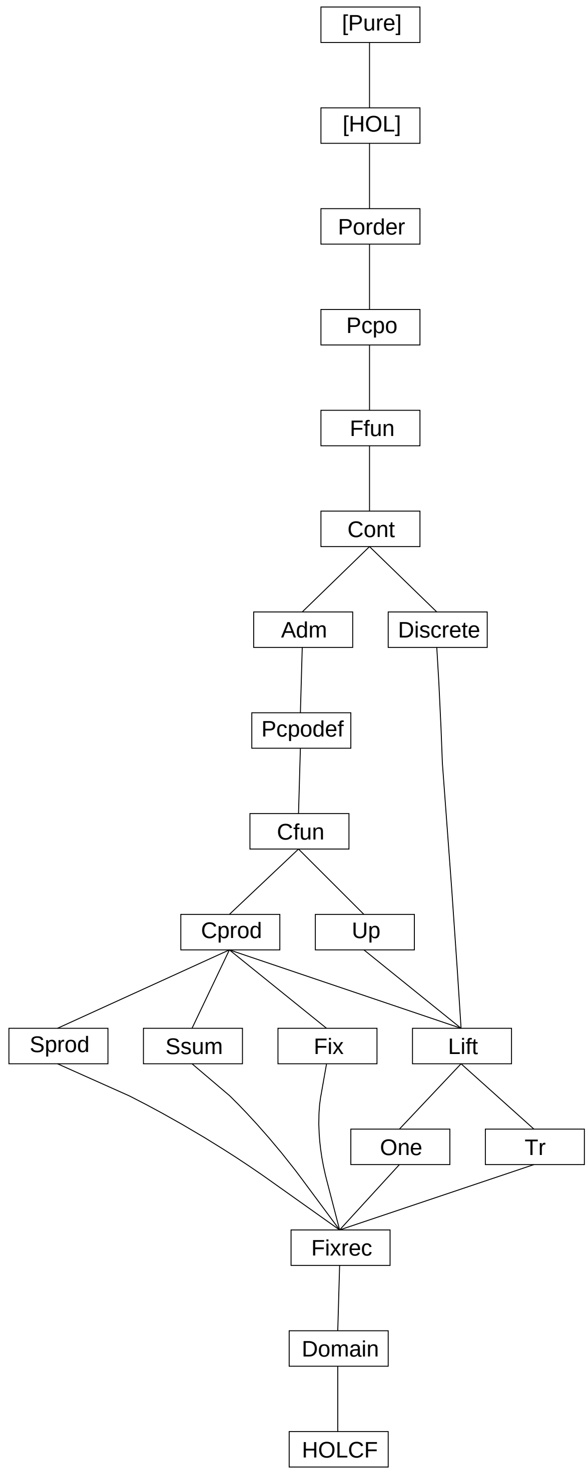
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

November 22, 2007

Contents


[Pure]
[HOL]
Porder
Pcpo
Ffun
Cont
Adm
Discrete
Pcpodef
Cfun
Cprod
Up
Sprod
Ssum
Fix
Lift
One
Tr
Fixrec
Domain
HOLCF

1 Porder: Partial orders

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

1.1 Type class for partial orders

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

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

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

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

minimal fixes least element

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

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

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

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

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

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

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

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

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

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

1.2 Chains and least upper bounds

class definitions

definition
$is\text{-}ub :: [{'a}\ set, {'a}{::}po] \Rightarrow bool$ (**infixl** <| 55) **where**
$(S <| x) = (\forall y.\ y \in S \longrightarrow y \sqsubseteq x)$

definition
$is\text{-}lub :: [{'a}\ set, {'a}{::}po] \Rightarrow bool$ (**infixl** <<| 55) **where**
$(S <<| x) = (S <| x \wedge (\forall u.\ S <| u \longrightarrow x \sqsubseteq u))$

definition
— Arbitrary chains are total orders
$tord :: {'a}{::}po\ set \Rightarrow bool$ **where**
$tord\ S = (\forall x\ y.\ x \in S \wedge y \in S \longrightarrow (x \sqsubseteq y \vee y \sqsubseteq x))$

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

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

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

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

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

notation (*xsymbols*)
Lub (**binder** $\bigsqcup$ *10*)

lubs are unique

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

chains are monotone functions

lemma *chain-mono* [*rule-format*]: $chain\ F \Longrightarrow x < y \longrightarrow F\ x \sqsubseteq F\ y$
⟨*proof*⟩

lemma *chain-mono3*: $[\![chain\ F;\ x \le y]\!] \Longrightarrow F\ x \sqsubseteq F\ y$
⟨*proof*⟩

The range of a chain is a totally ordered

lemma *chain-tord*: $chain\ F \Longrightarrow tord\ (range\ F)$
⟨*proof*⟩

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

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

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

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

access to some definition as inference rule

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

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

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

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

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

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

technical lemmas about (least) upper bounds of chains

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

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

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

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

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

results about finite chains

lemma *lub-finch1*: $[\![chain\ C;\ max\text{-}in\text{-}chain\ i\ C]\!] \Longrightarrow range\ C <<|\ C\ i$
⟨*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-tord-has-max* [*rule-format*]:
$finite\ S \Longrightarrow S \neq \{\} \longrightarrow tord\ S \longrightarrow (\exists\, y{\in}S.\ \forall\, x{\in}S.\ x \sqsubseteq y)$
⟨*proof*⟩

lemma *finite-range-imp-finch*:
$[\![chain\ Y;\ finite\ (range\ Y)]\!] \Longrightarrow finite\text{-}chain\ Y$
⟨*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 *lub-bin-chain*:
$x \sqsubseteq y \Longrightarrow range\ (\lambda i{::}nat.\ if\ i{=}0\ then\ x\ else\ y) <<|\ y$
⟨*proof*⟩

the maximal element in a chain is its lub

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

the lub of a constant chain is the constant

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

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

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

end

2 Pcpo: Classes cpo and pcpo

theory *Pcpo*
imports *Porder*
begin

2.1 Complete partial orders

The class cpo of chain complete partial orders

axclass *cpo* < *po*
— class axiom:
cpo: $chain\ S \Longrightarrow \exists x.\ range\ S <<|\ x$

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

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

Properties of the lub

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

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

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

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

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

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

lemma *lub-mono*:

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

the = relation between two chains is preserved by their lubs

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

more results about mono and = of lubs of chains

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

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

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

lemma *ch2ch-lub*:
fixes $Y :: nat \Rightarrow nat \Rightarrow 'a::cpo$
assumes *1*: $\bigwedge j.\ chain\ (\lambda i.\ Y\ i\ j)$
assumes *2*: $\bigwedge i.\ chain\ (\lambda j.\ Y\ i\ j)$
shows $chain\ (\lambda i.\ \bigsqcup j.\ Y\ i\ j)$
⟨*proof*⟩

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

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

2.2 Pointed cpos

The class pcpo of pointed cpos

axclass *pcpo* $<$ *cpo*
least: $\exists\, x.\ \forall\, y.\ x \sqsubseteq y$

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

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

derive the old rule minimal

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

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

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

⟨*ML*⟩

useful lemmas about $\perp$

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

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

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

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

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

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

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

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

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

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

axclass *chfin* < *po*
chfin: $\forall Y.\ chain\ Y \longrightarrow (\exists n.\ max\text{-}in\text{-}chain\ n\ Y)$

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

some properties for chfin and flat

chfin types are cpo

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

instance *chfin* < *cpo*
⟨*proof*⟩

flat types are chfin

lemma *flat-imp-chfin*:
$\forall Y{::}nat \Rightarrow {'a}{::}flat.\ chain\ Y \longrightarrow (\exists n.\ max\text{-}in\text{-}chain\ n\ Y)$
⟨*proof*⟩

instance *flat* < *chfin*
⟨*proof*⟩

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

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

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

lemmata for improved admissibility introdution rule

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

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

$\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
$\langle proof \rangle$

end

3 Ffun: Class instances for the full function space

theory *Ffun*
imports *Pcpo*
begin

3.1 Full function space is a partial order

instance *fun* :: (*type*, *sq-ord*) *sq-ord* $\langle proof \rangle$

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

lemma *refl-less-fun*: $(f::'a::type \Rightarrow 'b::po) \sqsubseteq f$
$\langle proof \rangle$

lemma *antisym-less-fun*:
$[\![(f1::'a::type \Rightarrow 'b::po) \sqsubseteq f2;\ f2 \sqsubseteq f1]\!] \Longrightarrow f1 = f2$
$\langle proof \rangle$

lemma *trans-less-fun*:
$[\![(f1::'a::type \Rightarrow 'b::po) \sqsubseteq f2;\ f2 \sqsubseteq f3]\!] \Longrightarrow f1 \sqsubseteq f3$
$\langle proof \rangle$

instance *fun* :: (*type*, *po*) *po*
$\langle proof \rangle$

make the symbol << accessible for type fun

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

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

3.2 Full function space is chain complete

chains of functions yield chains in the po range

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

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

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

lemma *ub2ub-fun*:
$range\ (S::nat \Rightarrow {}'a \Rightarrow {}'b::po)\ <|\ u \Longrightarrow range\ (\lambda i.\ S\ i\ x)\ <|\ u\ x$
⟨*proof*⟩

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

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

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

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

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

3.3 Full function space is pointed

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

lemma *least-fun*: $\exists x::{}'a \Rightarrow {}'b::pcpo.\ \forall y.\ x \sqsubseteq y$
⟨*proof*⟩

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

for compatibility with old HOLCF-Version

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

function application is strict in the left argument

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

end

4 Cont: Continuity and monotonicity

theory *Cont*

imports *Ffun*
begin

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

defaultsort *po*

4.1 Definitions

definition
$monofun :: ('a \Rightarrow 'b) \Rightarrow bool$ — monotonicity **where**
$monofun\ f = (\forall x\ y.\ x \sqsubseteq y \longrightarrow f\ x \sqsubseteq f\ y)$

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

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

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

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

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

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

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

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

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

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

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

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

4.2 $monofun\ f \wedge contlub\ f \equiv cont\ f$

monotone functions map chains to chains

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

monotone functions map upper bound to upper bounds

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

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

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

first a lemma about binary chains

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

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

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

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

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

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

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

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

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

4.3 Continuity of basic functions

The identity function is continuous

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

constant functions are continuous

lemma *cont-const*: $cont\ (\lambda x.\ c)$
$\langle proof \rangle$

if-then-else is continuous

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

4.4 Propagation of monotonicity and continuity

the lub of a chain of monotone functions is monotone

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

the lub of a chain of continuous functions is continuous

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

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

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

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

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

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

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

lemma *mono2mono-lambda*: $(\bigwedge y.\ monofun\ (\lambda x.\ f\ x\ y)) \Longrightarrow monofun\ f$
$\langle proof \rangle$

lemma *cont2cont-lambda*: $(\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)) \Longrightarrow cont\ f$
$\langle proof \rangle$

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

lemma *contlub-lambda*:
$(\bigwedge x{::}{'a}{::}type.\ chain\ (\lambda i.\ S\ i\ x{::}{'b}{::}cpo))$

$\Longrightarrow (\lambda x.\ \bigsqcup i.\ S\ i\ x) = (\bigsqcup i.\ (\lambda x.\ S\ i\ x))$
$\langle proof \rangle$

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

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

lemma *cont2contlub-app*:
$[\![cont\ f;\ \forall\, x.\ cont\ (f\ x);\ cont\ t]\!] \Longrightarrow contlub\ (\lambda x.\ (f\ x)\ (t\ x))$
$\langle proof \rangle$

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

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

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

4.5 Finite chains and flat pcpos

monotone functions map finite chains to finite chains

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

The same holds for continuous functions

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

lemma *chfindom-monofun2cont*: $monofun\ f \Longrightarrow cont\ (f{::}'a{::}chfin \Rightarrow 'b{::}pcpo)$
$\langle proof \rangle$

some properties of flat

lemma *flatdom-strict2mono*: $f\ \bot = \bot \Longrightarrow monofun\ (f{::}'a{::}flat \Rightarrow 'b{::}pcpo)$
$\langle proof \rangle$

lemma *flatdom-strict2cont*: $f\ \bot = \bot \Longrightarrow cont\ (f{::}'a{::}flat \Rightarrow 'b{::}pcpo)$
$\langle proof \rangle$

end

5 Adm: Admissibility and compactness

theory *Adm*
imports *Cont*
begin

defaultsort *cpo*

5.1 Definitions

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

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

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

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

lemma *admD*: $[\![adm\ P;\ chain\ Y;\ \forall\, i.\ P\ (Y\ i)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
$\langle proof \rangle$

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

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

improved admissibility introduction

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

5.2 Admissibility on chain-finite types

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

lemma *adm-max-in-chain*:
$\forall\, Y.\ chain\ (Y::nat \Rightarrow 'a) \longrightarrow (\exists\, n.\ max\text{-}in\text{-}chain\ n\ Y)$
$\Longrightarrow adm\ (P::'a \Rightarrow bool)$
$\langle proof \rangle$

lemmas *adm-chfin* = *chfin* [*THEN adm-max-in-chain*, *standard*]

lemma *compact-chfin*: $compact\ (x{::}'a{::}chfin)$
⟨*proof*⟩

5.3 Admissibility of special formulae and propagation

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

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

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

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

lemmas *adm-all2* = *adm-all* [*rule-format*]
lemmas *adm-ball2* = *adm-ball* [*rule-format*]

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

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

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

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

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

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

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

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

lemma *adm-iff*:

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

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

admissibility and continuity

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

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

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

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

admissibility and compactness

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

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

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

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

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

lemmas *adm-lemmas* [*simp*] =
adm-not-free adm-conj adm-all2 adm-ball2 adm-disj adm-imp adm-iff
adm-less adm-eq adm-not-less
adm-compact-not-less adm-compact-neq adm-neq-compact adm-not-UU

⟨*ML*⟩

end

6 Pcpodef: Subtypes of pcpos

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

6.1 Proving a subtype is a partial order

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

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

6.2 Proving a subtype is chain-finite

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

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

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

6.3 Proving a subtype is complete

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

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

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

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

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

6.3.1 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 *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
shows *cont Rep*
$\langle proof \rangle$

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

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

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

6.4 Proving subtype elements are compact

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

6.5 Proving a subtype is pointed

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

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

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

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

6.5.1 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 *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *UU-in-A*: $\perp \in A$
  shows $Abs\ \perp = \perp$
⟨*proof*⟩

theorem *typedef-Rep-strict*:
  assumes *type*: *type-definition Rep Abs A*
    and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *UU-in-A*: $\perp \in A$

shows $Rep\ \bot = \bot$
⟨*proof*⟩

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

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

6.6 Proving a subtype is flat

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

6.7 HOLCF type definition package

⟨*ML*⟩

end

7 Cfun: The type of continuous functions

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

defaultsort *cpo*

7.1 Definition of continuous function type

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

lemma *adm-cont*: *adm cont*

⟨*proof*⟩

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

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

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

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

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

7.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: $'a$

⟨*ML*⟩

To avoid eta-contraction of body:

⟨*ML*⟩

Syntax for nested abstractions

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

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

⟨*ML*⟩

Dummy patterns for continuous abstraction

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

7.3 Continuous function space is pointed

lemma *UU-CFun*: ⊥ ∈ *CFun*
⟨*proof*⟩

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

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

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

function application is strict in its first argument

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

for compatibility with old HOLCF-Version

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

7.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

Eta-equality for continuous functions

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

Extensionality for continuous functions

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

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

Extensionality wrt. ordering for continuous functions

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

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

Congruence for continuous function application

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

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

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

7.5 Continuity of application

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

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

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

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

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

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

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

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

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

monotonicity of application

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

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

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

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

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

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

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

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

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

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

strictness

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

the lub of a chain of continous functions is monotone

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

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

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

the lub of a chain of cont. functions is continuous

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

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

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

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

7.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

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

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

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

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

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

continuity simplification procedure

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

$\langle ML \rangle$

7.7 Miscellaneous

Monotonicity of *Abs-CFun*

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

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

lemma *chfin-Rep-CFunR*: *chain* (*Y*::*nat* => '*a*::*cpo*->'*b*::*chfin*)
      ==> !*s*. ? *n*. *lub*(*range*(*Y*))$*s* = *Y n*$*s*
$\langle proof \rangle$

lemma *adm-chfindom*: $adm\ (\lambda(u::'a::cpo \to 'b::chfin).\ P(u\cdot s))$
$\langle proof \rangle$

7.8 Continuous injection-retraction pairs

Continuous retractions are strict.

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

⟨*proof*⟩

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

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

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

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

propagation of flatness and chain-finiteness by retractions

lemma *chfin2chfin*:
$\forall y.\ (f::'a::chfin \rightarrow 'b)\cdot(g\cdot y) = y$
$\Longrightarrow \forall Y::nat \Rightarrow 'b.\ chain\ Y \longrightarrow (\exists n.\ max\text{-}in\text{-}chain\ n\ Y)$
⟨*proof*⟩

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

a result about functions with flat codomain

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

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

7.9 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
$cfcomp\text{-}syn :: ['b \rightarrow 'c,\ 'a \rightarrow 'b] \Rightarrow 'a \rightarrow 'c$ (**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-strict* [*simp*]: $\perp\ oo\ f = \perp$
⟨*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*⟩

7.10 Strictified functions

defaultsort *pcpo*

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

results about strictify

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

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

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

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

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

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

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

7.11 Continuous let-bindings

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

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

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

end

8 Cprod: The cpo of cartesian products

theory *Cprod*
imports *Cfun*
begin

defaultsort *cpo*

8.1 Type *unit* is a pcpo

instance *unit* :: *sq-ord* ⟨*proof*⟩

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

instance *unit* :: *po*
⟨*proof*⟩

instance *unit* :: *cpo*
⟨*proof*⟩

instance *unit* :: *pcpo*
⟨*proof*⟩

definition
unit-when :: $'a \rightarrow unit \rightarrow 'a$ **where**
unit-when = $(\Lambda\ a\ \text{-}.\ a)$

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

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

8.2 Product type is a partial order

instance $*$ :: (*sq-ord*, *sq-ord*) *sq-ord* ⟨*proof*⟩

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

lemma *refl-less-cprod*: $(p::'a * 'b) \sqsubseteq p$
⟨*proof*⟩

lemma *antisym-less-cprod*: $[\![(p1::'a * 'b) \sqsubseteq p2;\ p2 \sqsubseteq p1]\!] \Longrightarrow p1 = p2$
⟨*proof*⟩

lemma *trans-less-cprod*: $[\![(p1::'a*'b) \sqsubseteq p2;\ p2 \sqsubseteq p3]\!] \Longrightarrow p1 \sqsubseteq p3$
⟨*proof*⟩

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

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

Pair (-,-) is monotone in both arguments

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

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

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

fst and *snd* are monotone

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

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

8.4 Product type is a cpo

lemma *lub-cprod*:
$chain\ S \Longrightarrow range\ S <<|\ (\bigsqcup i.\ fst\ (S\ i),\ \bigsqcup i.\ snd\ (S\ i))$
⟨*proof*⟩

lemma *thelub-cprod*:
$chain\ S \Longrightarrow lub\ (range\ S) = (\bigsqcup i.\ fst\ (S\ i),\ \bigsqcup i.\ snd\ (S\ i))$
⟨*proof*⟩

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

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

8.5 Product type is pointed

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

lemma *least-cprod*: $EX\ x{::}{'a}{::}pcpo * {'b}{::}pcpo.\ ALL\ y.\ x \sqsubseteq y$
⟨*proof*⟩

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

for compatibility with old HOLCF-Version

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

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

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

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

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

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

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

lemma *contlub-snd*: *contlub snd*

⟨*proof*⟩

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

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

8.7 Continuous versions of constants

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

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

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

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

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

syntax (*xsymbols*)
-ctuple :: $['a,\ args] \Rightarrow 'a * 'b$ ((*1*⟨-,/ -⟩))

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

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

8.8 Convert all lemmas to the continuous versions

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

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

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

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

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

lemma *cpair-strict*: $<\bot,\ \bot> = \bot$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *surjective-pairing-Cprod2*: $<cfst \cdot p,\ csnd \cdot p> = p$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

end

9 Sprod: The type of strict products

theory *Sprod*
imports *Cprod*
begin

defaultsort *pcpo*

9.1 Definition of strict product type

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

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

lemma *spair-lemma*:
  $<strictify \cdot (\Lambda\ b.\ a) \cdot b,\ strictify \cdot (\Lambda\ a.\ b) \cdot a> \in Sprod$
⟨*proof*⟩

9.2 Definitions of constants

definition
  sfst :: ('*a* ** '*b*) → '*a* **where**
  $sfst = (\Lambda\ p.\ cfst \cdot (Rep\text{-}Sprod\ p))$

definition

$ssnd :: ('a ** 'b) \to 'b$ **where**
$ssnd = (\Lambda\ p.\ csnd \cdot (Rep\text{-}Sprod\ p))$

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

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

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

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

9.3 Case analysis

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

lemma *Exh-Sprod2*:
$z = \bot \lor (\exists\, a\ b.\ z = (:a,\ b:) \land a \neq \bot \land b \neq \bot)$
⟨*proof*⟩

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

9.4 Properties of *spair*

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

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

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

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

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

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

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

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

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

9.5 Properties of *sfst* and *ssnd*

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

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

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

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

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

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

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

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

lemma *surjective-pairing-Sprod2*: $(:sfst\cdot p,\ ssnd\cdot p:) = p$
$\langle proof \rangle$

lemma *less-sprod*: $x \sqsubseteq y = (sfst\cdot x \sqsubseteq sfst\cdot y \land ssnd\cdot x \sqsubseteq ssnd\cdot y)$
$\langle proof \rangle$

lemma *eq-sprod*: $(x = y) = (sfst\cdot x = sfst\cdot y \land ssnd\cdot x = ssnd\cdot y)$
$\langle proof \rangle$

lemma *spair-less*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) = (x \sqsubseteq a \land y \sqsubseteq b)$
$\langle proof \rangle$

9.6 Properties of *ssplit*

lemma *ssplit1* [*simp*]: $ssplit\cdot f\cdot\bot = \bot$
$\langle proof \rangle$

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

lemma *ssplit3* [*simp*]: $ssplit\cdot spair\cdot z = z$
$\langle proof \rangle$

end

10 Ssum: The type of strict sums

theory *Ssum*
imports *Cprod*
begin

defaultsort *pcpo*

10.1 Definition of strict sum type

pcpodef (*Ssum*) $('a,\ 'b)$ ++ (**infixr** ++ *10*) =
$\{p{::}'a \times 'b.\ cfst\cdot p = \bot \lor csnd\cdot p = \bot\}$
$\langle proof \rangle$

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

10.2 Definitions of constructors

definition
$sinl :: 'a \to ('a ++ 'b)$ **where**
$sinl = (\Lambda\ a.\ Abs\text{-}Ssum <a, \bot>)$

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

10.3 Properties of *sinl* and *sinr*

lemma *sinl-Abs-Ssum*: $sinl \cdot a = Abs\text{-}Ssum <a, \bot>$
⟨*proof*⟩

lemma *sinr-Abs-Ssum*: $sinr \cdot b = Abs\text{-}Ssum <\bot, b>$
⟨*proof*⟩

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

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

lemma *compact-sinl* [*simp*]: $compact\ x \Longrightarrow compact\ (sinl \cdot x)$
⟨*proof*⟩

lemma *compact-sinr* [*simp*]: $compact\ x \Longrightarrow compact\ (sinr \cdot x)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

10.4 Case analysis

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

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

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

10.5 Ordering properties of *sinl* and *sinr*

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

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

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

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

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

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

10.6 Chains of strict sums

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

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

lemma *ssum-chain-lemma*:
$chain\ Y \Longrightarrow (\exists\, A.\ chain\ A \wedge Y = (\lambda i.\ sinl{\cdot}(A\ i))) \vee$
$(\exists\, B.\ chain\ B \wedge Y = (\lambda i.\ sinr{\cdot}(B\ i)))$
⟨*proof*⟩

10.7 Definitions of constants

definition
$Iwhen :: [{'a} \to {'c},\ {'b} \to {'c},\ {'a} \mathbin{++} {'b}] \Rightarrow {'c}$ **where**
$Iwhen = (\lambda f\ g\ s.$
$\quad if\ cfst{\cdot}(Rep\text{-}Ssum\ s) \neq \bot\ then\ f{\cdot}(cfst{\cdot}(Rep\text{-}Ssum\ s))\ else$
$\quad if\ csnd{\cdot}(Rep\text{-}Ssum\ s) \neq \bot\ then\ g{\cdot}(csnd{\cdot}(Rep\text{-}Ssum\ s))\ else\ \bot)$

rewrites for *Iwhen*

lemma *Iwhen1* [*simp*]: $Iwhen\ f\ g\ \bot = \bot$
⟨*proof*⟩

lemma *Iwhen2* [*simp*]: $x \neq \bot \Longrightarrow Iwhen\ f\ g\ (sinl{\cdot}x) = f{\cdot}x$
⟨*proof*⟩

lemma *Iwhen3* [*simp*]: $y \neq \bot \Longrightarrow Iwhen\ f\ g\ (sinr{\cdot}y) = g{\cdot}y$
⟨*proof*⟩

lemma *Iwhen4*: $Iwhen\ f\ g\ (sinl{\cdot}x) = strictify{\cdot}f{\cdot}x$
⟨*proof*⟩

lemma *Iwhen5*: $Iwhen\ f\ g\ (sinr{\cdot}y) = strictify{\cdot}g{\cdot}y$
⟨*proof*⟩

10.8 Continuity of *Iwhen*

Iwhen is continuous in all arguments

lemma *cont-Iwhen1*: $cont\ (\lambda f.\ Iwhen\ f\ g\ s)$
⟨*proof*⟩

lemma *cont-Iwhen2*: $cont\ (\lambda g.\ Iwhen\ f\ g\ s)$
⟨*proof*⟩

lemma *cont-Iwhen3*: $cont\ (\lambda s.\ Iwhen\ f\ g\ s)$
⟨*proof*⟩

10.9 Continuous versions of constants

definition
$sscase :: ({'a} \to {'c}) \to ({'b} \to {'c}) \to ({'a} \mathbin{++} {'b}) \to {'c}$ **where**

$sscase = (\Lambda\ f\ g\ s.\ Iwhen\ f\ g\ s)$

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

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

continuous versions of lemmas for *sscase*

lemma *beta-sscase*: $sscase \cdot f \cdot g \cdot s = Iwhen\ f\ g\ s$
⟨*proof*⟩

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

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

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

lemma *sscase4* [*simp*]: $sscase \cdot sinl \cdot sinr \cdot z = z$
⟨*proof*⟩

end

11 Up: The type of lifted values

theory *Up*
imports *Cfun*
begin

defaultsort *cpo*

11.1 Definition of new type for lifting

datatype $'a\ u = Ibottom \mid Iup\ 'a$

syntax (*xsymbols*)
$u :: type \Rightarrow type\ ((\text{-}_{\bot})\ [1000]\ 999)$

consts
$Ifup :: ('a \rightarrow 'b{::}pcpo) \Rightarrow 'a\ u \Rightarrow 'b$

primrec
$Ifup\ f\ Ibottom = \bot$

Ifup f (Iup x) = f·x

11.2 Ordering on lifted cpo

instance *u* :: (*sq-ord*) *sq-ord* ⟨*proof*⟩

defs (overloaded)
less-up-def:
$(op \sqsubseteq) \equiv (\lambda x\ y.\ case\ x\ of\ Ibottom \Rightarrow True \mid Iup\ a \Rightarrow$
$(case\ y\ of\ Ibottom \Rightarrow False \mid Iup\ b \Rightarrow a \sqsubseteq b))$

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

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

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

11.3 Lifted cpo is a partial order

lemma *refl-less-up*: $(x{::}'a\ u) \sqsubseteq x$
⟨*proof*⟩

lemma *antisym-less-up*: $[\![(x{::}'a\ u) \sqsubseteq y;\ y \sqsubseteq x]\!] \Longrightarrow x = y$
⟨*proof*⟩

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

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

11.4 Lifted cpo is a cpo

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

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

instance $u :: (cpo)\ cpo$
⟨*proof*⟩

11.5 Lifted cpo is pointed

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

instance $u :: (cpo)\ pcpo$
⟨*proof*⟩

for compatibility with old HOLCF-Version

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

11.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

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

continuity for *Ifup*

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

$\langle proof \rangle$

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

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

11.7 Continuous versions of constants

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

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

translations
$case\ l\ of\ CONST\ up{\cdot}x \Rightarrow t == CONST\ fup{\cdot}(\Lambda\ x.\ t){\cdot}l$
$\Lambda(CONST\ up{\cdot}x).\ t == CONST\ fup{\cdot}(\Lambda\ x.\ t)$

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

lemma *Exh-Up*: $z = \perp \vee (\exists\, x.\ z = up{\cdot}x)$
$\langle proof \rangle$

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

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

lemma *up-defined* [*simp*]: $up{\cdot}x \neq \perp$
$\langle proof \rangle$

lemma *not-up-less-UU* [*simp*]: $\neg\ up{\cdot}x \sqsubseteq \perp$
$\langle proof \rangle$

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

lemma *upE*: $[\![p = \perp \Longrightarrow Q;\ \bigwedge x.\ p = up{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
$\langle proof \rangle$

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

lemma *compact-up* [*simp*]: *compact* $x \Longrightarrow$ *compact* (*up*·*x*)
⟨*proof*⟩

properties of fup

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

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

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

end

12 Discrete: Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

12.1 Type *'a discr* is a partial order

instance *discr* :: (*type*) *sq-ord* ⟨*proof*⟩

defs (**overloaded**)
less-discr-def: ((*op* <<)::(*'a*::*type*)*discr*=>*'a discr*=>*bool*) == *op* =

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

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

12.2 Type *'a discr* is a cpo

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

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

lemma *discr-cpo*:

!!*S*. *chain S* ==> ? *x*::('*a*::*type*)*discr*. *range*(*S*) <<| *x*
⟨*proof*⟩

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

12.3 *undiscr*

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

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

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

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

end

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

theory *Lift*
imports *Discrete Up Cprod*
begin

defaultsort *type*

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

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

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

13.1 Lift as a datatype

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

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

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

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

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

lemma *Def-not-UU*: $Def\ a \neq UU$
 ⟨*proof*⟩

$\bot$ and *Def*

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

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

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

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

For $x \neq \bot$ in assumptions *def-tac* replaces x by *Def* a in conclusion.

⟨*ML*⟩

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

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

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

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

13.2 Lift is flat

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

instance *lift* :: (*type*) *flat*

⟨*proof*⟩

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

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

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

13.3 Further operations

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

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

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

13.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

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

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

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

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

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

rewrites for *flift1*, *flift2*

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

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

⟨*proof*⟩

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

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

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

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

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

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

⟨*ML*⟩

end

14 One: The unit domain

theory *One*
imports *Lift*
begin

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

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

Exhaustion and Elimination for type *one*

lemma *Exh-one*: $t = \bot \lor t = \mathit{ONE}$
⟨*proof*⟩

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

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

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

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

Case analysis function for type *one*

definition
  $one\text{-}when :: 'a::pcpo \rightarrow one \rightarrow 'a$ **where**
  $one\text{-}when = (\Lambda\ a.\ strictify\cdot(\Lambda\ \text{-}.\ a))$

translations
  $case\ x\ of\ CONST\ ONE \Rightarrow t == CONST\ one\text{-}when\cdot t\cdot x$
  $\Lambda\ (CONST\ ONE).\ t == CONST\ one\text{-}when\cdot t$

lemma *one-when1* [*simp*]: $(case\ \perp\ of\ ONE \Rightarrow t) = \perp$
⟨*proof*⟩

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

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

end

15 Tr: The type of lifted booleans

theory *Tr*
imports *Lift*
begin

defaultsort *pcpo*

types
  tr = *bool lift*

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

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

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

definition
trifte :: $'c \rightarrow 'c \rightarrow tr \rightarrow 'c$ **where**
ifte-def: $trifte = (\Lambda\ t\ e.\ FLIFT\ b.\ if\ b\ then\ t\ else\ e)$
abbreviation
cifte-syn :: $[tr, 'c, 'c] \Rightarrow 'c$ $((3If\ \text{-}/\ (then\ \text{-}/\ else\ \text{-})\ fi)\ 60)$ **where**
$If\ b\ then\ e1\ else\ e2\ fi == trifte \cdot e1 \cdot e2 \cdot b$

definition
trand :: $tr \rightarrow tr \rightarrow tr$ **where**
andalso-def: $trand = (\Lambda\ x\ y.\ If\ x\ then\ y\ else\ FF\ fi)$
abbreviation
andalso-syn :: $tr \Rightarrow tr \Rightarrow tr$ $(\text{-}\ andalso\ \text{-}\ [36,35]\ 35)$ **where**
$x\ andalso\ y == trand \cdot x \cdot y$

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

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

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

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

Exhaustion and Elimination for type *tr*

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

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

tactic for tr-thms with case split

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

distinctness for type *tr*

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

lemma *dist-eq-tr* [*simp*]:

$TT \neq \bot$ $FF \neq \bot$ $TT \neq FF$ $\bot \neq TT$ $\bot \neq FF$ $FF \neq TT$
⟨*proof*⟩

lemmas about andalso, orelse, neg and if

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

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

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

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

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

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

⟨*ML*⟩

15.1 Rewriting of HOLCF operations to HOL functions

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

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

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

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

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

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

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

15.2 Compactness

lemma *compact-TT* [*simp*]: *compact TT*
$\langle proof \rangle$

lemma *compact-FF* [*simp*]: *compact FF*
$\langle proof \rangle$

end

16 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun Cprod Adm*
begin

defaultsort *pcpo*

16.1 Iteration

consts
  $iterate :: nat \Rightarrow ('a::cpo \to 'a) \to ('a \to 'a)$

primrec
  $iterate\ 0 = (\Lambda\ F\ x.\ x)$
  $iterate\ (Suc\ n) = (\Lambda\ F\ x.\ F\cdot(iterate\ n\cdot F\cdot x))$

Derive inductive properties of iterate from primitive recursion

lemma *iterate-0* [*simp*]: $iterate\ 0\cdot F\cdot x = x$
$\langle proof \rangle$

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

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

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

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

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

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

16.2 Least fixed point operator

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

Binder syntax for *fix*

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

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

translations
μ x. t == *CONST fix*$\cdot$(Λ x. t)

Properties of *fix*

direct connection between *fix* and iteration

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

Kleene's fixed point theorems for continuous functions in pointed omega cpo's

lemma *fix-eq*: *fix*$\cdot F$ = $F \cdot$(*fix*$\cdot F$)
⟨*proof*⟩

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

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

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

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

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

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

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

strictness of *fix*

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

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

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

fix applied to identity and constant functions

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

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

16.3 Fixed point induction

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

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

16.4 Recursive let bindings

definition
$\mathit{CLetrec} :: ('a \to 'a \times 'b) \to 'b$ **where**
$\mathit{CLetrec} = (\Lambda\ F.\ \mathit{csnd}\cdot(F\cdot(\mu\ x.\ \mathit{cfst}\cdot(F\cdot x))))$

nonterminals

recbinds recbindt recbind

syntax
-*recbind* :: ['a, 'a] ⇒ *recbind* ((*2*- =/ -) *10*)
:: *recbind* ⇒ *recbindt* (-)
-*recbindt* :: [*recbind*, *recbindt*] ⇒ *recbindt* (-,/ -)
:: *recbindt* ⇒ *recbinds* (-)
-*recbinds* :: [*recbindt*, *recbinds*] ⇒ *recbinds* (-;/ -)
-*Letrec* :: [*recbinds*, 'a] ⇒ 'a ((*Letrec* (-)/ *in* (-)) *10*)

translations
(*recbindt*) $x = a$, $\langle y,ys \rangle = \langle b,bs \rangle$ == (*recbindt*) $\langle x,y,ys \rangle = \langle a,b,bs \rangle$
(*recbindt*) $x = a$, $y = b$ == (*recbindt*) $\langle x,y \rangle = \langle a,b \rangle$

translations
-*Letrec* (-*recbinds* *b bs*) *e* == -*Letrec* *b* (-*Letrec* *bs e*)
Letrec $xs = a$ *in* $\langle e,es \rangle$ == *CONST CLetrec*$\cdot(\Lambda\ xs.\ \langle a,e,es \rangle)$
Letrec $xs = a$ *in* e == *CONST CLetrec*$\cdot(\Lambda\ xs.\ \langle a,e \rangle)$

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

lemma *fix-cprod*:
$fix\cdot(F::'a \times 'b \to 'a \times 'b) =$
$\langle \mu\ x.\ cfst\cdot(F\cdot\langle x,\ \mu\ y.\ csnd\cdot(F\cdot\langle x,\ y\rangle)\rangle),$
$\mu\ y.\ csnd\cdot(F\cdot\langle \mu\ x.\ cfst\cdot(F\cdot\langle x,\ \mu\ y.\ csnd\cdot(F\cdot\langle x,\ y\rangle)\rangle),\ y\rangle)\rangle$
(**is** $fix\cdot F = \langle ?x,\ ?y\rangle$)
⟨*proof*⟩

16.5 Weak admissibility

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

an admissible formula is also weak admissible

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

computational induction for weak admissible formulae

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

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

end

17 Fixrec: Package for defining recursive functions in HOLCF

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

17.1 Maybe monad type

defaultsort *cpo*

pcpodef (**open**) $'a\ maybe = UNIV::(one\ {+}{+}\ 'a\ u)\ set$
$\langle proof\rangle$

constdefs
$fail :: 'a\ maybe$
$fail \equiv Abs\text{-}maybe\ (sinl \cdot ONE)$

constdefs
$return :: 'a \rightarrow 'a\ maybe$ **where**
$return \equiv \Lambda\ x.\ Abs\text{-}maybe\ (sinr \cdot (up \cdot x))$

definition
$maybe\text{-}when :: 'b \rightarrow ('a \rightarrow 'b) \rightarrow 'a\ maybe \rightarrow 'b::pcpo$ **where**
$maybe\text{-}when = (\Lambda\ f\ r\ m.\ sscase \cdot (\Lambda\ x.\ f) \cdot (fup \cdot r) \cdot (Rep\text{-}maybe\ m))$

lemma *maybeE*:
$[\![p = \perp \Longrightarrow Q;\ p = fail \Longrightarrow Q;\ \bigwedge x.\ p = return \cdot x \Longrightarrow Q]\!] \Longrightarrow Q$
$\langle proof\rangle$

lemma *return-defined* [*simp*]: $return \cdot x \neq \perp$
$\langle proof\rangle$

lemma *fail-defined* [*simp*]: $fail \neq \perp$
$\langle proof\rangle$

lemma *return-eq* [*simp*]: $(return \cdot x = return \cdot y) = (x = y)$
$\langle proof\rangle$

lemma *return-neq-fail* [*simp*]:
$return \cdot x \neq fail\ fail \neq return \cdot x$
$\langle proof\rangle$

lemma *maybe-when-rews* [*simp*]:
$maybe\text{-}when \cdot f \cdot r \cdot \perp = \perp$
$maybe\text{-}when \cdot f \cdot r \cdot fail = f$
$maybe\text{-}when \cdot f \cdot r \cdot (return \cdot x) = r \cdot x$
$\langle proof\rangle$

translations
$case\ m\ of\ fail \Rightarrow t1 \mid return{\cdot}x \Rightarrow t2$ == $CONST\ maybe\text{-}when{\cdot}t1{\cdot}(\Lambda\ x.\ t2){\cdot}m$

17.1.1 Monadic bind operator

definition
$bind :: 'a\ maybe \to ('a \to 'b\ maybe) \to 'b\ maybe$ **where**
$bind = (\Lambda\ m\ f.\ case\ m\ of\ fail \Rightarrow fail \mid return{\cdot}x \Rightarrow f{\cdot}x)$

monad laws

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

lemma *bind-fail* [*simp*]: $bind{\cdot}fail{\cdot}f = fail$
⟨*proof*⟩

lemma *left-unit* [*simp*]: $bind{\cdot}(return{\cdot}a){\cdot}k = k{\cdot}a$
⟨*proof*⟩

lemma *right-unit* [*simp*]: $bind{\cdot}m{\cdot}return = m$
⟨*proof*⟩

lemma *bind-assoc*:
$bind{\cdot}(bind{\cdot}m{\cdot}k){\cdot}h = bind{\cdot}m{\cdot}(\Lambda\ a.\ bind{\cdot}(k{\cdot}a){\cdot}h)$
⟨*proof*⟩

17.1.2 Run operator

definition
$run{::}\ 'a\ maybe \to 'a{::}pcpo$ **where**
$run = maybe\text{-}when{\cdot}\bot{\cdot}ID$

rewrite rules for run

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

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

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

17.1.3 Monad plus operator

definition
$mplus :: 'a\ maybe \to 'a\ maybe \to 'a\ maybe$ **where**
$mplus = (\Lambda\ m1\ m2.\ case\ m1\ of\ fail \Rightarrow m2 \mid return{\cdot}x \Rightarrow m1)$

abbreviation
mplus-syn :: $['a\ maybe,\ 'a\ maybe] \Rightarrow 'a\ maybe$ (**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-return* [*simp*]: $return{\cdot}x +\!+\!+ m = return{\cdot}x$
⟨*proof*⟩

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

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

17.1.4 Fatbar combinator

definition
fatbar :: $('a \to 'b\ maybe) \to ('a \to 'b\ maybe) \to ('a \to 'b\ maybe)$ **where**
$fatbar = (\Lambda\ a\ b\ x.\ a{\cdot}x +\!+\!+ b{\cdot}x)$

abbreviation
fatbar-syn :: $['a \to 'b\ maybe,\ 'a \to 'b\ maybe] \Rightarrow 'a \to 'b\ maybe$ (**infixr** $\|$ *60*)
where
$m1 \parallel m2 == fatbar{\cdot}m1{\cdot}m2$

lemma *fatbar1*: $m{\cdot}x = \bot \Longrightarrow (m \parallel ms){\cdot}x = \bot$
⟨*proof*⟩

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

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

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

lemma *run-fatbar1*: $m{\cdot}x = \bot \Longrightarrow run{\cdot}((m \parallel ms){\cdot}x) = \bot$
⟨*proof*⟩

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

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

⟨*proof*⟩

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

17.2 Case branch combinator

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

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

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

17.3 Case syntax

nonterminals
Case-syn Cases-syn

syntax
-Case-syntax:: ['a, *Cases-syn*] => 'b ((*Case* - *of*/ -) *10*)
-Case1 :: ['a, 'b] => *Case-syn* ((*2*- =>/ -) *10*)
:: *Case-syn* => *Cases-syn* (-)
-Case2 :: [*Case-syn*, *Cases-syn*] => *Cases-syn* (-/ | -)

syntax (*xsymbols*)
-Case1 :: ['a, 'b] => *Case-syn* ((*2*- ⇒/ -) *10*)

translations
-Case-syntax x ms == *CONST Fixrec.run*·(*ms*·*x*)
-Case2 m ms == *m* ∥ *ms*

Parsing Case expressions

syntax
-pat :: 'a
-var :: 'a

translations
-Case1 p r => *XCONST branch* (*-pat p*)·(*-var p r*)
-var (*-args x y*) *r* => *XCONST csplit*·(*-var x* (*-var y r*))
-var () *r* => *XCONST unit-when*·*r*

⟨*ML*⟩

Printing Case expressions

syntax
-match :: $'a$

⟨*ML*⟩

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

17.4 Pattern combinators for data constructors

types $('a, 'b)$ *pat* = $'a \to 'b$ *maybe*

definition
cpair-pat :: $('a, 'c)$ *pat* ⇒ $('b, 'd)$ *pat* ⇒ $('a \times 'b, 'c \times 'd)$ *pat* **where**
cpair-pat p1 p2 = (Λ⟨*x*, *y*⟩.
bind·(*p1*·*x*)·(Λ *a*. *bind*·(*p2*·*y*)·(Λ *b*. *return*·⟨*a*, *b*⟩)))

definition
spair-pat ::
$('a, 'c)$ *pat* ⇒ $('b, 'd)$ *pat* ⇒ $('a::pcpo \otimes 'b::pcpo, 'c \times 'd)$ *pat* **where**
spair-pat p1 p2 = (Λ(:*x*, *y*:). *cpair-pat p1 p2*·⟨*x*, *y*⟩)

definition
sinl-pat :: $('a, 'c)$ *pat* ⇒ $('a::pcpo \oplus 'b::pcpo, 'c)$ *pat* **where**
sinl-pat p = *sscase*·*p*·(Λ *x*. *fail*)

definition
sinr-pat :: $('b, 'c)$ *pat* ⇒ $('a::pcpo \oplus 'b::pcpo, 'c)$ *pat* **where**
sinr-pat p = *sscase*·(Λ *x*. *fail*)·*p*

definition
up-pat :: $('a, 'b)$ *pat* ⇒ $('a\ u, 'b)$ *pat* **where**
up-pat p = *fup*·*p*

definition
TT-pat :: (*tr*, *unit*) *pat* **where**
TT-pat = (Λ *b*. *If b then return*·() *else fail fi*)

definition
FF-pat :: (*tr*, *unit*) *pat* **where**
FF-pat = (Λ *b*. *If b then fail else return*·() *fi*)

definition
ONE-pat :: (*one*, *unit*) *pat* **where**
ONE-pat = (Λ *ONE*. *return*·())

Parse translations (patterns)

translations
-pat (*XCONST cpair*·*x*·*y*) => *XCONST cpair-pat* (*-pat x*) (*-pat y*)

-pat (XCONST spair·x·y) => XCONST spair-pat (-pat x) (-pat y)
-pat (XCONST sinl·x) => XCONST sinl-pat (-pat x)
-pat (XCONST sinr·x) => XCONST sinr-pat (-pat x)
-pat (XCONST up·x) => XCONST up-pat (-pat x)
-pat (XCONST TT) => XCONST TT-pat
-pat (XCONST FF) => XCONST FF-pat
-pat (XCONST ONE) => XCONST ONE-pat

Parse translations (variables)

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

Print translations

translations
CONST cpair·(-match p1 v1)·(-match p2 v2)
  <= -match (CONST cpair-pat p1 p2) (-args v1 v2)
CONST spair·(-match p1 v1)·(-match p2 v2)
  <= -match (CONST spair-pat p1 p2) (-args v1 v2)
CONST sinl·(-match p1 v1) <= -match (CONST sinl-pat p1) v1
CONST sinr·(-match p1 v1) <= -match (CONST sinr-pat p1) v1
CONST up·(-match p1 v1) <= -match (CONST up-pat p1) v1
CONST TT <= -match (CONST TT-pat) ()
CONST FF <= -match (CONST FF-pat) ()
CONST ONE <= -match (CONST ONE-pat) ()

lemma *cpair-pat1*:
branch p·r·x = ⊥ ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = ⊥
⟨proof⟩

lemma *cpair-pat2*:
branch p·r·x = fail ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = fail
⟨proof⟩

lemma *cpair-pat3*:
branch p·r·x = return·s ⟹
  branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = branch q·s·y
⟨proof⟩

lemmas *cpair-pat* [*simp*] =
cpair-pat1 cpair-pat2 cpair-pat3

lemma *spair-pat* [*simp*]:

$branch\ (spair\text{-}pat\ p1\ p2)\cdot r\cdot\bot = \bot$
$[\![x \neq \bot;\ y \neq \bot]\!]$
$\Longrightarrow branch\ (spair\text{-}pat\ p1\ p2)\cdot r\cdot(:x,\ y:) =$
$branch\ (cpair\text{-}pat\ p1\ p2)\cdot r\cdot\langle x,\ y\rangle$
⟨*proof*⟩

lemma *sinl-pat* [*simp*]:
$branch\ (sinl\text{-}pat\ p)\cdot r\cdot\bot = \bot$
$x \neq \bot \Longrightarrow branch\ (sinl\text{-}pat\ p)\cdot r\cdot(sinl\cdot x) = branch\ p\cdot r\cdot x$
$y \neq \bot \Longrightarrow branch\ (sinl\text{-}pat\ p)\cdot r\cdot(sinr\cdot y) = fail$
⟨*proof*⟩

lemma *sinr-pat* [*simp*]:
$branch\ (sinr\text{-}pat\ p)\cdot r\cdot\bot = \bot$
$x \neq \bot \Longrightarrow branch\ (sinr\text{-}pat\ p)\cdot r\cdot(sinl\cdot x) = fail$
$y \neq \bot \Longrightarrow branch\ (sinr\text{-}pat\ p)\cdot r\cdot(sinr\cdot y) = branch\ p\cdot r\cdot y$
⟨*proof*⟩

lemma *up-pat* [*simp*]:
$branch\ (up\text{-}pat\ p)\cdot r\cdot\bot = \bot$
$branch\ (up\text{-}pat\ p)\cdot r\cdot(up\cdot x) = branch\ p\cdot r\cdot x$
⟨*proof*⟩

lemma *TT-pat* [*simp*]:
$branch\ TT\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot\bot = \bot$
$branch\ TT\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot TT = return\cdot r$
$branch\ TT\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot FF = fail$
⟨*proof*⟩

lemma *FF-pat* [*simp*]:
$branch\ FF\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot\bot = \bot$
$branch\ FF\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot TT = fail$
$branch\ FF\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot FF = return\cdot r$
⟨*proof*⟩

lemma *ONE-pat* [*simp*]:
$branch\ ONE\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot\bot = \bot$
$branch\ ONE\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot ONE = return\cdot r$
⟨*proof*⟩

17.5 Wildcards, as-patterns, and lazy patterns

syntax
$\text{-}as\text{-}pat :: [idt,\ 'a] \Rightarrow 'a$ (**infixr** *as* *10*)
$\text{-}lazy\text{-}pat :: 'a \Rightarrow 'a\ (\sim\ \text{-}\ [1000]\ 1000)$

definition
$wild\text{-}pat :: 'a \rightarrow unit\ maybe$ **where**
$wild\text{-}pat = (\Lambda\ x.\ return\cdot())$

definition
as-pat :: $('a \to 'b\ maybe) \Rightarrow 'a \to ('a \times 'b)\ maybe$ **where**
$as\text{-}pat\ p = (\Lambda\ x.\ bind\cdot(p\cdot x)\cdot(\Lambda\ a.\ return\cdot\langle x,\ a\rangle))$

definition
lazy-pat :: $('a \to 'b{::}pcpo\ maybe) \Rightarrow ('a \to 'b\ maybe)$ **where**
$lazy\text{-}pat\ p = (\Lambda\ x.\ return\cdot(run\cdot(p\cdot x)))$

Parse translations (patterns)

translations
-pat - => XCONST wild-pat
-pat (-as-pat x y) => XCONST as-pat (-pat y)
-pat (-lazy-pat x) => XCONST lazy-pat (-pat x)

Parse translations (variables)

translations
-var - r => -var () r
-var (-as-pat x y) r => -var (-args x y) r
-var (-lazy-pat x) r => -var x r

Print translations

translations
- <= -match (CONST wild-pat) ()
-as-pat x (-match p v) <= -match (CONST as-pat p) (-args (-var x) v)
-lazy-pat (-match p v) <= -match (CONST lazy-pat p) v

Lazy patterns in lambda abstractions

translations
-cabs (-lazy-pat p) r == CONST Fixrec.run oo (-Case1 (-lazy-pat p) r)

lemma *wild-pat* [*simp*]: $branch\ wild\text{-}pat\cdot(unit\text{-}when\cdot r)\cdot x = return\cdot r$
⟨*proof*⟩

lemma *as-pat* [*simp*]:
$branch\ (as\text{-}pat\ p)\cdot(csplit\cdot r)\cdot x = branch\ p\cdot(r\cdot x)\cdot x$
⟨*proof*⟩

lemma *lazy-pat* [*simp*]:
$branch\ p\cdot r\cdot x = \bot \Longrightarrow branch\ (lazy\text{-}pat\ p)\cdot r\cdot x = return\cdot(r\cdot\bot)$
$branch\ p\cdot r\cdot x = fail \Longrightarrow branch\ (lazy\text{-}pat\ p)\cdot r\cdot x = return\cdot(r\cdot\bot)$
$branch\ p\cdot r\cdot x = return\cdot s \Longrightarrow branch\ (lazy\text{-}pat\ p)\cdot r\cdot x = return\cdot s$
⟨*proof*⟩

17.6 Match functions for built-in types

defaultsort *pcpo*

definition

$match\text{-}UU :: 'a \rightarrow unit\ maybe$ **where**
$match\text{-}UU = (\Lambda\ x.\ fail)$

definition
$match\text{-}cpair :: 'a{::}cpo \times 'b{::}cpo \rightarrow ('a \times 'b)\ maybe$ **where**
$match\text{-}cpair = csplit\cdot(\Lambda\ x\ y.\ return\cdot{<}x,y{>})$

definition
$match\text{-}spair :: 'a \otimes 'b \rightarrow ('a \times 'b)\ maybe$ **where**
$match\text{-}spair = ssplit\cdot(\Lambda\ x\ y.\ return\cdot{<}x,y{>})$

definition
$match\text{-}sinl :: 'a \oplus 'b \rightarrow 'a\ maybe$ **where**
$match\text{-}sinl = sscase\cdot return\cdot(\Lambda\ y.\ fail)$

definition
$match\text{-}sinr :: 'a \oplus 'b \rightarrow 'b\ maybe$ **where**
$match\text{-}sinr = sscase\cdot(\Lambda\ x.\ fail)\cdot return$

definition
$match\text{-}up :: 'a{::}cpo\ u \rightarrow 'a\ maybe$ **where**
$match\text{-}up = fup\cdot return$

definition
$match\text{-}ONE :: one \rightarrow unit\ maybe$ **where**
$match\text{-}ONE = (\Lambda\ ONE.\ return\cdot())$

definition
$match\text{-}TT :: tr \rightarrow unit\ maybe$ **where**
$match\text{-}TT = (\Lambda\ b.\ If\ b\ then\ return\cdot()\ else\ fail\ fi)$

definition
$match\text{-}FF :: tr \rightarrow unit\ maybe$ **where**
$match\text{-}FF = (\Lambda\ b.\ If\ b\ then\ fail\ else\ return\cdot()\ fi)$

lemma *match-UU-simps* [*simp*]:
$match\text{-}UU\cdot x = fail$
$\langle proof \rangle$

lemma *match-cpair-simps* [*simp*]:
$match\text{-}cpair\cdot{<}x,y{>} = return\cdot{<}x,y{>}$
$\langle proof \rangle$

lemma *match-spair-simps* [*simp*]:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow match\text{-}spair\cdot(:x,y:) = return\cdot{<}x,y{>}$
$match\text{-}spair\cdot\bot = \bot$
$\langle proof \rangle$

lemma *match-sinl-simps* [*simp*]:

$x \neq \bot \Longrightarrow$ *match-sinl*$\cdot$(*sinl*$\cdot x$) = *return*$\cdot x$
$x \neq \bot \Longrightarrow$ *match-sinl*$\cdot$(*sinr*$\cdot x$) = *fail*
match-sinl$\cdot\bot = \bot$
⟨*proof*⟩

lemma *match-sinr-simps* [*simp*]:
$x \neq \bot \Longrightarrow$ *match-sinr*$\cdot$(*sinr*$\cdot x$) = *return*$\cdot x$
$x \neq \bot \Longrightarrow$ *match-sinr*$\cdot$(*sinl*$\cdot x$) = *fail*
match-sinr$\cdot\bot = \bot$
⟨*proof*⟩

lemma *match-up-simps* [*simp*]:
match-up$\cdot$(*up*$\cdot x$) = *return*$\cdot x$
match-up$\cdot\bot = \bot$
⟨*proof*⟩

lemma *match-ONE-simps* [*simp*]:
match-ONE$\cdot$*ONE* = *return*$\cdot$()
match-ONE$\cdot\bot = \bot$
⟨*proof*⟩

lemma *match-TT-simps* [*simp*]:
match-TT$\cdot$*TT* = *return*$\cdot$()
match-TT$\cdot$*FF* = *fail*
match-TT$\cdot\bot = \bot$
⟨*proof*⟩

lemma *match-FF-simps* [*simp*]:
match-FF$\cdot$*FF* = *return*$\cdot$()
match-FF$\cdot$*TT* = *fail*
match-FF$\cdot\bot = \bot$
⟨*proof*⟩

17.7 Mutual recursion

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

lemma *cpair-equalI*: $[\![x \equiv$ *cfst*$\cdot p$; $y \equiv$ *csnd*$\cdot p]\!] \Longrightarrow \langle x,y\rangle \equiv p$
⟨*proof*⟩

lemma *cpair-eqD1*: $<x,y> = <x',y'> \Longrightarrow x = x'$
⟨*proof*⟩

lemma *cpair-eqD2*: $<x,y> = <x',y'> \Longrightarrow y = y'$
⟨*proof*⟩

lemma for proving rewrite rules

lemma *ssubst-lhs*: $[\![t = s;\ P\ s = Q]\!] \Longrightarrow P\ t = Q$
⟨*proof*⟩

17.8 Initializing the fixrec package

⟨*ML*⟩

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

end

18 Domain: Domain package

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

defaultsort *pcpo*

18.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
  fixes *abs* :: $'a \to 'b$
  fixes *rep* :: $'b \to 'a$
  assumes *abs-iso* [*simp*]: $rep{\cdot}(abs{\cdot}x) = x$
  assumes *rep-iso* [*simp*]: $abs{\cdot}(rep{\cdot}y) = y$
begin

lemma *swap*: *iso rep abs*
  ⟨*proof*⟩

lemma *abs-less*: $(abs{\cdot}x \sqsubseteq abs{\cdot}y) = (x \sqsubseteq y)$
⟨*proof*⟩

lemma *rep-less*: $(rep{\cdot}x \sqsubseteq rep{\cdot}y) = (x \sqsubseteq y)$
  ⟨*proof*⟩

lemma *abs-eq*: $(abs{\cdot}x = abs{\cdot}y) = (x = y)$
  ⟨*proof*⟩

lemma *rep-eq*: $(rep{\cdot}x = rep{\cdot}y) = (x = y)$
  ⟨*proof*⟩

lemma *abs-strict*: $abs{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *rep-strict*: $rep{\cdot}\bot = \bot$
  ⟨*proof*⟩

lemma *abs-defin'*: $abs{\cdot}x = \bot \Longrightarrow x = \bot$

⟨*proof*⟩

lemma *rep-defin'*: $rep{\cdot}z = \bot \Longrightarrow z = \bot$
⟨*proof*⟩

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

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

lemma *abs-defined-iff*: $(abs{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

lemma *rep-defined-iff*: $(rep{\cdot}x = \bot) = (x = \bot)$
⟨*proof*⟩

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

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

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

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

lemma *iso-swap*: $(x = abs{\cdot}y) = (rep{\cdot}x = y)$
⟨*proof*⟩

end

18.2 Casedist

lemma *ex-one-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) = P\ ONE$
⟨*proof*⟩

lemma *ex-up-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) = (\exists\, x.\ P\ (up{\cdot}x))$
⟨*proof*⟩

lemma *ex-sprod-defined-iff*:
$(\exists\, y.\ P\ y \wedge y \neq \bot) =$
$(\exists\, x\ y.\ (P\ (:x,\ y:) \wedge x \neq \bot) \wedge y \neq \bot)$
⟨*proof*⟩

lemma *ex-sprod-up-defined-iff*:
$(\exists\, y.\ P\ y \wedge y \neq \bot) =$
$(\exists\, x\ y.\ P\ (:up{\cdot}x,\ y:) \wedge y \neq \bot)$
$\langle proof \rangle$

lemma *ex-ssum-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) =$
$((\exists\, x.\ P\ (sinl{\cdot}x) \wedge x \neq \bot) \vee$
$(\exists\, x.\ P\ (sinr{\cdot}x) \wedge x \neq \bot))$
$\langle proof \rangle$

lemma *exh-start*: $p = \bot \vee (\exists\, x.\ p = x \wedge x \neq \bot)$
$\langle proof \rangle$

lemmas *ex-defined-iffs* =
ex-ssum-defined-iff
ex-sprod-up-defined-iff
ex-sprod-defined-iff
ex-up-defined-iff
ex-one-defined-iff

Rules for turning exh into casedist

lemma *exh-casedist0*: $[\![R;\ R \Longrightarrow P]\!] \Longrightarrow P$
$\langle proof \rangle$

lemma *exh-casedist1*: $((P \vee Q \Longrightarrow R) \Longrightarrow S) \equiv ([\![P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow S)$
$\langle proof \rangle$

lemma *exh-casedist2*: $(\exists\, x.\ P\ x \Longrightarrow Q) \equiv (\bigwedge x.\ P\ x \Longrightarrow Q)$
$\langle proof \rangle$

lemma *exh-casedist3*: $(P \wedge Q \Longrightarrow R) \equiv (P \Longrightarrow Q \Longrightarrow R)$
$\langle proof \rangle$

lemmas *exh-casedists* = *exh-casedist1 exh-casedist2 exh-casedist3*

18.3 Setting up the package

$\langle ML \rangle$

end

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

Tools/domain/domain-library.ML
Tools/domain/domain-syntax.ML
Tools/domain/domain-axioms.ML
Tools/domain/domain-theorems.ML
Tools/domain/domain-extender.ML
Tools/adm-tac.ML

begin

⟨*ML*⟩

end