-pd
⟨*proof*⟩

Convex powerdomain is pointed

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

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

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

32.3 Monadic unit and plus

definition
convex-unit :: $'a \rightarrow 'a$ *convex-pd* **where**
convex-unit = *compact-basis.extension* (λ*a*. *convex-principal* (*PDUnit a*))

definition
convex-plus :: $'a$ *convex-pd* → $'a$ *convex-pd* → $'a$ *convex-pd* **where**
convex-plus = *convex-pd.extension* (λ*t*. *convex-pd.extension* (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* ⇒ *logic* ({-}♮)

translations
{*x*,*xs*}♮ == {*x*}♮ ∪♮ {*xs*}♮
{*x*}♮ == *CONST convex-unit*·*x*

lemma *convex-unit-Rep-compact-basis* [*simp*]:
$\{$*Rep-compact-basis a*$\}\natural$ = *convex-principal* (*PDUnit a*)
⟨*proof*⟩

lemma *convex-plus-principal* [*simp*]:
convex-principal t $\cup\natural$ *convex-principal u* = *convex-principal* (*PDPlus t u*)
⟨*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*]:
$\{x\}\natural \sqsubseteq ys \cup\natural zs \longleftrightarrow \{x\}\natural \sqsubseteq ys \wedge \{x\}\natural \sqsubseteq zs$
⟨*proof*⟩

lemma *convex-plus-below-unit-iff* [*simp*]:
$xs \cup\natural ys \sqsubseteq \{z\}\natural \longleftrightarrow xs \sqsubseteq \{z\}\natural \wedge ys \sqsubseteq \{z\}\natural$
⟨*proof*⟩

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

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

lemma *convex-unit-strict* [*simp*]: $\{\bot\}\natural = \bot$
⟨*proof*⟩

lemma *convex-unit-bottom-iff* [*simp*]: $\{x\}\natural = \bot \longleftrightarrow x = \bot$
⟨*proof*⟩

lemma *compact-convex-unit*: *compact x* $\Longrightarrow$ *compact* $\{x\}\natural$
⟨*proof*⟩

lemma *compact-convex-unit-iff* [*simp*]: *compact* $\{x\}\natural \longleftrightarrow$ *compact x*

⟨*proof*⟩

lemma *compact-convex-plus* [*simp*]:
$[\![compact\ xs;\ compact\ ys]\!] \Longrightarrow compact\ (xs \cup\natural\ ys)$
⟨*proof*⟩

32.4 Induction rules

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

lemma *convex-pd-induct*
[*case-names adm convex-unit convex-plus*, *induct type*: *convex-pd*]:
assumes P: $adm\ P$
assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs \cup\natural\ ys)$
shows $P\ (xs{::}'a\ convex\text{-}pd)$
⟨*proof*⟩

32.5 Monadic bind

definition
convex-bind-basis ::
$'a\ pd\text{-}basis \Rightarrow ('a \rightarrow 'b\ convex\text{-}pd) \rightarrow 'b\ convex\text{-}pd$ **where**
$convex\text{-}bind\text{-}basis = fold\text{-}pd$
$(\lambda a.\ \Lambda\ f.\ f\cdot(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x\cdot f \cup\natural\ y\cdot f)$

lemma *ACI-convex-bind*:
$semilattice\ (\lambda x\ y.\ \Lambda\ f.\ x\cdot f \cup\natural\ y\cdot f)$
⟨*proof*⟩

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

lemma *convex-bind-basis-mono*:
$t \le\natural\ u \Longrightarrow convex\text{-}bind\text{-}basis\ t \sqsubseteq convex\text{-}bind\text{-}basis\ u$
⟨*proof*⟩

definition
$convex\text{-}bind :: 'a\ convex\text{-}pd \rightarrow ('a \rightarrow 'b\ convex\text{-}pd) \rightarrow 'b\ convex\text{-}pd$ **where**
$convex\text{-}bind = convex\text{-}pd.extension\ convex\text{-}bind\text{-}basis$

syntax
  -convex-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
    ((*3*⋃♮-∈-./ -) [*0*, *0*, *10*] *10*)

translations
  ⋃♮*x*∈*xs*. *e* == *CONST convex-bind·xs*·(Λ *x*. *e*)

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*⟩

lemma *convex-bind-bind*:
  convex-bind·(*convex-bind·xs·f*)·*g* =
    convex-bind·xs·(Λ *x*. *convex-bind*·(*f*·*x*)·*g*)
⟨*proof*⟩

32.6 Map

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

lemma *convex-map-unit* [*simp*]:
  convex-map·f·{*x*}♮ = {*f*·*x*}♮
⟨*proof*⟩

lemma *convex-map-plus* [*simp*]:
  convex-map·f·(*xs* ∪♮ *ys*) = *convex-map·f·xs* ∪♮ *convex-map·f·ys*
⟨*proof*⟩

lemma *convex-map-bottom* [*simp*]: *convex-map·f*·⊥ = {*f*·⊥}♮
⟨*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*:
  $convex\text{-}map \cdot f \cdot (convex\text{-}map \cdot g \cdot xs) = convex\text{-}map \cdot (\Lambda\ x.\ f \cdot (g \cdot x)) \cdot xs$
⟨*proof*⟩

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

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

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

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

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

32.7 Convex powerdomain is bifinite

lemma *approx-chain-convex-map*:
  assumes *approx-chain a*
  shows *approx-chain* $(\lambda i.\ convex\text{-}map \cdot (a\ i))$
  ⟨*proof*⟩

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

32.8 Join

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

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

lemma *convex-join-plus* [*simp*]:
  $convex\text{-}join \cdot (xss \cup\natural\ yss) = convex\text{-}join \cdot xss \cup\natural\ convex\text{-}join \cdot yss$
⟨*proof*⟩

lemma *convex-join-bottom* [*simp*]: $convex\text{-}join \cdot \bot = \bot$

⟨*proof*⟩

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

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

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

32.9 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: $t \leq\natural\ u \Longrightarrow t \leq\sharp\ u$
⟨*proof*⟩

definition
  $convex\text{-}to\text{-}upper :: 'a\ convex\text{-}pd \rightarrow 'a\ upper\text{-}pd$ **where**
  $convex\text{-}to\text{-}upper = convex\text{-}pd.extension\ upper\text{-}principal$

lemma *convex-to-upper-principal* [*simp*]:
  $convex\text{-}to\text{-}upper\cdot(convex\text{-}principal\ t) = upper\text{-}principal\ t$
⟨*proof*⟩

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

lemma *convex-to-upper-plus* [*simp*]:
  $convex\text{-}to\text{-}upper\cdot(xs \cup\natural\ ys) = convex\text{-}to\text{-}upper\cdot xs \cup\sharp\ convex\text{-}to\text{-}upper\cdot ys$
⟨*proof*⟩

lemma *convex-to-upper-bind* [*simp*]:
  $convex\text{-}to\text{-}upper\cdot(convex\text{-}bind\cdot xs\cdot f) =$
    $upper\text{-}bind\cdot(convex\text{-}to\text{-}upper\cdot xs)\cdot(convex\text{-}to\text{-}upper\ oo\ f)$
⟨*proof*⟩

lemma *convex-to-upper-map* [*simp*]:
  $convex\text{-}to\text{-}upper\cdot(convex\text{-}map\cdot f\cdot xs) = upper\text{-}map\cdot f\cdot(convex\text{-}to\text{-}upper\cdot xs)$
⟨*proof*⟩

lemma *convex-to-upper-join* [*simp*]:
  $convex\text{-}to\text{-}upper\cdot(convex\text{-}join\cdot xss) =$
    $upper\text{-}bind\cdot(convex\text{-}to\text{-}upper\cdot xss)\cdot convex\text{-}to\text{-}upper$

⟨*proof*⟩

Convex to lower

lemma *convex-le-imp-lower-le*: $t \leq\natural\ u \Longrightarrow t \leq\flat\ u$
⟨*proof*⟩

definition
convex-to-lower :: *'a convex-pd* → *'a lower-pd* **where**
convex-to-lower = *convex-pd.extension 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·$\{x\}\natural = \{x\}\flat$
⟨*proof*⟩

lemma *convex-to-lower-plus* [*simp*]:
convex-to-lower·(*xs* ∪♮ *ys*) = *convex-to-lower*·*xs* ∪♭ *convex-to-lower*·*ys*
⟨*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*]
for *xs ys zs ws*

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

33 Powerdomains

theory *Powerdomains*
imports *ConvexPD Domain*
begin

33.1 Universal domain embeddings

definition *upper-emb* = *udom-emb* ($\lambda i.$ *upper-map*·(*udom-approx* i))
definition *upper-prj* = *udom-prj* ($\lambda i.$ *upper-map*·(*udom-approx* i))

definition *lower-emb* = *udom-emb* ($\lambda i.$ *lower-map*·(*udom-approx* i))
definition *lower-prj* = *udom-prj* ($\lambda i.$ *lower-map*·(*udom-approx* i))

definition *convex-emb* = *udom-emb* ($\lambda i.$ *convex-map*·(*udom-approx* i))
definition *convex-prj* = *udom-prj* ($\lambda i.$ *convex-map*·(*udom-approx* i))

lemma *ep-pair-upper*: *ep-pair upper-emb upper-prj*
⟨*proof*⟩

lemma *ep-pair-lower*: *ep-pair lower-emb lower-prj*
⟨*proof*⟩

lemma *ep-pair-convex*: *ep-pair convex-emb convex-prj*
⟨*proof*⟩

33.2 Deflation combinators

definition *upper-defl* :: *udom defl* → *udom defl*
where *upper-defl* = *defl-fun1 upper-emb upper-prj upper-map*

definition *lower-defl* :: *udom defl* → *udom defl*
where *lower-defl* = *defl-fun1 lower-emb lower-prj lower-map*

definition *convex-defl* :: *udom defl* → *udom defl*
where *convex-defl* = *defl-fun1 convex-emb convex-prj convex-map*

lemma *cast-upper-defl*:
cast·(upper-defl·A) = upper-emb oo upper-map·(cast·A) oo upper-prj
⟨*proof*⟩

lemma *cast-lower-defl*:
cast·(lower-defl·A) = lower-emb oo lower-map·(cast·A) oo lower-prj
⟨*proof*⟩

lemma *cast-convex-defl*:
cast·(convex-defl·A) = convex-emb oo convex-map·(cast·A) oo convex-prj
⟨*proof*⟩

33.3 Domain class instances

instantiation *upper-pd* :: (*domain*) *domain*
begin

definition
emb = upper-emb oo upper-map·emb

definition
prj = upper-map·prj oo upper-prj

definition
defl (t::'a upper-pd itself) = upper-defl·DEFL('a)

definition
(liftemb :: 'a upper-pd u → udom u) = u-map·emb

definition
(liftprj :: udom u → 'a upper-pd u) = u-map·prj

definition
liftdefl (t::'a upper-pd itself) = liftdefl-of·DEFL('a upper-pd)

instance ⟨*proof*⟩

end

instantiation *lower-pd* :: (*domain*) *domain*
begin

definition
emb = lower-emb oo lower-map·emb

definition
prj = lower-map·prj oo lower-prj

definition

defl (*t*::*'a lower-pd itself*) = *lower-defl·DEFL*(*'a*)

definition
(*liftemb* :: *'a lower-pd u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → *'a lower-pd u*) = *u-map·prj*

definition
liftdefl (*t*::*'a lower-pd itself*) = *liftdefl-of·DEFL*(*'a lower-pd*)

instance ⟨*proof*⟩

end

instantiation *convex-pd* :: (*domain*) *domain*
begin

definition
emb = *convex-emb oo convex-map·emb*

definition
prj = *convex-map·prj oo convex-prj*

definition
defl (*t*::*'a convex-pd itself*) = *convex-defl·DEFL*(*'a*)

definition
(*liftemb* :: *'a convex-pd u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → *'a convex-pd u*) = *u-map·prj*

definition
liftdefl (*t*::*'a convex-pd itself*) = *liftdefl-of·DEFL*(*'a convex-pd*)

instance ⟨*proof*⟩

end

lemma *DEFL-upper*: *DEFL*(*'a*::*domain upper-pd*) = *upper-defl·DEFL*(*'a*)
⟨*proof*⟩

lemma *DEFL-lower*: *DEFL*(*'a*::*domain lower-pd*) = *lower-defl·DEFL*(*'a*)
⟨*proof*⟩

lemma *DEFL-convex*: *DEFL*(*'a*::*domain convex-pd*) = *convex-defl·DEFL*(*'a*)
⟨*proof*⟩

33.4 Isomorphic deflations

lemma *isodefl-upper*:
isodefl d t $\Longrightarrow$ *isodefl* (*upper-map*·*d*) (*upper-defl*·*t*)
⟨*proof*⟩

lemma *isodefl-lower*:
isodefl d t $\Longrightarrow$ *isodefl* (*lower-map*·*d*) (*lower-defl*·*t*)
⟨*proof*⟩

lemma *isodefl-convex*:
isodefl d t $\Longrightarrow$ *isodefl* (*convex-map*·*d*) (*convex-defl*·*t*)
⟨*proof*⟩

33.5 Domain package setup for powerdomains

lemmas [*domain-defl-simps*] = *DEFL-upper DEFL-lower DEFL-convex*
lemmas [*domain-map-ID*] = *upper-map-ID lower-map-ID convex-map-ID*
lemmas [*domain-isodefl*] = *isodefl-upper isodefl-lower isodefl-convex*

lemmas [*domain-deflation*] =
deflation-upper-map deflation-lower-map deflation-convex-map

⟨*ML*⟩

end

theory *HOLCF*
imports
Main
Domain
Powerdomains
begin

default-sort *domain*

end