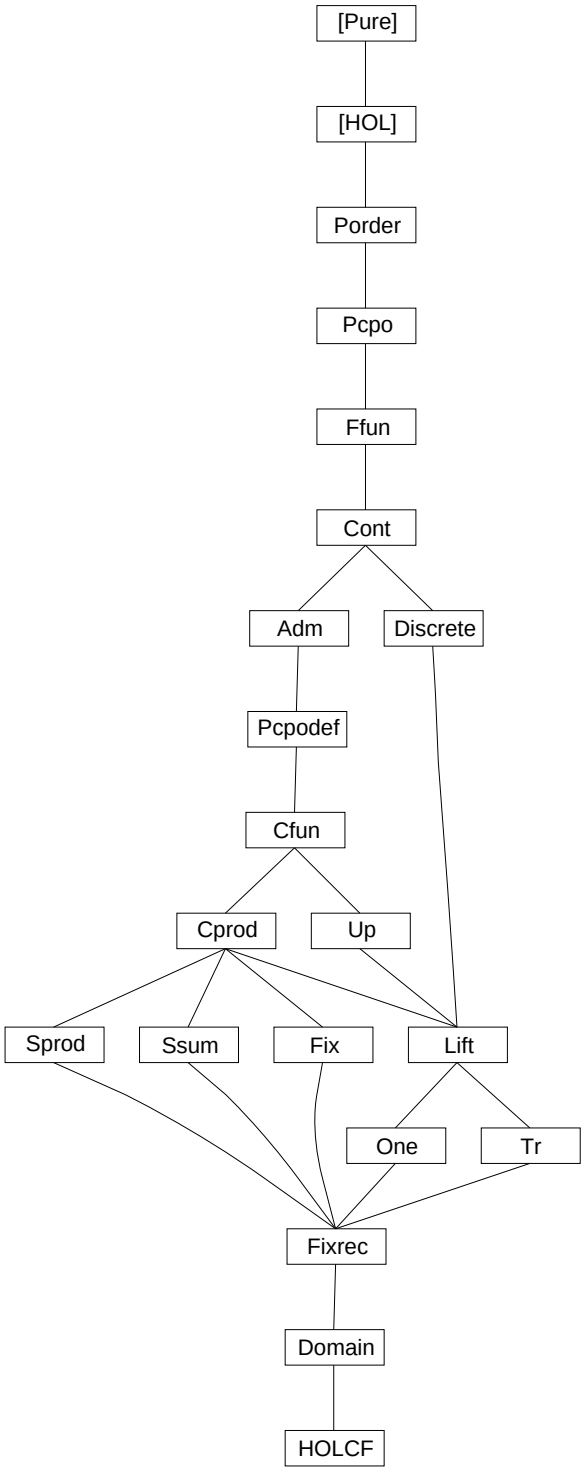
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

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*] : $\forall x::'a::po.\ uu \sqsubseteq x \Longrightarrow uu = (THE\ u.\ \forall y.\ u \sqsubseteq y)$
by (*blast intro*: *theI2 antisym-less*)

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

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

lemma *box-less*: $[\![(a::'a::po) \sqsubseteq b; c \sqsubseteq a; b \sqsubseteq d]\!] \Longrightarrow c \sqsubseteq d$
by (*rule trans-less* [*OF trans-less*])

lemma *po-eq-conv*: $((x::'a::po) = y) = (x \sqsubseteq y \wedge y \sqsubseteq x)$
by (*fast elim*!: *antisym-less-inverse intro*!: *antisym-less*)

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

lemma *sq-ord-less-eq-trans*: $[\![a \sqsubseteq b; b = c]\!] \Longrightarrow a \sqsubseteq c$
by (*rule subst*)

lemma *sq-ord-eq-less-trans*: $[\![a = b; b \sqsubseteq c]\!] \Longrightarrow a \sqsubseteq c$
by (*rule ssubst*)

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

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

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

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

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

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

definition
lub :: *'a set* ⇒ *'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** ⨆ *10*)

lubs are unique

lemma *unique-lub*: ⟦*S* <<| *x*; *S* <<| *y*⟧ ⟹ *x* = *y*
apply (*unfold is-lub-def is-ub-def*)
apply (*blast intro*: *antisym-less*)
done

chains are monotone functions

lemma *chain-mono* [*rule-format*]: *chain F* $\Longrightarrow$ *x* < *y* $\longrightarrow$ *F x* $\sqsubseteq$ *F y*
apply (*unfold chain-def*)
apply (*induct-tac y*)
apply *simp*
apply (*blast elim*: *less-SucE intro*: *trans-less*)
done

lemma *chain-mono3*: ⟦*chain F*; *x* ≤ *y*⟧ $\Longrightarrow$ *F x* $\sqsubseteq$ *F y*
apply (*drule le-imp-less-or-eq*)
apply (*blast intro*: *chain-mono*)
done

The range of a chain is a totally ordered

lemma *chain-tord*: *chain F* $\Longrightarrow$ *tord* (*range F*)
apply (*unfold tord-def*, *clarify*)
apply (*rule nat-less-cases*)
apply (*fast intro*: *chain-mono*)+
done

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

lemma *lubI*: *M* <<| *x* $\Longrightarrow$ *M* <<| *lub M*
apply (*unfold lub-def*)
apply (*rule theI*)
apply *assumption*
apply (*erule* (*1*) *unique-lub*)
done

lemma *thelubI*: *M* <<| *l* $\Longrightarrow$ *lub M* = *l*
by (*rule unique-lub* [*OF lubI*])

lemma *lub-singleton* [*simp*]: *lub* {*x*} = *x*
by (*simp add*: *thelubI is-lub-def is-ub-def*)

access to some definition as inference rule

lemma *is-lubD1*: *S* <<| *x* $\Longrightarrow$ *S* <| *x*
by (*unfold is-lub-def*, *simp*)

lemma *is-lub-lub*: ⟦*S* <<| *x*; *S* <| *u*⟧ $\Longrightarrow$ *x* $\sqsubseteq$ *u*
by (*unfold is-lub-def*, *simp*)

lemma *is-lubI*: ⟦*S* <| *x*; $\bigwedge$*u*. *S* <| *u* $\Longrightarrow$ *x* $\sqsubseteq$ *u*⟧ $\Longrightarrow$ *S* <<| *x*
by (*unfold is-lub-def*, *fast*)

lemma *chainE*: *chain F* $\Longrightarrow$ *F i* $\sqsubseteq$ *F* (*Suc i*)
by (*unfold chain-def*, *simp*)

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

by (*unfold chain-def*, *simp*)

lemma *chain-shift*: *chain Y* $\Longrightarrow$ *chain* ($\lambda i.$ *Y* (*i* + *j*))
apply (*rule chainI*)
apply *simp*
apply (*erule chainE*)
done

technical lemmas about (least) upper bounds of chains

lemma *ub-rangeD*: *range S* <| *x* $\Longrightarrow$ *S i* $\sqsubseteq$ *x*
by (*unfold is-ub-def*, *simp*)

lemma *ub-rangeI*: ($\bigwedge i.$ *S i* $\sqsubseteq$ *x*) $\Longrightarrow$ *range S* <| *x*
by (*unfold is-ub-def*, *fast*)

lemma *is-ub-lub*: *range S* <<| *x* $\Longrightarrow$ *S i* $\sqsubseteq$ *x*
by (*rule is-lubD1* [*THEN ub-rangeD*])

lemma *is-ub-range-shift*:
chain S $\Longrightarrow$ *range* ($\lambda i.$ *S* (*i* + *j*)) <| *x* = *range S* <| *x*
apply (*rule iffI*)
apply (*rule ub-rangeI*)
apply (*rule-tac y*=*S* (*i* + *j*) **in** *trans-less*)
apply (*erule chain-mono3*)
apply (*rule le-add1*)
apply (*erule ub-rangeD*)
apply (*rule ub-rangeI*)
apply (*erule ub-rangeD*)
done

lemma *is-lub-range-shift*:
chain S $\Longrightarrow$ *range* ($\lambda i.$ *S* (*i* + *j*)) <<| *x* = *range S* <<| *x*
by (*simp add*: *is-lub-def is-ub-range-shift*)

results about finite chains

lemma *lub-finch1*: $[\![$*chain C*; *max-in-chain i C*$]\!]$ $\Longrightarrow$ *range C* <<| *C i*
apply (*unfold max-in-chain-def*)
apply (*rule is-lubI*)
apply (*rule ub-rangeI*, *rename-tac j*)
apply (*rule-tac x*=*i* **and** *y*=*j* **in** *linorder-le-cases*)
apply *simp*
apply (*erule* (*1*) *chain-mono3*)
apply (*erule ub-rangeD*)
done

lemma *lub-finch2*:
finite-chain C $\Longrightarrow$ *range C* <<| *C* (*LEAST i. max-in-chain i C*)
apply (*unfold finite-chain-def*)
apply (*erule conjE*)

apply (*erule LeastI2-ex*)
apply (*erule* (*1*) *lub-finch1*)
done

lemma *finch-imp-finite-range*: *finite-chain Y* $\Longrightarrow$ *finite* (*range Y*)
apply (*unfold finite-chain-def*, *clarify*)
apply (*rule-tac f*=*Y* **and** *n*=*Suc i* **in** *nat-seg-image-imp-finite*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule rangeE*, *rename-tac j*)
apply (*rule-tac x*=*i* **and** *y*=*j* **in** *linorder-le-cases*)
apply (*subgoal-tac Y j* = *Y i*, *simp*)
apply (*simp add*: *max-in-chain-def*)
apply *simp*
apply *fast*
done

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*)
apply (*erule finite-induct*, *simp*)
apply (*rename-tac a S*, *clarify*)
apply (*case-tac S* = {}, *simp*)
apply (*drule* (*1*) *mp*)
apply (*drule mp*, *simp add*: *tord-def*)
apply (*erule bexE*, *rename-tac z*)
apply (*subgoal-tac a* $\sqsubseteq$ *z* $\vee$ *z* $\sqsubseteq$ *a*)
apply (*erule disjE*)
apply (*rule-tac x*=*z* **in** *bexI*, *simp*, *simp*)
apply (*rule-tac x*=*a* **in** *bexI*)
apply (*clarsimp elim*!: *rev-trans-less*)
apply *simp*
apply (*simp add*: *tord-def*)
done

lemma *finite-range-imp-finch*:
$[\![$*chain Y*; *finite* (*range Y*)$]\!]$ $\Longrightarrow$ *finite-chain Y*
apply (*subgoal-tac* $\exists$ *y*$\in$*range Y*. $\forall$ *x*$\in$*range Y*. *x* $\sqsubseteq$ *y*)
apply (*clarsimp*, *rename-tac i*)
apply (*subgoal-tac max-in-chain i Y*)
apply (*simp add*: *finite-chain-def exI*)
apply (*simp add*: *max-in-chain-def po-eq-conv chain-mono3*)
apply (*erule finite-tord-has-max*, *simp*)
apply (*erule chain-tord*)
done

lemma *bin-chain*: *x* $\sqsubseteq$ *y* $\Longrightarrow$ *chain* (λ*i*. *if i=0 then x else y*)
by (*rule chainI*, *simp*)

lemma *bin-chainmax*:

$x \sqsubseteq y \Longrightarrow$ *max-in-chain* (*Suc 0*) ($\lambda i.$ *if i=0 then x else y*)
by (*unfold max-in-chain-def*, *simp*)

lemma *lub-bin-chain*:
$x \sqsubseteq y \Longrightarrow$ *range* (λi::*nat*. *if i=0 then x else y*) <<| *y*
apply (*frule bin-chain*)
apply (*drule bin-chainmax*)
apply (*drule* (*1*) *lub-finch1*)
apply *simp*
done

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*
by (*blast dest*: *ub-rangeD intro*: *thelubI is-lubI ub-rangeI*)

the lub of a constant chain is the constant

lemma *chain-const* [*simp*]: *chain* ($\lambda i.$ *c*)
by (*simp add*: *chainI*)

lemma *lub-const*: *range* ($\lambda x.$ *c*) <<| *c*
by (*blast dest*: *ub-rangeD intro*: *is-lubI ub-rangeI*)

lemma *thelub-const* [*simp*]: ($\bigsqcup i.$ *c*) = *c*
by (*rule lub-const* [*THEN thelubI*])

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*
by (*blast dest*: *cpo intro*: *lubI*)

Properties of the lub

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

by (*blast dest*: *cpo intro*: *lubI* [*THEN is-ub-lub*])

lemma *is-lub-thelub*:
⟦*chain* (*S*::*nat* ⇒ '*a*::*cpo*); *range S* <| *x*⟧ ⟹ (⨆ *i*. *S i*) ⊑ *x*
by (*blast dest*: *cpo intro*: *lubI* [*THEN is-lub-lub*])

lemma *lub-range-mono*:
⟦*range X* ⊆ *range Y*; *chain Y*; *chain* (*X*::*nat* ⇒ '*a*::*cpo*)⟧
⟹ (⨆ *i*. *X i*) ⊑ (⨆ *i*. *Y i*)
apply (*erule is-lub-thelub*)
apply (*rule ub-rangeI*)
apply (*subgoal-tac* ∃ *j*. *X i* = *Y j*)
apply *clarsimp*
apply (*erule is-ub-thelub*)
apply *auto*
done

lemma *lub-range-shift*:
chain (*Y*::*nat* ⇒ '*a*::*cpo*) ⟹ (⨆ *i*. *Y* (*i* + *j*)) = (⨆ *i*. *Y i*)
apply (*rule antisym-less*)
apply (*rule lub-range-mono*)
apply *fast*
apply *assumption*
apply (*erule chain-shift*)
apply (*rule is-lub-thelub*)
apply *assumption*
apply (*rule ub-rangeI*)
apply (*rule-tac y*=*Y* (*i* + *j*) **in** *trans-less*)
apply (*erule chain-mono3*)
apply (*rule le-add1*)
apply (*rule is-ub-thelub*)
apply (*erule chain-shift*)
done

lemma *maxinch-is-thelub*:
chain Y ⟹ *max-in-chain i Y* = ((⨆ *i*. *Y i*) = ((*Y i*)::'*a*::*cpo*))
apply (*rule iffI*)
apply (*fast intro*!: *thelubI lub-finch1*)
apply (*unfold max-in-chain-def*)
apply (*safe intro*!: *antisym-less*)
apply (*fast elim*!: *chain-mono3*)
apply (*drule sym*)
apply (*force elim*!: *is-ub-thelub*)
done

the ⊑ relation between two chains is preserved by their lubs

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

apply (*erule is-lub-thelub*)
apply (*rule ub-rangeI*)
apply (*rule trans-less*)
apply (*erule spec*)
apply (*erule is-ub-thelub*)
done

the = relation between two chains is preserved by their lubs

lemma *lub-equal*:
  ⟦*chain* (*X*::*nat* ⇒ 'a::*cpo*); *chain Y*; ∀ *k*. *X k* = *Y k*⟧
    ⟹ (⨆ *i*. *X i*) = (⨆ *i*. *Y i*)
by (*simp only*: *expand-fun-eq* [*symmetric*])

more results about mono and = of lubs of chains

lemma *lub-mono2*:
  ⟦∃ *j*. ∀ *i*>*j*. *X i* = *Y i*; *chain* (*X*::*nat* ⇒ 'a::*cpo*); *chain Y*⟧
    ⟹ (⨆ *i*. *X i*) ⊑ (⨆ *i*. *Y i*)
apply (*erule exE*)
apply (*subgoal-tac* (⨆ *i*. *X* (*i* + *Suc j*)) ⊑ (⨆ *i*. *Y* (*i* + *Suc j*)))
apply (*thin-tac* ∀ *i*>*j*. *X i* = *Y i*)
apply (*simp only*: *lub-range-shift*)
apply *simp*
done

lemma *lub-equal2*:
  ⟦∃ *j*. ∀ *i*>*j*. *X i* = *Y i*; *chain* (*X*::*nat* ⇒ 'a::*cpo*); *chain Y*⟧
    ⟹ (⨆ *i*. *X i*) = (⨆ *i*. *Y i*)
by (*blast intro*: *antisym-less lub-mono2 sym*)

lemma *lub-mono3*:
  ⟦*chain* (*Y*::*nat* ⇒ 'a::*cpo*); *chain X*; ∀ *i*. ∃ *j*. *Y i* ⊑ *X j*⟧
    ⟹ (⨆ *i*. *Y i*) ⊑ (⨆ *i*. *X i*)
apply (*erule is-lub-thelub*)
apply (*rule ub-rangeI*)
apply (*erule allE*)
apply (*erule exE*)
apply (*erule trans-less*)
apply (*erule is-ub-thelub*)
done

lemma *ch2ch-lub*:
  fixes *Y* :: *nat* ⇒ *nat* ⇒ 'a::*cpo*
  assumes *1*: ⋀*j*. *chain* (λ*i*. *Y i j*)
  assumes *2*: ⋀*i*. *chain* (λ*j*. *Y i j*)
  shows *chain* (λ*i*. ⨆ *j*. *Y i j*)
apply (*rule chainI*)
apply (*rule lub-mono* [*rule-format*, *OF 2 2*])
apply (*rule chainE* [*OF 1*])
done

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 (*rule antisym-less*)
have *3*: $chain\ (\lambda i.\ Y\ i\ i)$
apply (*rule chainI*)
apply (*rule trans-less*)
apply (*rule chainE* [*OF 1*])
apply (*rule chainE* [*OF 2*])
done
have *4*: $chain\ (\lambda i.\ \bigsqcup j.\ Y\ i\ j)$
by (*rule ch2ch-lub* [*OF 1 2*])
show $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) \sqsubseteq (\bigsqcup i.\ Y\ i\ i)$
apply (*rule is-lub-thelub* [*OF 4*])
apply (*rule ub-rangeI*)
apply (*rule lub-mono3* [*rule-format*, *OF 2 3*])
apply (*rule exI*)
apply (*rule trans-less*)
apply (*rule chain-mono3* [*OF 1 le-maxI1*])
apply (*rule chain-mono3* [*OF 2 le-maxI2*])
done
show $(\bigsqcup i.\ Y\ i\ i) \sqsubseteq (\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j)$
apply (*rule lub-mono* [*rule-format*, *OF 3 4*])
apply (*rule is-ub-thelub* [*OF 2*])
done
qed

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 $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) = (\bigsqcup j.\ \bigsqcup i.\ Y\ i\ j)$
by (*simp add*: *diag-lub 1 2*)

2.2 Pointed cpos

The class pcpo of pointed cpos

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

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

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

derive the old rule minimal

lemma *UU-least*: $\forall z.\ \bot \sqsubseteq z$
apply (*unfold UU-def*)
apply (*rule theI'*)
apply (*rule ex-ex1I*)
apply (*rule least*)
apply (*blast intro*: *antisym-less*)
done

lemma *minimal* [*iff*]: $\bot \sqsubseteq x$
by (*rule UU-least* [*THEN spec*])

lemma *UU-reorient*: $(\bot = x) = (x = \bot)$
by *auto*

ML-setup ⟪
local
  val meta-UU-reorient = thm UU-reorient RS eq-reflection;
  fun reorient-proc sg - (- $ t $ u) =
    case u of
        Const(Pcpo.UU,-) => NONE
      | *Const(HOL.zero, -) => NONE*
      | *Const(HOL.one, -) => NONE*
      | *Const(Numeral.number-of, -) $ - => NONE*
      | *- => SOME meta-UU-reorient*;
in
  val UU-reorient-simproc =
    Simplifier.simproc @{theory} UU-reorient-simproc [UU=x] reorient-proc
end;

Addsimprocs [*UU-reorient-simproc*];
⟫

useful lemmas about $\bot$

lemma *less-UU-iff* [*simp*]: $(x \sqsubseteq \bot) = (x = \bot)$
by (*simp add*: *po-eq-conv*)

lemma *eq-UU-iff*: $(x = \bot) = (x \sqsubseteq \bot)$
by *simp*

lemma *UU-I*: $x \sqsubseteq \bot \Longrightarrow x = \bot$
by (*subst eq-UU-iff*)

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

lemma *chain-UU-I*: $[\![chain\ Y;\ (\bigsqcup i.\ Y\ i) = \bot]\!] \Longrightarrow \forall i.\ Y\ i = \bot$
apply (*rule allI*)
apply (*rule UU-I*)

apply (*erule subst*)
apply (*erule is-ub-thelub*)
done

lemma *chain-UU-I-inverse*: $\forall i{::}nat.\ Y\ i = \bot \Longrightarrow (\bigsqcup i.\ Y\ i) = \bot$
apply (*rule lub-chain-maxelem*)
apply (*erule spec*)
apply *simp*
done

lemma *chain-UU-I-inverse2*: $(\bigsqcup i.\ Y\ i) \neq \bot \Longrightarrow \exists i{::}nat.\ Y\ i \neq \bot$
by (*blast intro*: *chain-UU-I-inverse*)

lemma *notUU-I*: $[\![x \sqsubseteq y;\ x \neq \bot]\!] \Longrightarrow y \neq \bot$
by (*blast intro*: *UU-I*)

lemma *chain-mono2*: $[\![\exists j.\ Y\ j \neq \bot;\ chain\ Y]\!] \Longrightarrow \exists j.\ \forall i{>}j.\ Y\ i \neq \bot$
by (*blast dest*: *notUU-I chain-mono*)

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*:
$chain\ (S{::}nat \Rightarrow {'a}{::}chfin) \Longrightarrow \exists x.\ range\ S <<|\ x$
apply (*frule chfin* [*rule-format*])
apply (*blast intro*: *lub-finch1*)
done

instance *chfin* < *cpo*
by *intro-classes* (*rule chfin-imp-cpo*)

flat types are chfin

lemma *flat-imp-chfin*:
$\forall Y{::}nat \Rightarrow {'a}{::}flat.\ chain\ Y \longrightarrow (\exists n.\ max\text{-}in\text{-}chain\ n\ Y)$
apply (*unfold max-in-chain-def*)
apply *clarify*
apply (*case-tac* $\forall i.\ Y\ i = \bot$)
apply *simp*
apply *simp*

apply (*erule exE*)
apply (*rule-tac x=i* **in** *exI*)
apply *clarify*
apply (*blast dest*: *chain-mono3 ax-flat* [*rule-format*])
done

instance *flat* < *chfin*
by *intro-classes* (*rule flat-imp-chfin*)

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

lemma *flat-eq*: (*a*::'*a*::*flat*) ≠ ⊥ ⟹ *a* ⊑ *b* = (*a* = *b*)
by (*safe dest*!: *ax-flat* [*rule-format*])

lemma *chfin2finch*: *chain* (*Y*::*nat* ⇒ '*a*::*chfin*) ⟹ *finite-chain Y*
by (*simp add*: *chfin finite-chain-def*)

lemmata for improved admissibility introdution rule

lemma *infinite-chain-adm-lemma*:
⟦*chain Y*; ∀ *i*. *P* (*Y i*);
⋀*Y*. ⟦*chain Y*; ∀ *i*. *P* (*Y i*); ¬ *finite-chain Y*⟧ ⟹ *P* (⨆ *i*. *Y i*)⟧
⟹ *P* (⨆ *i*. *Y i*)
apply (*case-tac finite-chain Y*)
prefer *2* **apply** *fast*
apply (*unfold finite-chain-def*)
apply *safe*
apply (*erule lub-finch1* [*THEN thelubI*, *THEN ssubst*])
apply *assumption*
apply (*erule spec*)
done

lemma *increasing-chain-adm-lemma*:
⟦*chain Y*; ∀ *i*. *P* (*Y i*); ⋀*Y*. ⟦*chain Y*; ∀ *i*. *P* (*Y i*);
∀ *i*. ∃ *j*>*i*. *Y i* ≠ *Y j* ∧ *Y i* ⊑ *Y j*⟧ ⟹ *P* (⨆ *i*. *Y i*)⟧
⟹ *P* (⨆ *i*. *Y i*)
apply (*erule infinite-chain-adm-lemma*)
apply *assumption*
apply (*erule thin-rl*)
apply (*unfold finite-chain-def*)
apply (*unfold max-in-chain-def*)
apply (*fast dest*: *le-imp-less-or-eq elim*: *chain-mono*)
done

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* **..**

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

lemma *refl-less-fun*: $(f::'a::type \Rightarrow 'b::po) \sqsubseteq f$
by (*simp add*: *less-fun-def*)

lemma *antisym-less-fun*:
$[\![(f1::'a::type \Rightarrow 'b::po) \sqsubseteq f2;\ f2 \sqsubseteq f1]\!] \Longrightarrow f1 = f2$
by (*simp add*: *less-fun-def expand-fun-eq antisym-less*)

lemma *trans-less-fun*:
$[\![(f1::'a::type \Rightarrow 'b::po) \sqsubseteq f2;\ f2 \sqsubseteq f3]\!] \Longrightarrow f1 \sqsubseteq f3$
apply (*unfold less-fun-def*)
apply *clarify*
apply (*rule trans-less*)
apply (*erule spec*)
apply (*erule spec*)
done

instance *fun* :: (*type*, *po*) *po*
by *intro-classes*
(*assumption* | *rule refl-less-fun antisym-less-fun trans-less-fun*)+

make the symbol << accessible for type fun

lemma *expand-fun-less*: $(f \sqsubseteq g) = (\forall x.\ f\ x \sqsubseteq g\ x)$
by (*simp add*: *less-fun-def*)

lemma *less-fun-ext*: $(\bigwedge x.\ f\ x \sqsubseteq g\ x) \Longrightarrow f \sqsubseteq g$
by (*simp add*: *less-fun-def*)

3.2 Full function space is chain complete

chains of functions yield chains in the po range

lemma *ch2ch-fun*: *chain* $S \Longrightarrow$ *chain* $(\lambda i.\ S\ i\ x)$
by (*simp add*: *chain-def less-fun-def*)

lemma *ch2ch-lambda*: $(\bigwedge x.$ *chain* $(\lambda i.\ S\ i\ x)) \Longrightarrow$ *chain* S
by (*simp add*: *chain-def less-fun-def*)

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

lemma *ub2ub-fun*:

range (*S*::*nat* ⇒ 'a ⇒ 'b::*po*) <| *u* ⟹ *range* (λ*i*. *S i x*) <| *u x*
by (*auto simp add*: *is-ub-def less-fun-def*)

Type 'a ⇒ 'b is chain complete

lemma *lub-fun*:
chain (*S*::*nat* ⇒ 'a::*type* ⇒ 'b::*cpo*)
⟹ *range S* <<| (λ*x*. ⨆ *i*. *S i x*)
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*rule less-fun-ext*)
apply (*rule is-ub-thelub*)
apply (*erule ch2ch-fun*)
apply (*rule less-fun-ext*)
apply (*rule is-lub-thelub*)
apply (*erule ch2ch-fun*)
apply (*erule ub2ub-fun*)
done

lemma *thelub-fun*:
chain (*S*::*nat* ⇒ 'a::*type* ⇒ 'b::*cpo*)
⟹ *lub* (*range S*) = (λ*x*. ⨆ *i*. *S i x*)
by (*rule lub-fun* [*THEN thelubI*])

lemma *cpo-fun*:
chain (*S*::*nat* ⇒ 'a::*type* ⇒ 'b::*cpo*) ⟹ ∃ *x*. *range S* <<| *x*
by (*rule exI*, *erule lub-fun*)

instance *fun* :: (*type*, *cpo*) *cpo*
by *intro-classes* (*rule cpo-fun*)

3.3 Full function space is pointed

lemma *minimal-fun*: (λ*x*. ⊥) ⊑ *f*
by (*simp add*: *less-fun-def*)

lemma *least-fun*: ∃ *x*::'a ⇒ 'b::*pcpo*. ∀ *y*. *x* ⊑ *y*
apply (*rule-tac x* = λ*x*. ⊥ **in** *exI*)
apply (*rule minimal-fun* [*THEN allI*])
done

instance *fun* :: (*type*, *pcpo*) *pcpo*
by *intro-classes* (*rule least-fun*)

for compatibility with old HOLCF-Version

lemma *inst-fun-pcpo*: ⊥ = (λ*x*. ⊥)
by (*rule minimal-fun* [*THEN UU-I*, *symmetric*])

function application is strict in the left argument

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

by (*simp add*: *inst-fun-pcpo*)

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*:
$[\![\bigwedge Y.\ chain\ Y \Longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i))]\!] \Longrightarrow contlub\ f$
by (*simp add*: *contlub-def*)

lemma *contlubE*:
$[\![contlub\ f;\ chain\ Y]\!] \Longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i))$
by (*simp add*: *contlub-def*)

lemma *contI*:
$[\![\bigwedge Y.\ chain\ Y \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow cont\ f$
by (*simp add*: *cont-def*)

lemma *contE*:
$[\![cont\ f;\ chain\ Y]\!] \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i)$
by (*simp add*: *cont-def*)

lemma *monofunI*:
$[\![\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y]\!] \Longrightarrow monofun\ f$
by (*simp add*: *monofun-def*)

lemma *monofunE*:
⟦*monofun f*; $x \sqsubseteq y$⟧ $\Longrightarrow$ $f\ x \sqsubseteq f\ y$
by (*simp add*: *monofun-def*)

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

lemma *monofun-fun-fun*: $f \sqsubseteq g \Longrightarrow f\ x \sqsubseteq g\ x$
by (*simp add*: *less-fun-def*)

lemma *monofun-fun-arg*: ⟦*monofun f*; $x \sqsubseteq y$⟧ $\Longrightarrow$ $f\ x \sqsubseteq f\ y$
by (*rule monofunE*)

lemma *monofun-fun*: ⟦*monofun f*; *monofun g*; $f \sqsubseteq g$; $x \sqsubseteq y$⟧ $\Longrightarrow$ $f\ x \sqsubseteq g\ y$
by (*rule trans-less* [*OF monofun-fun-arg monofun-fun-fun*])

4.2 *monofun f* ∧ *contlub f* ≡ *cont f*

monotone functions map chains to chains

lemma *ch2ch-monofun*: ⟦*monofun f*; *chain Y*⟧ $\Longrightarrow$ *chain* ($\lambda i.\ f\ (Y\ i)$)
apply (*rule chainI*)
apply (*erule monofunE*)
apply (*erule chainE*)
done

monotone functions map upper bound to upper bounds

lemma *ub2ub-monofun*:
⟦*monofun f*; *range Y* <| *u*⟧ $\Longrightarrow$ *range* ($\lambda i.\ f\ (Y\ i)$) <| $f\ u$
apply (*rule ub-rangeI*)
apply (*erule monofunE*)
apply (*erule ub-rangeD*)
done

left to right: *monofun f* ∧ *contlub f* $\Longrightarrow$ *cont f*

lemma *monocontlub2cont*: ⟦*monofun f*; *contlub f*⟧ $\Longrightarrow$ *cont f*
apply (*rule contI*)
apply (*rule thelubE*)
apply (*erule* (*1*) *ch2ch-monofun*)
apply (*erule* (*1*) *contlubE* [*symmetric*])
done

first a lemma about binary chains

lemma *binchain-cont*:
⟦*cont f*; $x \sqsubseteq y$⟧ $\Longrightarrow$ *range* ($\lambda i{::}nat.\ f\ (if\ i = 0\ then\ x\ else\ y)$) <<| $f\ y$
apply (*subgoal-tac f* ($\bigsqcup i{::}nat.\ if\ i = 0\ then\ x\ else\ y$) = $f\ y$)
apply (*erule subst*)
apply (*erule contE*)
apply (*erule bin-chain*)
apply (*rule-tac f*=*f* **in** *arg-cong*)

apply (*erule lub-bin-chain* [*THEN thelubI*])
done

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

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

lemma *cont2mono*: $cont\ f \Longrightarrow monofun\ f$
apply (*rule monofunI*)
apply (*drule* (*1*) *binchain-cont*)
apply (*drule-tac i=0* **in** *is-ub-lub*)
apply *simp*
done

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*: $cont\ f \Longrightarrow contlub\ f$
apply (*rule contlubI*)
apply (*rule thelubI* [*symmetric*])
apply (*erule* (*1*) *contE*)
done

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

4.3 Continuity of basic functions

The identity function is continuous

lemma *cont-id*: $cont\ (\lambda x.\ x)$
apply (*rule contI*)
apply (*erule thelubE*)
apply (*rule refl*)
done

constant functions are continuous

lemma *cont-const*: $cont\ (\lambda x.\ c)$
apply (*rule contI*)
apply (*rule lub-const*)
done

if-then-else is continuous

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

4.4 Propagation of monotonicity and continuity

the lub of a chain of monotone functions is monotone

lemma *monofun-lub-fun*:
⟦*chain* (*F*::*nat* ⇒ *'a* ⇒ *'b*::*cpo*); ∀ *i*. *monofun* (*F i*)⟧
⟹ *monofun* (⨆ *i*. *F i*)
apply (*rule monofunI*)
apply (*simp add*: *thelub-fun*)
apply (*rule lub-mono* [*rule-format*])
apply (*erule ch2ch-fun*)
apply (*erule ch2ch-fun*)
apply (*simp add*: *monofunE*)
done

the lub of a chain of continuous functions is continuous

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

lemma *contlub-lub-fun*:
⟦*chain F*; ∀ *i*. *cont* (*F i*)⟧ ⟹ *contlub* (⨆ *i*. *F i*)
apply (*rule contlubI*)
apply (*simp add*: *thelub-fun*)
apply (*simp add*: *cont2contlubE*)
apply (*rule ex-lub*)
apply (*erule ch2ch-fun*)
apply (*simp add*: *ch2ch-cont*)
done

lemma *cont-lub-fun*:
⟦*chain F*; ∀ *i*. *cont* (*F i*)⟧ ⟹ *cont* (⨆ *i*. *F i*)
apply (*rule monocontlub2cont*)
apply (*erule monofun-lub-fun*)
apply (*simp add*: *cont2mono*)
apply (*erule* (*1*) *contlub-lub-fun*)
done

lemma *cont2cont-lub*:
⟦*chain F*; ⋀*i*. *cont* (*F i*)⟧ ⟹ *cont* (λ*x*. ⨆ *i*. *F i x*)
by (*simp add*: *thelub-fun* [*symmetric*] *cont-lub-fun*)

lemma *mono2mono-fun*: *monofun f* ⟹ *monofun* (λ*x*. *f x y*)
apply (*rule monofunI*)
apply (*erule* (*1*) *monofun-fun-arg* [*THEN monofun-fun-fun*])
done

lemma *cont2cont-fun*: *cont f* ⟹ *cont* (λ*x*. *f x y*)
apply (*rule monocontlub2cont*)
apply (*erule cont2mono* [*THEN mono2mono-fun*])
apply (*rule contlubI*)
apply (*simp add*: *cont2contlubE*)
apply (*simp add*: *thelub-fun ch2ch-cont*)
done

Note (λ*x*. λ*y*. *f x y*) = *f*

lemma *mono2mono-lambda*: $(\bigwedge y.\ monofun\ (\lambda x.\ f\ x\ y)) \Longrightarrow monofun\ f$
apply (*rule monofunI*)
apply (*rule less-fun-ext*)
apply (*blast dest*: *monofunE*)
done

lemma *cont2cont-lambda*: $(\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)) \Longrightarrow cont\ f$
apply (*subgoal-tac monofun f*)
apply (*rule monocontlub2cont*)
apply *assumption*
apply (*rule contlubI*)
apply (*rule ext*)
apply (*simp add*: *thelub-fun ch2ch-monofun*)
apply (*blast dest*: *cont2contlubE*)
apply (*simp add*: *mono2mono-lambda cont2mono*)
done

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

lemma *contlub-lambda*:
$(\bigwedge x{::}'a{::}type.\ chain\ (\lambda i.\ S\ i\ x{::}'b{::}cpo))$
$\Longrightarrow (\lambda x.\ \bigsqcup i.\ S\ i\ x) = (\bigsqcup i.\ (\lambda x.\ S\ i\ x))$
by (*simp add*: *thelub-fun ch2ch-lambda*)

lemma *contlub-abstraction*:
$[\![chain\ Y;\ \forall y.\ cont\ (\lambda x.(c{::}'a{::}cpo \Rightarrow 'b{::}type \Rightarrow 'c{::}cpo)\ x\ y)]\!] \Longrightarrow$
$(\lambda y.\ \bigsqcup i.\ c\ (Y\ i)\ y) = (\bigsqcup i.\ (\lambda y.\ c\ (Y\ i)\ y))$
apply (*rule thelub-fun* [*symmetric*])
apply (*rule ch2ch-cont*)
apply (*simp add*: *cont2cont-lambda*)
apply *assumption*
done

lemma *mono2mono-app*:
$[\![monofun\ f;\ \forall x.\ monofun\ (f\ x);\ monofun\ t]\!] \Longrightarrow monofun\ (\lambda x.\ (f\ x)\ (t\ x))$
apply (*rule monofunI*)
apply (*simp add*: *monofun-fun monofunE*)
done

lemma *cont2contlub-app*:
$[\![cont\ f;\ \forall x.\ cont\ (f\ x);\ cont\ t]\!] \Longrightarrow contlub\ (\lambda x.\ (f\ x)\ (t\ x))$
apply (*rule contlubI*)
apply (*subgoal-tac chain* $(\lambda i.\ f\ (Y\ i))$)
apply (*subgoal-tac chain* $(\lambda i.\ t\ (Y\ i))$)
apply (*simp add*: *cont2contlubE thelub-fun*)
apply (*rule diag-lub*)
apply (*erule ch2ch-fun*)
apply (*drule spec*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule* (*1*) *ch2ch-cont*)

apply (*erule* (*1*) *ch2ch-cont*)
done

lemma *cont2cont-app*:
⟦*cont f*; ∀ *x*. *cont* (*f x*); *cont t*⟧ ⟹ *cont* (λ*x*. (*f x*) (*t x*))
by (*blast intro*: *monocontlub2cont mono2mono-app cont2mono cont2contlub-app*)

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

lemma *cont2cont-app3*: ⟦*cont f*; *cont t*⟧ ⟹ *cont* (λ*x*. *f* (*t x*))
by (*rule cont2cont-app2* [*OF cont-const*])

4.5 Finite chains and flat pcpos

monotone functions map finite chains to finite chains

lemma *monofun-finch2finch*:
⟦*monofun f*; *finite-chain Y*⟧ ⟹ *finite-chain* (λ*n*. *f* (*Y n*))
apply (*unfold finite-chain-def*)
apply (*simp add*: *ch2ch-monofun*)
apply (*force simp add*: *max-in-chain-def*)
done

The same holds for continuous functions

lemma *cont-finch2finch*:
⟦*cont f*; *finite-chain Y*⟧ ⟹ *finite-chain* (λ*n*. *f* (*Y n*))
by (*rule cont2mono* [*THEN monofun-finch2finch*])

lemma *chfindom-monofun2cont*: *monofun f* ⟹ *cont* (*f*::'*a*::*chfin* ⇒ '*b*::*pcpo*)
apply (*rule monocontlub2cont*)
apply *assumption*
apply (*rule contlubI*)
apply (*frule chfin2finch*)
apply (*clarsimp simp add*: *finite-chain-def*)
apply (*subgoal-tac max-in-chain i* (λ*i*. *f* (*Y i*)))
apply (*simp add*: *maxinch-is-thelub ch2ch-monofun*)
apply (*force simp add*: *max-in-chain-def*)
done

some properties of flat

lemma *flatdom-strict2mono*: *f* ⊥ = ⊥ ⟹ *monofun* (*f*::'*a*::*flat* ⇒ '*b*::*pcpo*)
apply (*rule monofunI*)
apply (*drule ax-flat* [*rule-format*])
apply *auto*
done

lemma *flatdom-strict2cont*: *f* ⊥ = ⊥ ⟹ *cont* (*f*::'*a*::*flat* ⇒ '*b*::*pcpo*)
by (*rule flatdom-strict2mono* [*THEN chfindom-monofun2cont*])

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*:
$(\bigwedge Y.\ [\![chain\ Y; \forall i.\ P\ (Y\ i)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)) \Longrightarrow adm\ P$
by (*unfold adm-def*, *fast*)

lemma *triv-admI*: $\forall x.\ P\ x \Longrightarrow adm\ P$
by (*rule admI*, *erule spec*)

lemma *admD*: $[\![adm\ P; chain\ Y; \forall i.\ P\ (Y\ i)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
by (*unfold adm-def*, *fast*)

lemma *compactI*: $adm\ (\lambda x.\ \neg\ k \sqsubseteq x) \Longrightarrow compact\ k$
by (*unfold compact-def*)

lemma *compactD*: $compact\ k \Longrightarrow adm\ (\lambda x.\ \neg\ k \sqsubseteq x)$
by (*unfold compact-def*)

improved admissibility introduction

lemma *admI2*:
$(\bigwedge Y.\ [\![chain\ Y; \forall i.\ P\ (Y\ i); \forall i.\ \exists j{>}i.\ Y\ i \neq Y\ j \wedge Y\ i \sqsubseteq Y\ j]\!]$
$\Longrightarrow P\ (\bigsqcup i.\ Y\ i)) \Longrightarrow adm\ P$
apply (*rule admI*)
apply (*erule (1) increasing-chain-adm-lemma*)
apply *fast*
done

5.2 Admissibility on chain-finite types

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

lemma *adm-max-in-chain*:
$\forall Y.\ chain\ (Y::nat \Rightarrow 'a) \longrightarrow (\exists n.\ max\text{-}in\text{-}chain\ n\ Y)$
$\Longrightarrow adm\ (P::'a \Rightarrow bool)$
by (*auto simp add*: *adm-def maxinch-is-thelub*)

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

lemma *compact-chfin*: *compact* ($x::'a::chfin$)
by (*rule compactI*, *rule adm-chfin*)

5.3 Admissibility of special formulae and propagation

lemma *adm-not-free*: *adm* ($\lambda x.\ t$)
by (*rule admI*, *simp*)

lemma *adm-conj*: $[\![adm\ P;\ adm\ Q]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \wedge Q\ x)$
by (*fast elim*: *admD intro*: *admI*)

lemma *adm-all*: $\forall y.\ adm\ (P\ y) \Longrightarrow adm\ (\lambda x.\ \forall y.\ P\ y\ x)$
by (*fast intro*: *admI elim*: *admD*)

lemma *adm-ball*: $\forall y{\in}A.\ adm\ (P\ y) \Longrightarrow adm\ (\lambda x.\ \forall y{\in}A.\ P\ y\ x)$
by (*fast intro*: *admI elim*: *admD*)

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*:
$[\![chain\ (Y::nat \Rightarrow 'a::cpo);\ \forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!]$
$\Longrightarrow chain\ (\lambda i.\ Y\ (LEAST\ j.\ i \leq j \wedge P\ (Y\ j)))$
apply (*rule chainI*)
apply (*erule chain-mono3*)
apply (*rule Least-le*)
apply (*rule LeastI2-ex*)
apply *simp-all*
done

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

lemma *adm-disj-lemma3*:
$[\![chain\ (Y::nat \Rightarrow 'a::cpo);\ \forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!] \Longrightarrow$
$(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ Y\ (LEAST\ j.\ i \leq j \wedge P\ (Y\ j)))$
apply (*frule* (*1*) *adm-disj-lemma1*)
apply (*rule antisym-less*)
apply (*rule lub-mono* [*rule-format*], *assumption*+)
apply (*erule chain-mono3*)
apply (*simp add*: *adm-disj-lemma2*)
apply (*rule lub-range-mono*, *fast*, *assumption*+)

done

lemma *adm-disj-lemma4*:
⟦*adm P*; *chain Y*; ∀ *i*. ∃ *j*≥*i*. *P* (*Y j*)⟧ ⟹ *P* (⨆ *i*. *Y i*)
apply (*subst adm-disj-lemma3*, *assumption*+)
apply (*erule admD*)
apply (*simp add*: *adm-disj-lemma1*)
apply (*simp add*: *adm-disj-lemma2*)
done

lemma *adm-disj-lemma5*:
∀ *n*::*nat*. *P n* ∨ *Q n* ⟹ (∀ *i*. ∃ *j*≥*i*. *P j*) ∨ (∀ *i*. ∃ *j*≥*i*. *Q j*)
apply (*erule contrapos-pp*)
apply (*clarsimp*, *rename-tac a b*)
apply (*rule-tac x*=*max a b* **in** *exI*)
apply (*simp add*: *le-maxI1 le-maxI2*)
done

lemma *adm-disj*: ⟦*adm P*; *adm Q*⟧ ⟹ *adm* (λ*x*. *P x* ∨ *Q x*)
apply (*rule admI*)
apply (*erule adm-disj-lemma5* [*THEN disjE*])
apply (*erule* (*2*) *adm-disj-lemma4* [*THEN disjI1*])
apply (*erule* (*2*) *adm-disj-lemma4* [*THEN disjI2*])
done

lemma *adm-imp*: ⟦*adm* (λ*x*. ¬ *P x*); *adm Q*⟧ ⟹ *adm* (λ*x*. *P x* ⟶ *Q x*)
by (*subst imp-conv-disj*, *rule adm-disj*)

lemma *adm-iff*:
⟦*adm* (λ*x*. *P x* ⟶ *Q x*); *adm* (λ*x*. *Q x* ⟶ *P x*)⟧
⟹ *adm* (λ*x*. *P x* = *Q x*)
by (*subst iff-conv-conj-imp*, *rule adm-conj*)

lemma *adm-not-conj*:
⟦*adm* (λ*x*. ¬ *P x*); *adm* (λ*x*. ¬ *Q x*)⟧ ⟹ *adm* (λ*x*. ¬ (*P x* ∧ *Q x*))
by (*simp add*: *adm-imp*)

admissibility and continuity

lemma *adm-less*: ⟦*cont u*; *cont v*⟧ ⟹ *adm* (λ*x*. *u x* ⊑ *v x*)
apply (*rule admI*)
apply (*simp add*: *cont2contlubE*)
apply (*rule lub-mono*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule* (*1*) *ch2ch-cont*)
apply *assumption*
done

lemma *adm-eq*: ⟦*cont u*; *cont v*⟧ ⟹ *adm* (λ*x*. *u x* = *v x*)
by (*simp add*: *po-eq-conv adm-conj adm-less*)

lemma *adm-subst*: $[\![cont\ t;\ adm\ P]\!] \Longrightarrow adm\ (\lambda x.\ P\ (t\ x))$
apply (*rule admI*)
apply (*simp add*: *cont2contlubE*)
apply (*erule admD*)
apply (*erule* (*1*) *ch2ch-cont*)
apply *assumption*
done

lemma *adm-not-less*: $cont\ t \Longrightarrow adm\ (\lambda x.\ \neg\ t\ x \sqsubseteq u)$
apply (*rule admI*)
apply (*drule-tac x=0* **in** *spec*)
apply (*erule contrapos-nn*)
apply (*erule rev-trans-less*)
apply (*erule cont2mono* [*THEN monofun-fun-arg*])
apply (*erule is-ub-thelub*)
done

admissibility and compactness

lemma *adm-compact-not-less*: $[\![compact\ k;\ cont\ t]\!] \Longrightarrow adm\ (\lambda x.\ \neg\ k \sqsubseteq t\ x)$
by (*unfold compact-def*, *erule adm-subst*)

lemma *adm-neq-compact*: $[\![compact\ k;\ cont\ t]\!] \Longrightarrow adm\ (\lambda x.\ t\ x \neq k)$
by (*simp add*: *po-eq-conv adm-imp adm-not-less adm-compact-not-less*)

lemma *adm-compact-neq*: $[\![compact\ k;\ cont\ t]\!] \Longrightarrow adm\ (\lambda x.\ k \neq t\ x)$
by (*simp add*: *po-eq-conv adm-imp adm-not-less adm-compact-not-less*)

lemma *compact-UU* [*simp*, *intro*]: $compact\ \bot$
by (*rule compactI*, *simp add*: *adm-not-free*)

lemma *adm-not-UU*: $cont\ t \Longrightarrow adm\ (\lambda x.\ t\ x \neq \bot)$
by (*simp add*: *adm-neq-compact*)

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
⟪
val adm-def = thm adm-def;
val admI = thm admI;
val triv-admI = thm triv-admI;
val admD = thm admD;
val adm-max-in-chain = thm adm-max-in-chain;
val adm-chfin = thm adm-chfin;
val admI2 = thm admI2;

val adm-less = thm adm-less;
val adm-conj = thm adm-conj;
val adm-not-free = thm adm-not-free;
val adm-not-less = thm adm-not-less;
val adm-all = thm adm-all;
val adm-all2 = thm adm-all2;
val adm-ball = thm adm-ball;
val adm-ball2 = thm adm-ball2;
val adm-subst = thm adm-subst;
val adm-not-UU = thm adm-not-UU;
val adm-eq = thm adm-eq;
val adm-disj-lemma1 = thm adm-disj-lemma1;
val adm-disj-lemma2 = thm adm-disj-lemma2;
val adm-disj-lemma3 = thm adm-disj-lemma3;
val adm-disj-lemma4 = thm adm-disj-lemma4;
val adm-disj-lemma5 = thm adm-disj-lemma5;
val adm-disj = thm adm-disj;
val adm-imp = thm adm-imp;
val adm-iff = thm adm-iff;
val adm-not-conj = thm adm-not-conj;
val adm-lemmas = thms adm-lemmas;
⟫

end

6 Pcpodef: Subtypes of pcpos

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

6.1 Proving a subtype is a partial order

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

theorem *typedef-po*:
  fixes *Abs* :: *'a*::*po* ⇒ *'b*::*sq-ord*
  assumes *type*: *type-definition Rep Abs A*
    and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
  shows *OFCLASS*(*'b*, *po-class*)
 apply (*intro-classes*, *unfold less*)
   apply (*rule refl-less*)
  apply (*rule type-definition.Rep-inject* [*OF type*, *THEN iffD1*])
  apply (*erule* (*1*) *antisym-less*)
 apply (*erule* (*1*) *trans-less*)
done

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*
by (*rule monofunI*, *unfold less*)

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-class*)
 apply (*intro-classes*, *clarify*)
 apply (*drule ch2ch-Rep* [*OF less*])
 apply (*drule chfin* [*rule-format*])
 apply (*unfold max-in-chain-def*)
 apply (*simp add*: *type-definition.Rep-inject* [*OF type*])
done

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* ($\bigsqcup i.\ Rep\ (S\ i)$)) = ($\bigsqcup i.\ Rep\ (S\ i)$)
 apply (*rule type-definition.Abs-inverse* [*OF type*])
 apply (*erule admD* [*OF adm ch2ch-Rep* [*OF less*], *rule-format*])
 apply (*rule type-definition.Rep* [*OF type*])
done

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* ($\bigsqcup i.\ Rep\ (S\ i)$)
 apply (*frule ch2ch-Rep* [*OF less*])
 apply (*rule is-lubI*)
  apply (*rule ub-rangeI*)
  apply (*simp only*: *less Abs-inverse-lub-Rep* [*OF type less adm*])
  apply (*erule is-ub-thelub*)

apply (*simp only*: *less Abs-inverse-lub-Rep* [*OF type less adm*])
apply (*erule is-lub-thelub*)
apply (*erule ub2ub-Rep* [*OF less*])
done

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

theorem *typedef-cpo*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *OFCLASS*(*'b*, *cpo-class*)
proof
fix *S*::*nat* ⇒ *'b* **assume** *chain S*
hence *range S* <<| *Abs* (⨆ *i*. *Rep* (*S i*))
by (*rule typedef-lub* [*OF type less adm*])
thus ∃ *x*. *range S* <<| *x* **..**
qed

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* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *cont Rep*
apply (*rule contI*)
apply (*simp only*: *typedef-thelub* [*OF type less adm*])
apply (*simp only*: *Abs-inverse-lub-Rep* [*OF type less adm*])
apply (*rule thelubE* [*OF - refl*])
apply (*erule ch2ch-Rep* [*OF less*])
done

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

theorem *typedef-is-lubI*:
assumes *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
shows *range* (λ*i*. *Rep* (*S i*)) <<| *Rep x* ⟹ *range S* <<| *x*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*subst less*)
apply (*erule is-ub-lub*)
apply (*subst less*)
apply (*erule is-lub-lub*)

apply (*erule ub2ub-Rep* [*OF less*])
done

theorem *typedef-cont-Abs*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
fixes *f* :: *'c*::*cpo* ⇒ *'a*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
and *f-in-A*: ⋀*x*. *f x* ∈ *A*
and *cont-f*: *cont f*
shows *cont* (λ*x*. *Abs* (*f x*))
apply (*rule contI*)
apply (*rule typedef-is-lubI* [*OF less*])
apply (*simp only*: *type-definition.Abs-inverse* [*OF type f-in-A*])
apply (*erule cont-f* [*THEN contE*])
done

6.4 Proving subtype elements are compact

theorem *typedef-compact*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *compact* (*Rep k*) ⟹ *compact k*
proof (*unfold compact-def*)
have *cont-Rep*: *cont Rep*
by (*rule typedef-cont-Rep* [*OF type less adm*])
assume *adm* (λ*x*. ¬ *Rep k* ⊑ *x*)
with *cont-Rep* **have** *adm* (λ*x*. ¬ *Rep k* ⊑ *Rep x*) **by** (*rule adm-subst*)
thus *adm* (λ*x*. ¬ *k* ⊑ *x*) **by** (*unfold less*)
qed

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* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *z-in-A*: *z* ∈ *A*
and *z-least*: ⋀*x*. *x* ∈ *A* ⟹ *z* ⊑ *x*
shows *OFCLASS*(*'b*, *pcpo-class*)
apply (*intro-classes*)
apply (*rule-tac x*=*Abs z* **in** *exI*, *rule allI*)
apply (*unfold less*)
apply (*subst type-definition.Abs-inverse* [*OF type z-in-A*])

apply (*rule z-least* [*OF type-definition.Rep* [*OF type*]])
done

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

theorem *typedef-pcpo*:
fixes *Abs* :: $'a$::*pcpo* $\Rightarrow$ $'b$::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq \equiv \lambda x\ y.$ *Rep x* $\sqsubseteq$ *Rep y*
and *UU-in-A*: $\bot \in A$
shows *OFCLASS*($'b$, *pcpo-class*)
by (*rule typedef-pcpo-generic* [*OF type less UU-in-A*], *rule minimal*)

6.5.1 Strictness of *Rep* and *Abs*

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

theorem *typedef-Abs-strict*:
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq \equiv \lambda x\ y.$ *Rep x* $\sqsubseteq$ *Rep y*
and *UU-in-A*: $\bot \in A$
shows *Abs* $\bot = \bot$
apply (*rule UU-I*, *unfold less*)
apply (*simp add*: *type-definition.Abs-inverse* [*OF type UU-in-A*])
done

theorem *typedef-Rep-strict*:
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq \equiv \lambda x\ y.$ *Rep x* $\sqsubseteq$ *Rep y*
and *UU-in-A*: $\bot \in A$
shows *Rep* $\bot = \bot$
apply (*rule typedef-Abs-strict* [*OF type less UU-in-A*, *THEN subst*])
apply (*rule type-definition.Abs-inverse* [*OF type UU-in-A*])
done

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* $\neq \bot$
apply (*rule typedef-Abs-strict* [*OF type less UU-in-A*, *THEN subst*])
apply (*simp add*: *type-definition.Abs-inject* [*OF type*] *UU-in-A*)
done

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 $x \neq \bot \Longrightarrow Rep\ x \neq \bot$
apply (*rule typedef-Rep-strict* [*OF type less UU-in-A*, *THEN subst*])
apply (*simp add*: *type-definition.Rep-inject* [*OF type*])
done

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*)
apply (*intro-classes*)
apply (*unfold less*)
apply (*simp add*: *type-definition.Rep-inject* [*OF type*, *symmetric*])
apply (*simp add*: *typedef-Rep-strict* [*OF type less UU-in-A*])
apply (*simp add*: *ax-flat*)
done

6.7 HOLCF type definition package

use *Tools/pcpodef-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*: $\exists f.\ cont\ f$
by (*rule exI*, *rule cont-const*)

lemma *adm-cont*: *adm cont*
by (*rule admI*, *rule cont-lub-fun*)

cpodef (*CFun*) ($'a$, $'b$) -> (**infixr** -> *0*) = $\{f::'a \Rightarrow 'b.\ cont\ f\}$
by (*simp add*: *Ex-cont adm-cont*)

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*

parse-translation ⟪
(∗ *rewrites* (*-cabs x t*) => (*Abs-CFun* (%*x*. *t*)) ∗)
[*mk-binder-tr* (*-cabs*, @{*const-syntax Abs-CFun*})];
⟫

To avoid eta-contraction of body:

typed-print-translation ⟪
let
fun cabs-tr' - - [Abs abs] = let
val (*x*,*t*) = *atomic-abs-tr' abs*
in Syntax.const -cabs $ x $ t end

| *cabs-tr' - T* [*t*] = *let*
val xT = domain-type (*domain-type T*);
val abs' = (*x*,*xT*,(*incr-boundvars 1 t*)$*Bound 0*);
val (*x*,*t'*) = *atomic-abs-tr' abs'*;
in Syntax.const -cabs $ x $ t' end;

in [(@{*const-syntax Abs-CFun*}, *cabs-tr'*)] *end*;
⟫

Syntax for nested abstractions

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

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

parse-ast-translation ⟪
(∗ *rewrites* (*LAM x y z. t*) => (*-cabs x* (*-cabs y* (*-cabs z t*))) ∗)
(∗ *cf. Syntax.lambda-ast-tr from Syntax/syn-trans.ML* ∗)
let
fun Lambda-ast-tr [*pats*, *body*] =
Syntax.fold-ast-p -cabs (*Syntax.unfold-ast -cargs pats*, *body*)
| *Lambda-ast-tr asts = raise Syntax.AST* (*Lambda-ast-tr*, *asts*);

```
  in [(-Lambda, Lambda-ast-tr)] end;
⟫

print-ast-translation ⟪
(* rewrites (-cabs x (-cabs y (-cabs z t))) => (LAM x y z. t) *)
(* cf. Syntax.abs-ast-tr' from Syntax/syn-trans.ML *)
  let
    fun cabs-ast-tr' asts =
      (case Syntax.unfold-ast-p -cabs
          (Syntax.Appl (Syntax.Constant -cabs :: asts)) of
        ([], -) => raise Syntax.AST (cabs-ast-tr', asts)
      | (xs, body) => Syntax.Appl
          [Syntax.Constant -Lambda, Syntax.fold-ast -cargs xs, body]);
  in [(-cabs, cabs-ast-tr')] end;
⟫
```

Dummy patterns for continuous abstraction

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

7.3 Continuous function space is pointed

```
lemma UU-CFun: ⊥ ∈ CFun
by (simp add: CFun-def inst-fun-pcpo cont-const)

instance -> :: (cpo, pcpo) pcpo
by (rule typedef-pcpo [OF type-definition-CFun less-CFun-def UU-CFun])

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

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

function application is strict in its first argument

```
lemma Rep-CFun-strict1 [simp]: ⊥·x = ⊥
by (simp add: Rep-CFun-strict)
```

for compatibility with old HOLCF-Version

```
lemma inst-cfun-pcpo: ⊥ = (Λ x. ⊥)
by (simp add: inst-fun-pcpo [symmetric] Abs-CFun-strict)
```

7.4 Basic properties of continuous functions

Beta-equality for continuous functions

```
lemma Abs-CFun-inverse2: cont f ⟹ Rep-CFun (Abs-CFun f) = f
by (simp add: Abs-CFun-inverse CFun-def)
```

lemma *beta-cfun* [*simp*]: *cont f* $\Longrightarrow$ $(\Lambda\ x.\ f\,x)\cdot u = f\,u$
by (*simp add*: *Abs-CFun-inverse2*)

Eta-equality for continuous functions

lemma *eta-cfun*: $(\Lambda\ x.\ f\cdot x) = f$
by (*rule Rep-CFun-inverse*)

Extensionality for continuous functions

lemma *expand-cfun-eq*: $(f = g) = (\forall x.\ f\cdot x = g\cdot x)$
by (*simp add*: *Rep-CFun-inject* [*symmetric*] *expand-fun-eq*)

lemma *ext-cfun*: $(\bigwedge x.\ f\cdot x = g\cdot x) \Longrightarrow f = g$
by (*simp add*: *expand-cfun-eq*)

Extensionality wrt. ordering for continuous functions

lemma *expand-cfun-less*: $f \sqsubseteq g = (\forall x.\ f\cdot x \sqsubseteq g\cdot x)$
by (*simp add*: *less-CFun-def expand-fun-less*)

lemma *less-cfun-ext*: $(\bigwedge x.\ f\cdot x \sqsubseteq g\cdot x) \Longrightarrow f \sqsubseteq g$
by (*simp add*: *expand-cfun-less*)

Congruence for continuous function application

lemma *cfun-cong*: $[\![f = g;\ x = y]\!] \Longrightarrow f\cdot x = g\cdot y$
by *simp*

lemma *cfun-fun-cong*: $f = g \Longrightarrow f\cdot x = g\cdot x$
by *simp*

lemma *cfun-arg-cong*: $x = y \Longrightarrow f\cdot x = f\cdot y$
by *simp*

7.5 Continuity of application

lemma *cont-Rep-CFun1*: *cont* $(\lambda f.\ f\cdot x)$
by (*rule cont-Rep-CFun* [*THEN cont2cont-fun*])

lemma *cont-Rep-CFun2*: *cont* $(\lambda x.\ f\cdot x)$
apply (*cut-tac x*=*f* **in** *Rep-CFun*)
apply (*simp add*: *CFun-def*)
done

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*: $chain\ Y \Longrightarrow f{\cdot}(lub\ (range\ Y)) = (\bigsqcup i.\ f{\cdot}(Y\ i))$
by (*rule contlub-Rep-CFun2* [*THEN contlubE*])

lemma *cont-cfun-arg*: $chain\ Y \Longrightarrow range\ (\lambda i.\ f{\cdot}(Y\ i)) <<|\ f{\cdot}(lub\ (range\ Y))$
by (*rule cont-Rep-CFun2* [*THEN contE*])

lemma *contlub-cfun-fun*: $chain\ F \Longrightarrow lub\ (range\ F){\cdot}x = (\bigsqcup i.\ F\ i{\cdot}x)$
by (*rule contlub-Rep-CFun1* [*THEN contlubE*])

lemma *cont-cfun-fun*: $chain\ F \Longrightarrow range\ (\lambda i.\ F\ i{\cdot}x) <<|\ lub\ (range\ F){\cdot}x$
by (*rule cont-Rep-CFun1* [*THEN contE*])

monotonicity of application

lemma *monofun-cfun-fun*: $f \sqsubseteq g \Longrightarrow f{\cdot}x \sqsubseteq g{\cdot}x$
by (*simp add*: *expand-cfun-less*)

lemma *monofun-cfun-arg*: $x \sqsubseteq y \Longrightarrow f{\cdot}x \sqsubseteq f{\cdot}y$
by (*rule monofun-Rep-CFun2* [*THEN monofunE*])

lemma *monofun-cfun*: $[\![f \sqsubseteq g;\ x \sqsubseteq y]\!] \Longrightarrow f{\cdot}x \sqsubseteq g{\cdot}y$
by (*rule trans-less* [*OF monofun-cfun-fun monofun-cfun-arg*])

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

lemma *chain-monofun*: $chain\ Y \Longrightarrow chain\ (\lambda i.\ f{\cdot}(Y\ i))$
by (*erule monofun-Rep-CFun2* [*THEN ch2ch-monofun*])

lemma *ch2ch-Rep-CFunR*: $chain\ Y \Longrightarrow chain\ (\lambda i.\ f{\cdot}(Y\ i))$
by (*rule monofun-Rep-CFun2* [*THEN ch2ch-monofun*])

lemma *ch2ch-Rep-CFunL*: $chain\ F \Longrightarrow chain\ (\lambda i.\ (F\ i){\cdot}x)$
by (*rule monofun-Rep-CFun1* [*THEN ch2ch-monofun*])

lemma *ch2ch-Rep-CFun* [*simp*]:
$[\![chain\ F;\ chain\ Y]\!] \Longrightarrow chain\ (\lambda i.\ (F\ i){\cdot}(Y\ i))$
apply (*rule chainI*)
apply (*rule monofun-cfun*)
apply (*erule chainE*)
apply (*erule chainE*)
done

lemma *ch2ch-LAM*: $[\![\bigwedge x.\ chain\ (\lambda i.\ S\ i\ x);\ \bigwedge i.\ cont\ (\lambda x.\ S\ i\ x)]\!]$
$\Longrightarrow chain\ (\lambda i.\ \Lambda\ x.\ S\ i\ x)$
by (*simp add*: *chain-def expand-cfun-less*)

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

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

by (*simp add*: *contlub-cfun-fun contlub-cfun-arg diag-lub*)

lemma *cont-cfun*:
⟦*chain F*; *chain Y*⟧ ⟹ *range* (λ*i*. *F i*·(*Y i*)) <<| (⨆ *i*. *F i*)·(⨆ *i*. *Y i*)
apply (*rule thelubE*)
apply (*simp only*: *ch2ch-Rep-CFun*)
apply (*simp only*: *contlub-cfun*)
done

lemma *contlub-LAM*:
⟦⋀*x*. *chain* (λ*i*. *F i x*); ⋀*i*. *cont* (λ*x*. *F i x*)⟧
⟹ (Λ *x*. ⨆ *i*. *F i x*) = (⨆ *i*. Λ *x*. *F i x*)
apply (*simp add*: *thelub-CFun ch2ch-LAM*)
apply (*simp add*: *Abs-CFun-inverse2*)
apply (*simp add*: *thelub-fun ch2ch-lambda*)
done

strictness

lemma *strictI*: *f*·*x* = ⊥ ⟹ *f*·⊥ = ⊥
apply (*rule UU-I*)
apply (*erule subst*)
apply (*rule minimal* [*THEN monofun-cfun-arg*])
done

the lub of a chain of continous functions is monotone

lemma *lub-cfun-mono*: *chain F* ⟹ *monofun* (λ*x*. ⨆ *i*. *F i*·*x*)
apply (*drule ch2ch-monofun* [*OF monofun-Rep-CFun*])
apply (*simp add*: *thelub-fun* [*symmetric*])
apply (*erule monofun-lub-fun*)
apply (*simp add*: *monofun-Rep-CFun2*)
done

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

lemma *ex-lub-cfun*:
⟦*chain F*; *chain Y*⟧ ⟹ (⨆ *j*. ⨆ *i*. *F j*·(*Y i*)) = (⨆ *i*. ⨆ *j*. *F j*·(*Y i*))
by (*simp add*: *diag-lub*)

the lub of a chain of cont. functions is continuous

lemma *cont-lub-cfun*: *chain F* ⟹ *cont* (λ*x*. ⨆ *i*. *F i*·*x*)
apply (*rule cont2cont-lub*)
apply (*erule monofun-Rep-CFun* [*THEN ch2ch-monofun*])
apply (*rule cont-Rep-CFun2*)
done

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

lemma *lub-cfun*: *chain F* ⟹ *range F* <<| (Λ *x*. ⨆ *i*. *F i*·*x*)
by (*simp only*: *contlub-cfun-fun* [*symmetric*] *eta-cfun thelubE*)

lemma *thelub-cfun*: *chain F* $\Longrightarrow$ *lub* (*range F*) = (Λ *x*. $\bigsqcup$ *i*. *F i*·*x*)
by (*rule lub-cfun* [*THEN thelubI*])

7.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

lemma *cont2cont-Rep-CFun*:
⟦*cont f*; *cont t*⟧ $\Longrightarrow$ *cont* (λx. (*f x*)·(*t x*))
by (*best intro*: *cont2cont-app2 cont-const cont-Rep-CFun cont-Rep-CFun2*)

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*)
apply (*rule monofunI*)
apply (*rule less-cfun-ext*)
apply (*simp add*: *p1*)
apply (*erule p2* [*THEN monofunE*])
done

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*)
apply (*rule cont-Abs-CFun*)
apply (*simp add*: *p1 CFun-def*)
apply (*simp add*: *p2 cont2cont-lambda*)
done

continuity simplification procedure

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

use *Tools/cont-proc.ML*
setup *ContProc.setup*

7.7 Miscellaneous

Monotonicity of *Abs-CFun*

lemma *semi-monofun-Abs-CFun*:
⟦*cont f*; *cont g*; *f* $\sqsubseteq$ *g*⟧ $\Longrightarrow$ *Abs-CFun f* $\sqsubseteq$ *Abs-CFun g*
by (*simp add*: *less-CFun-def Abs-CFun-inverse2*)

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

lemma *chfin-Rep-CFunR*: *chain* (*Y*::*nat* => *'a*::*cpo*−>*'b*::*chfin*)

==> !s. ? n. $lub(range(Y))\$s = Y\ n\$s$
apply (*rule allI*)
apply (*subst contlub-cfun-fun*)
apply *assumption*
apply (*fast intro*!: *thelubI chfin lub-finch2 chfin2finch ch2ch-Rep-CFunL*)
done

lemma *adm-chfindom*: *adm* ($\lambda u{::}'a{::}cpo \to 'b{::}chfin.\ P(u{\cdot}s)$)
by (*rule adm-subst*, *simp*, *rule adm-chfin*)

7.8 Continuous injection-retraction pairs

Continuous retractions are strict.

lemma *retraction-strict*:
$\forall x.\ f{\cdot}(g{\cdot}x) = x \Longrightarrow f{\cdot}\bot = \bot$
apply (*rule UU-I*)
apply (*drule-tac x*=$\bot$ **in** *spec*)
apply (*erule subst*)
apply (*rule monofun-cfun-arg*)
apply (*rule minimal*)
done

lemma *injection-eq*:
$\forall x.\ f{\cdot}(g{\cdot}x) = x \Longrightarrow (g{\cdot}x = g{\cdot}y) = (x = y)$
apply (*rule iffI*)
apply (*drule-tac f*=*f* **in** *cfun-arg-cong*)
apply *simp*
apply *simp*
done

lemma *injection-less*:
$\forall x.\ f{\cdot}(g{\cdot}x) = x \Longrightarrow (g{\cdot}x \sqsubseteq g{\cdot}y) = (x \sqsubseteq y)$
apply (*rule iffI*)
apply (*drule-tac f*=*f* **in** *monofun-cfun-arg*)
apply *simp*
apply (*erule monofun-cfun-arg*)
done

lemma *injection-defined-rev*:
$[\![\forall x.\ f{\cdot}(g{\cdot}x) = x;\ g{\cdot}z = \bot]\!] \Longrightarrow z = \bot$
apply (*drule-tac f*=*f* **in** *cfun-arg-cong*)
apply (*simp add*: *retraction-strict*)
done

lemma *injection-defined*:
$[\![\forall x.\ f{\cdot}(g{\cdot}x) = x;\ z \neq \bot]\!] \Longrightarrow g{\cdot}z \neq \bot$
by (*erule contrapos-nn*, *rule injection-defined-rev*)

propagation of flatness and chain-finiteness by retractions

lemma *chfin2chfin*:
$\forall y.\ (f::'a::chfin \to 'b)\cdot(g\cdot y) = y$
$\Longrightarrow \forall Y::nat \Rightarrow 'b.\ chain\ Y \longrightarrow (\exists n.\ max\text{-}in\text{-}chain\ n\ Y)$
apply *clarify*
apply (*drule-tac f=g* **in** *chain-monofun*)
apply (*drule chfin* [*rule-format*])
apply (*unfold max-in-chain-def*)
apply (*simp add*: *injection-eq*)
done

lemma *flat2flat*:
$\forall y.\ (f::'a::flat \to 'b::pcpo)\cdot(g\cdot y) = y$
$\Longrightarrow \forall x\ y::'b.\ x \sqsubseteq y \longrightarrow x = \bot \vee x = y$
apply *clarify*
apply (*drule-tac f=g* **in** *monofun-cfun-arg*)
apply (*drule ax-flat* [*rule-format*])
apply (*erule disjE*)
apply (*simp add*: *injection-defined-rev*)
apply (*simp add*: *injection-eq*)
done

a result about functions with flat codomain

lemma *flat-eqI*: $[\![(x::'a::flat) \sqsubseteq y;\ x \neq \bot]\!] \Longrightarrow x = y$
by (*drule ax-flat* [*rule-format*], *simp*)

lemma *flat-codom*:
$f\cdot x = (c::'b::flat) \Longrightarrow f\cdot\bot = \bot \vee (\forall z.\ f\cdot z = c)$
apply (*case-tac* $f\cdot x = \bot$)
apply (*rule disjI1*)
apply (*rule UU-I*)
apply (*erule-tac t=*$\bot$ **in** *subst*)
apply (*rule minimal* [*THEN monofun-cfun-arg*])
apply *clarify*
apply (*rule-tac a* $= f\cdot\bot$ **in** *refl* [*THEN box-equals*])
apply (*erule minimal* [*THEN monofun-cfun-arg*, *THEN flat-eqI*])
apply (*erule minimal* [*THEN monofun-cfun-arg*, *THEN flat-eqI*])
done

7.9 Identity and composition

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

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

abbreviation

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

lemma *ID1* [*simp*]: *ID*·*x* = *x*
by (*simp add*: *ID-def*)

lemma *cfcomp1*: (*f oo g*) = (Λ *x*. *f*·(*g*·*x*))
by (*simp add*: *oo-def*)

lemma *cfcomp2* [*simp*]: (*f oo g*)·*x* = *f*·(*g*·*x*)
by (*simp add*: *cfcomp1*)

lemma *cfcomp-strict* [*simp*]: ⊥ *oo f* = ⊥
by (*simp add*: *expand-cfun-eq*)

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

lemma *ID2* [*simp*]: *f oo ID* = *f*
by (*rule ext-cfun*, *simp*)

lemma *ID3* [*simp*]: *ID oo f* = *f*
by (*rule ext-cfun*, *simp*)

lemma *assoc-oo*: *f oo* (*g oo h*) = (*f oo g*) *oo h*
by (*rule ext-cfun*, *simp*)

7.10 Strictified functions

defaultsort *pcpo*

definition
strictify :: $('a \to 'b) \to 'a \to 'b$ **where**
strictify = (Λ *f x*. *if x* = ⊥ *then* ⊥ *else f*·*x*)

results about strictify

lemma *cont-strictify1*: *cont* (λ*f*. *if x* = ⊥ *then* ⊥ *else f*·*x*)
by (*simp add*: *cont-if*)

lemma *monofun-strictify2*: *monofun* (λ*x*. *if x* = ⊥ *then* ⊥ *else f*·*x*)
apply (*rule monofunI*)
apply (*auto simp add*: *monofun-cfun-arg eq-UU-iff* [*symmetric*])
done

lemma *contlub-strictify2*: *contlub* (λ*x*. *if x* = ⊥ *then* ⊥ *else f*·*x*)
apply (*rule contlubI*)
apply (*case-tac lub* (*range Y*) = ⊥)

apply (*drule* (*1*) *chain-UU-I*)
apply *simp*
apply (*simp del*: *if-image-distrib*)
apply (*simp only*: *contlub-cfun-arg*)
apply (*rule lub-equal2*)
apply (*rule chain-mono2* [*THEN exE*])
apply (*erule chain-UU-I-inverse2*)
apply (*assumption*)
apply (*rule-tac x=x* **in** *exI*, *clarsimp*)
apply (*erule chain-monofun*)
apply (*erule monofun-strictify2* [*THEN ch2ch-monofun*])
done

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

lemma *strictify-conv-if*: *strictify*·*f*·*x* = (*if x* = ⊥ *then* ⊥ *else f*·*x*)
by (*unfold strictify-def*, *simp add*: *cont-strictify1 cont-strictify2*)

lemma *strictify1* [*simp*]: *strictify*·*f*·⊥ = ⊥
by (*simp add*: *strictify-conv-if*)

lemma *strictify2* [*simp*]: *x* ≠ ⊥ ⟹ *strictify*·*f*·*x* = *f*·*x*
by (*simp add*: *strictify-conv-if*)

7.11 Continuous let-bindings

definition
CLet :: *'a* → (*'a* → *'b*) → *'b* **where**
CLet = (Λ *s f*. *f*·*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* **..**

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

instance *unit* :: *po*
by *intro-classes simp-all*

instance *unit* :: *cpo*
by *intro-classes* (*simp add*: *is-lub-def is-ub-def*)

instance *unit* :: *pcpo*
by *intro-classes simp*

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

translations
$\Lambda().\ t == CONST\ unit\text{-}when{\cdot}t$

lemma *unit-when* [*simp*]: *unit-when*·*a*·*u* = *a*
by (*simp add*: *unit-when-def*)

8.2 Product type is a partial order

instance $*$ :: (*sq-ord*, *sq-ord*) *sq-ord* **..**

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*: $(p::'a * 'b) \sqsubseteq p$
by (*simp add*: *less-cprod-def*)

lemma *antisym-less-cprod*: $[\![(p1::'a * 'b) \sqsubseteq p2;\ p2 \sqsubseteq p1]\!] \Longrightarrow p1 = p2$
apply (*unfold less-cprod-def*)
apply (*rule injective-fst-snd*)
apply (*fast intro*: *antisym-less*)
apply (*fast intro*: *antisym-less*)
done

lemma *trans-less-cprod*: $[\![(p1::'a*'b) \sqsubseteq p2;\ p2 \sqsubseteq p3]\!] \Longrightarrow p1 \sqsubseteq p3$
apply (*unfold less-cprod-def*)
apply (*fast intro*: *trans-less*)
done

instance $*$ :: (*cpo*, *cpo*) *po*
by *intro-classes*

(*assumption* | *rule refl-less-cprod antisym-less-cprod trans-less-cprod*)+

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

Pair (-,-) is monotone in both arguments

lemma *monofun-pair1*: *monofun* ($\lambda x.\ (x,\ y)$)
by (*simp add*: *monofun-def less-cprod-def*)

lemma *monofun-pair2*: *monofun* ($\lambda y.\ (x,\ y)$)
by (*simp add*: *monofun-def less-cprod-def*)

lemma *monofun-pair*:
$[\![x1 \sqsubseteq x2;\ y1 \sqsubseteq y2]\!] \Longrightarrow (x1,\ y1) \sqsubseteq (x2,\ y2)$
by (*simp add*: *less-cprod-def*)

fst and *snd* are monotone

lemma *monofun-fst*: *monofun fst*
by (*simp add*: *monofun-def less-cprod-def*)

lemma *monofun-snd*: *monofun snd*
by (*simp add*: *monofun-def less-cprod-def*)

8.4 Product type is a cpo

lemma *lub-cprod*:
chain S $\Longrightarrow$ *range S* <<| ($\bigsqcup i.\ fst\ (S\ i)$, $\bigsqcup i.\ snd\ (S\ i)$)
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*rule-tac t* = *S i* **in** *surjective-pairing* [*THEN ssubst*])
apply (*rule monofun-pair*)
apply (*rule is-ub-thelub*)
apply (*erule monofun-fst* [*THEN ch2ch-monofun*])
apply (*rule is-ub-thelub*)
apply (*erule monofun-snd* [*THEN ch2ch-monofun*])
apply (*rule-tac t* = *u* **in** *surjective-pairing* [*THEN ssubst*])
apply (*rule monofun-pair*)
apply (*rule is-lub-thelub*)
apply (*erule monofun-fst* [*THEN ch2ch-monofun*])
apply (*erule monofun-fst* [*THEN ub2ub-monofun*])
apply (*rule is-lub-thelub*)
apply (*erule monofun-snd* [*THEN ch2ch-monofun*])
apply (*erule monofun-snd* [*THEN ub2ub-monofun*])
done

lemma *thelub-cprod*:
chain S $\Longrightarrow$ *lub* (*range S*) = ($\bigsqcup i.\ fst\ (S\ i)$, $\bigsqcup i.\ snd\ (S\ i)$)
by (*rule lub-cprod* [*THEN thelubI*])

lemma *cpo-cprod*:

chain (*S*::*nat* ⇒ '*a*::*cpo* ∗ '*b*::*cpo*) ⟹ ∃ *x*. *range S* <<| *x*
by (*rule exI*, *erule lub-cprod*)

instance ∗ :: (*cpo*, *cpo*) *cpo*
by *intro-classes* (*rule cpo-cprod*)

8.5 Product type is pointed

lemma *minimal-cprod*: (⊥, ⊥) ⊑ *p*
by (*simp add*: *less-cprod-def*)

lemma *least-cprod*: *EX x*::'*a*::*pcpo* ∗ '*b*::*pcpo*. *ALL y*. *x* ⊑ *y*
apply (*rule-tac x* = (⊥, ⊥) **in** *exI*)
apply (*rule minimal-cprod* [*THEN allI*])
done

instance ∗ :: (*pcpo*, *pcpo*) *pcpo*
by *intro-classes* (*rule least-cprod*)

for compatibility with old HOLCF-Version

lemma *inst-cprod-pcpo*: *UU* = (*UU*, *UU*)
by (*rule minimal-cprod* [*THEN UU-I*, *symmetric*])

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

lemma *contlub-pair1*: *contlub* (λ*x*. (*x*, *y*))
apply (*rule contlubI*)
apply (*subst thelub-cprod*)
apply (*erule monofun-pair1* [*THEN ch2ch-monofun*])
apply *simp*
done

lemma *contlub-pair2*: *contlub* (λ*y*. (*x*, *y*))
apply (*rule contlubI*)
apply (*subst thelub-cprod*)
apply (*erule monofun-pair2* [*THEN ch2ch-monofun*])
apply *simp*
done

lemma *cont-pair1*: *cont* (λ*x*. (*x*, *y*))
apply (*rule monocontlub2cont*)
apply (*rule monofun-pair1*)
apply (*rule contlub-pair1*)
done

lemma *cont-pair2*: *cont* (λ*y*. (*x*, *y*))
apply (*rule monocontlub2cont*)
apply (*rule monofun-pair2*)
apply (*rule contlub-pair2*)

done

lemma *contlub-fst*: *contlub fst*
apply (*rule contlubI*)
apply (*simp add*: *thelub-cprod*)
done

lemma *contlub-snd*: *contlub snd*
apply (*rule contlubI*)
apply (*simp add*: *thelub-cprod*)
done

lemma *cont-fst*: *cont fst*
apply (*rule monocontlub2cont*)
apply (*rule monofun-fst*)
apply (*rule contlub-fst*)
done

lemma *cont-snd*: *cont snd*
apply (*rule monocontlub2cont*)
apply (*rule monofun-snd*)
apply (*rule contlub-snd*)
done

8.7 Continuous versions of constants

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

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

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

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

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

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

translations

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

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

8.8 Convert all lemmas to the continuous versions

lemma *cpair-eq-pair*: $<x,\ y> = (x,\ y)$
by (*simp add*: *cpair-def cont-pair1 cont-pair2*)

lemma *inject-cpair*: $<a,b> = <aa,ba> \Longrightarrow a = aa \wedge b = ba$
by (*simp add*: *cpair-eq-pair*)

lemma *cpair-eq* [*iff*]: $(<a,\ b> = <a',\ b'>) = (a = a' \wedge b = b')$
by (*simp add*: *cpair-eq-pair*)

lemma *cpair-less* [*iff*]: $(<a,\ b> \sqsubseteq <a',\ b'>) = (a \sqsubseteq a' \wedge b \sqsubseteq b')$
by (*simp add*: *cpair-eq-pair less-cprod-def*)

lemma *cpair-defined-iff* [*iff*]: $(<x,\ y> = \bot) = (x = \bot \wedge y = \bot)$
by (*simp add*: *inst-cprod-pcpo cpair-eq-pair*)

lemma *cpair-strict*: $<\bot,\ \bot> = \bot$
by *simp*

lemma *inst-cprod-pcpo2*: $\bot = <\bot,\ \bot>$
by (*rule cpair-strict* [*symmetric*])

lemma *defined-cpair-rev*:
$<a,b> = \bot \Longrightarrow a = \bot \wedge b = \bot$
by *simp*

lemma *Exh-Cprod2*: $\exists\, a\ b.\ z = <a,\ b>$
by (*simp add*: *cpair-eq-pair*)

lemma *cprodE*: $[\![\bigwedge x\ y.\ p = <x,\ y> \Longrightarrow Q]\!] \Longrightarrow Q$
by (*cut-tac Exh-Cprod2*, *auto*)

lemma *cfst-cpair* [*simp*]: $cfst{\cdot}<x,\ y> = x$
by (*simp add*: *cpair-eq-pair cfst-def cont-fst*)

lemma *csnd-cpair* [*simp*]: $csnd{\cdot}<x,\ y> = y$
by (*simp add*: *cpair-eq-pair csnd-def cont-snd*)

lemma *cfst-strict* [*simp*]: $cfst{\cdot}\bot = \bot$
by (*simp add*: *inst-cprod-pcpo2*)

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

by (*simp add*: *inst-cprod-pcpo2*)

lemma *surjective-pairing-Cprod2*: $<cfst{\cdot}p, csnd{\cdot}p> = p$
apply (*unfold cfst-def csnd-def*)
apply (*simp add*: *cont-fst cont-snd cpair-eq-pair*)
done

lemma *less-cprod*: $x \sqsubseteq y = (cfst{\cdot}x \sqsubseteq cfst{\cdot}y \wedge csnd{\cdot}x \sqsubseteq csnd{\cdot}y)$
by (*simp add*: *less-cprod-def cfst-def csnd-def cont-fst cont-snd*)

lemma *eq-cprod*: $(x = y) = (cfst{\cdot}x = cfst{\cdot}y \wedge csnd{\cdot}x = csnd{\cdot}y)$
by (*auto simp add*: *po-eq-conv less-cprod*)

lemma *compact-cpair* [*simp*]: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact\ <x,\ y>$
by (*rule compactI*, *simp add*: *less-cprod*)

lemma *lub-cprod2*:
$chain\ S \Longrightarrow range\ S <<|\ <\bigsqcup i.\ cfst{\cdot}(S\ i),\ \bigsqcup i.\ csnd{\cdot}(S\ i)>$
apply (*simp add*: *cpair-eq-pair cfst-def csnd-def cont-fst cont-snd*)
apply (*erule lub-cprod*)
done

lemma *thelub-cprod2*:
$chain\ S \Longrightarrow lub\ (range\ S) = <\bigsqcup i.\ cfst{\cdot}(S\ i),\ \bigsqcup i.\ csnd{\cdot}(S\ i)>$
by (*rule lub-cprod2* [*THEN thelubI*])

lemma *csplit1* [*simp*]: $csplit{\cdot}f{\cdot}\bot = f{\cdot}\bot{\cdot}\bot$
by (*simp add*: *csplit-def*)

lemma *csplit2* [*simp*]: $csplit{\cdot}f{\cdot}<x,y> = f{\cdot}x{\cdot}y$
by (*simp add*: *csplit-def*)

lemma *csplit3* [*simp*]: $csplit{\cdot}cpair{\cdot}z = z$
by (*simp add*: *csplit-def surjective-pairing-Cprod2*)

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* × '*b*. *p* = ⊥ ∨ (*cfst*·*p* ≠ ⊥ ∧ *csnd*·*p* ≠ ⊥)}
by *simp*

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

lemma *spair-lemma*:
<*strictify*·(Λ *b*. *a*)·*b*, *strictify*·(Λ *a*. *b*)·*a*> ∈ *Sprod*
by (*simp add*: *Sprod-def strictify-conv-if cpair-strict*)

9.2 Definitions of constants

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

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

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

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

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

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

9.3 Case analysis

lemma *spair-Abs-Sprod*:
(:*a*, *b*:) = *Abs-Sprod* <*strictify*·(Λ *b*. *a*)·*b*, *strictify*·(Λ *a*. *b*)·*a*>
apply (*unfold spair-def*)
apply (*simp add*: *cont-Abs-Sprod spair-lemma*)
done

lemma *Exh-Sprod2*:
$z = \bot \vee (\exists\, a\ b.\ z = (:a,\ b:) \wedge a \neq \bot \wedge b \neq \bot)$
apply (*rule-tac x=z* **in** *Abs-Sprod-cases*)
apply (*simp add*: *Sprod-def*)
apply (*erule disjE*)
apply (*simp add*: *Abs-Sprod-strict*)
apply (*rule disjI2*)
apply (*rule-tac x=cfst·y* **in** *exI*)
apply (*rule-tac x=csnd·y* **in** *exI*)
apply (*simp add*: *spair-Abs-Sprod Abs-Sprod-inject spair-lemma*)
apply (*simp add*: *surjective-pairing-Cprod2*)
done

lemma *sprodE*:
$[\![p = \bot \Longrightarrow Q;\ \bigwedge x\ y.\ [\![p = (:x,\ y:);\ x \neq \bot;\ y \neq \bot]\!] \Longrightarrow Q]\!] \Longrightarrow Q$
by (*cut-tac z=p* **in** *Exh-Sprod2*, *auto*)

9.4 Properties of *spair*

lemma *spair-strict1* [*simp*]: $(:\bot,\ y:) = \bot$
by (*simp add*: *spair-Abs-Sprod strictify-conv-if cpair-strict Abs-Sprod-strict*)

lemma *spair-strict2* [*simp*]: $(:x,\ \bot:) = \bot$
by (*simp add*: *spair-Abs-Sprod strictify-conv-if cpair-strict Abs-Sprod-strict*)

lemma *spair-strict*: $x = \bot \vee y = \bot \Longrightarrow (:x,\ y:) = \bot$
by *auto*

lemma *spair-strict-rev*: $(:x,\ y:) \neq \bot \Longrightarrow x \neq \bot \wedge y \neq \bot$
by (*erule contrapos-np*, *auto*)

lemma *spair-defined* [*simp*]:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \neq \bot$
by (*simp add*: *spair-Abs-Sprod Abs-Sprod-defined Sprod-def*)

lemma *spair-defined-rev*: $(:x,\ y:) = \bot \Longrightarrow x = \bot \vee y = \bot$
by (*erule contrapos-pp*, *simp*)

lemma *spair-eq*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow ((:x,\ y:) = (:a,\ b:)) = (x = a \wedge y = b)$
apply (*simp add*: *spair-Abs-Sprod*)
apply (*simp add*: *Abs-Sprod-inject* [*OF - spair-lemma*] *Sprod-def*)
apply (*simp add*: *strictify-conv-if*)
done

lemma *spair-inject*:
$[\![x \neq \bot;\ y \neq \bot;\ (:x,\ y:) = (:a,\ b:)]\!] \Longrightarrow x = a \wedge y = b$
by (*rule spair-eq* [*THEN iffD1*])

lemma *inst-sprod-pcpo2*: $UU = (:UU, UU:)$
by *simp*

lemma *Rep-Sprod-spair*:
Rep-Sprod $(:a,\ b:) = <$*strictify*$\cdot(\Lambda\ b.\ a)\cdot b,$ *strictify*$\cdot(\Lambda\ a.\ b)\cdot a>$
apply (*unfold spair-def*)
apply (*simp add*: *cont-Abs-Sprod Abs-Sprod-inverse spair-lemma*)
done

lemma *compact-spair*: $[\![$*compact* x; *compact* $y]\!] \Longrightarrow$ *compact* $(:x,\ y:)$
by (*rule compact-Sprod*, *simp add*: *Rep-Sprod-spair strictify-conv-if*)

9.5 Properties of *sfst* and *ssnd*

lemma *sfst-strict* [*simp*]: *sfst*$\cdot\bot = \bot$
by (*simp add*: *sfst-def cont-Rep-Sprod Rep-Sprod-strict*)

lemma *ssnd-strict* [*simp*]: *ssnd*$\cdot\bot = \bot$
by (*simp add*: *ssnd-def cont-Rep-Sprod Rep-Sprod-strict*)

lemma *sfst-spair* [*simp*]: $y \neq \bot \Longrightarrow$ *sfst*$\cdot(:x,\ y:) = x$
by (*simp add*: *sfst-def cont-Rep-Sprod Rep-Sprod-spair*)

lemma *ssnd-spair* [*simp*]: $x \neq \bot \Longrightarrow$ *ssnd*$\cdot(:x,\ y:) = y$
by (*simp add*: *ssnd-def cont-Rep-Sprod Rep-Sprod-spair*)

lemma *sfst-defined-iff* [*simp*]: $($*sfst*$\cdot p = \bot) = (p = \bot)$
by (*rule-tac p=p* **in** *sprodE*, *simp-all*)

lemma *ssnd-defined-iff* [*simp*]: $($*ssnd*$\cdot p = \bot) = (p = \bot)$
by (*rule-tac p=p* **in** *sprodE*, *simp-all*)

lemma *sfst-defined*: $p \neq \bot \Longrightarrow$ *sfst*$\cdot p \neq \bot$
by *simp*

lemma *ssnd-defined*: $p \neq \bot \Longrightarrow$ *ssnd*$\cdot p \neq \bot$
by *simp*

lemma *surjective-pairing-Sprod2*: $(:$*sfst*$\cdot p,$ *ssnd*$\cdot p:) = p$
by (*rule-tac p=p* **in** *sprodE*, *simp-all*)

lemma *less-sprod*: $x \sqsubseteq y = ($*sfst*$\cdot x \sqsubseteq$ *sfst*$\cdot y \wedge$ *ssnd*$\cdot x \sqsubseteq$ *ssnd*$\cdot y)$
apply (*simp add*: *less-Sprod-def sfst-def ssnd-def cont-Rep-Sprod*)
apply (*rule less-cprod*)
done

lemma *eq-sprod*: $(x = y) = ($*sfst*$\cdot x =$ *sfst*$\cdot y \wedge$ *ssnd*$\cdot x =$ *ssnd*$\cdot y)$
by (*auto simp add*: *po-eq-conv less-sprod*)

lemma *spair-less*:
  $[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) = (x \sqsubseteq a \wedge y \sqsubseteq b)$
apply (*case-tac* $a = \bot$)
apply (*simp add*: *eq-UU-iff* [*symmetric*])
apply (*case-tac* $b = \bot$)
apply (*simp add*: *eq-UU-iff* [*symmetric*])
apply (*simp add*: *less-sprod*)
done

9.6 Properties of *ssplit*

lemma *ssplit1* [*simp*]: $ssplit{\cdot}f{\cdot}\bot = \bot$
by (*simp add*: *ssplit-def*)

lemma *ssplit2* [*simp*]: $[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow ssplit{\cdot}f{\cdot}(:x,\ y:) = f{\cdot}x{\cdot}y$
by (*simp add*: *ssplit-def*)

lemma *ssplit3* [*simp*]: $ssplit{\cdot}spair{\cdot}z = z$
by (*rule-tac p*=*z* **in** *sprodE*, *simp-all*)

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*) =
    $\{p::'a \times 'b.\ cfst{\cdot}p = \bot \vee csnd{\cdot}p = \bot\}$
by *simp*

syntax (*xsymbols*)
  $++$ :: [*type*, *type*] => *type* ((- $\oplus$/ -) [*21*, *20*] *20*)
syntax (*HTML* **output**)
  $++$ :: [*type*, *type*] => *type* ((- $\oplus$/ -) [*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 :: $'b \to ('a \mathbin{++} 'b)$ **where**
sinr = $(\Lambda\ b.\ \mathit{Abs\text{-}Ssum} <\bot, b>)$

10.3 Properties of *sinl* and *sinr*

lemma *sinl-Abs-Ssum*: $\mathit{sinl}{\cdot}a = \mathit{Abs\text{-}Ssum} <a, \bot>$
by (*unfold sinl-def*, *simp add*: *cont-Abs-Ssum Ssum-def*)

lemma *sinr-Abs-Ssum*: $\mathit{sinr}{\cdot}b = \mathit{Abs\text{-}Ssum} <\bot, b>$
by (*unfold sinr-def*, *simp add*: *cont-Abs-Ssum Ssum-def*)

lemma *Rep-Ssum-sinl*: $\mathit{Rep\text{-}Ssum}\ (\mathit{sinl}{\cdot}a) = <a, \bot>$
by (*unfold sinl-def*, *simp add*: *cont-Abs-Ssum Abs-Ssum-inverse Ssum-def*)

lemma *Rep-Ssum-sinr*: $\mathit{Rep\text{-}Ssum}\ (\mathit{sinr}{\cdot}b) = <\bot, b>$
by (*unfold sinr-def*, *simp add*: *cont-Abs-Ssum Abs-Ssum-inverse Ssum-def*)

lemma *compact-sinl* [*simp*]: $\mathit{compact}\ x \Longrightarrow \mathit{compact}\ (\mathit{sinl}{\cdot}x)$
by (*rule compact-Ssum*, *simp add*: *Rep-Ssum-sinl*)

lemma *compact-sinr* [*simp*]: $\mathit{compact}\ x \Longrightarrow \mathit{compact}\ (\mathit{sinr}{\cdot}x)$
by (*rule compact-Ssum*, *simp add*: *Rep-Ssum-sinr*)

lemma *sinl-strict* [*simp*]: $\mathit{sinl}{\cdot}\bot = \bot$
by (*simp add*: *sinl-Abs-Ssum Abs-Ssum-strict cpair-strict*)

lemma *sinr-strict* [*simp*]: $\mathit{sinr}{\cdot}\bot = \bot$
by (*simp add*: *sinr-Abs-Ssum Abs-Ssum-strict cpair-strict*)

lemma *sinl-eq* [*simp*]: $(\mathit{sinl}{\cdot}x = \mathit{sinl}{\cdot}y) = (x = y)$
by (*simp add*: *sinl-Abs-Ssum Abs-Ssum-inject Ssum-def*)

lemma *sinr-eq* [*simp*]: $(\mathit{sinr}{\cdot}x = \mathit{sinr}{\cdot}y) = (x = y)$
by (*simp add*: *sinr-Abs-Ssum Abs-Ssum-inject Ssum-def*)

lemma *sinl-inject*: $\mathit{sinl}{\cdot}x = \mathit{sinl}{\cdot}y \Longrightarrow x = y$
by (*rule sinl-eq* [*THEN iffD1*])

lemma *sinr-inject*: $\mathit{sinr}{\cdot}x = \mathit{sinr}{\cdot}y \Longrightarrow x = y$
by (*rule sinr-eq* [*THEN iffD1*])

lemma *sinl-defined-iff* [*simp*]: $(\mathit{sinl}{\cdot}x = \bot) = (x = \bot)$
by (*cut-tac sinl-eq* [*of x* $\bot$], *simp*)

lemma *sinr-defined-iff* [*simp*]: $(\mathit{sinr}{\cdot}x = \bot) = (x = \bot)$
by (*cut-tac sinr-eq* [*of x* $\bot$], *simp*)

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

by *simp*

lemma *sinr-defined* [*intro*!]: $x \neq \bot \Longrightarrow sinr{\cdot}x \neq \bot$
by *simp*

10.4 Case analysis

lemma *Exh-Ssum*:
$z = \bot \vee (\exists\, a.\ z = sinl{\cdot}a \wedge a \neq \bot) \vee (\exists\, b.\ z = sinr{\cdot}b \wedge b \neq \bot)$
apply (*rule-tac x=z* **in** *Abs-Ssum-induct*)
apply (*rule-tac p=y* **in** *cprodE*)
apply (*simp add*: *sinl-Abs-Ssum sinr-Abs-Ssum*)
apply (*simp add*: *Abs-Ssum-inject Ssum-def*)
apply (*auto simp add*: *cpair-strict Abs-Ssum-strict*)
done

lemma *ssumE*:
$[\![p = \bot \Longrightarrow Q;$
$\bigwedge x.\ [\![p = sinl{\cdot}x;\ x \neq \bot]\!] \Longrightarrow Q;$
$\bigwedge y.\ [\![p = sinr{\cdot}y;\ y \neq \bot]\!] \Longrightarrow Q]\!] \Longrightarrow Q$
by (*cut-tac z=p* **in** *Exh-Ssum*, *auto*)

lemma *ssumE2*:
$[\![\bigwedge x.\ p = sinl{\cdot}x \Longrightarrow Q;\ \bigwedge y.\ p = sinr{\cdot}y \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*rule-tac p=p* **in** *ssumE*)
apply (*simp only*: *sinl-strict* [*symmetric*])
apply *simp*
apply *simp*
done

10.5 Ordering properties of *sinl* and *sinr*

lemma *sinl-less* [*simp*]: $(sinl{\cdot}x \sqsubseteq sinl{\cdot}y) = (x \sqsubseteq y)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl*)

lemma *sinr-less* [*simp*]: $(sinr{\cdot}x \sqsubseteq sinr{\cdot}y) = (x \sqsubseteq y)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinr*)

lemma *sinl-less-sinr* [*simp*]: $(sinl{\cdot}x \sqsubseteq sinr{\cdot}y) = (x = \bot)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl Rep-Ssum-sinr*)

lemma *sinr-less-sinl* [*simp*]: $(sinr{\cdot}x \sqsubseteq sinl{\cdot}y) = (x = \bot)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl Rep-Ssum-sinr*)

lemma *sinl-eq-sinr* [*simp*]: $(sinl{\cdot}x = sinr{\cdot}y) = (x = \bot \wedge y = \bot)$
by (*subst po-eq-conv*, *simp*)

lemma *sinr-eq-sinl* [*simp*]: $(sinr{\cdot}x = sinl{\cdot}y) = (x = \bot \wedge y = \bot)$
by (*subst po-eq-conv*, *simp*)

10.6 Chains of strict sums

lemma *less-sinlD*: $p \sqsubseteq sinl{\cdot}x \Longrightarrow \exists\, y.\ p = sinl{\cdot}y \wedge y \sqsubseteq x$
apply (*rule-tac p=p* **in** *ssumE*)
apply (*rule-tac x=⊥* **in** *exI*, *simp*)
apply *simp*
apply *simp*
done

lemma *less-sinrD*: $p \sqsubseteq sinr{\cdot}x \Longrightarrow \exists\, y.\ p = sinr{\cdot}y \wedge y \sqsubseteq x$
apply (*rule-tac p=p* **in** *ssumE*)
apply (*rule-tac x=⊥* **in** *exI*, *simp*)
apply *simp*
apply *simp*
done

lemma *ssum-chain-lemma*:
$chain\ Y \Longrightarrow (\exists\, A.\ chain\ A \wedge Y = (\lambda i.\ sinl{\cdot}(A\ i))) \vee$
$\quad (\exists\, B.\ chain\ B \wedge Y = (\lambda i.\ sinr{\cdot}(B\ i)))$
apply (*rule-tac p=lub* (*range Y*) **in** *ssumE2*)
apply (*rule disjI1*)
apply (*rule-tac x=λi. cfst·*(*Rep-Ssum* (*Y i*)) **in** *exI*)
apply (*rule conjI*)
apply (*rule chain-monofun*)
apply (*erule cont-Rep-Ssum* [*THEN ch2ch-cont*])
apply (*rule ext*, *drule-tac x=i* **in** *is-ub-thelub*, *simp*)
apply (*drule less-sinlD*, *clarify*)
apply (*simp add*: *Rep-Ssum-sinl*)
apply (*rule disjI2*)
apply (*rule-tac x=λi. csnd·*(*Rep-Ssum* (*Y i*)) **in** *exI*)
apply (*rule conjI*)
apply (*rule chain-monofun*)
apply (*erule cont-Rep-Ssum* [*THEN ch2ch-cont*])
apply (*rule ext*, *drule-tac x=i* **in** *is-ub-thelub*, *simp*)
apply (*drule less-sinrD*, *clarify*)
apply (*simp add*: *Rep-Ssum-sinr*)
done

10.7 Definitions of constants

definition
Iwhen :: [*'a* → *'c*, *'b* → *'c*, *'a* ++ *'b*] ⇒ *'c* **where**
Iwhen = (λ*f g s*.
if cfst·(*Rep-Ssum s*) ≠ ⊥ *then f·*(*cfst·*(*Rep-Ssum s*)) *else*
if csnd·(*Rep-Ssum s*) ≠ ⊥ *then g·*(*csnd·*(*Rep-Ssum s*)) *else* ⊥)

rewrites for *Iwhen*

lemma *Iwhen1* [*simp*]: *Iwhen f g* ⊥ = ⊥
by (*simp add*: *Iwhen-def Rep-Ssum-strict*)

lemma *Iwhen2* [*simp*]: $x \neq \bot \Longrightarrow Iwhen\ f\ g\ (sinl{\cdot}x) = f{\cdot}x$
by (*simp add*: *Iwhen-def Rep-Ssum-sinl*)

lemma *Iwhen3* [*simp*]: $y \neq \bot \Longrightarrow Iwhen\ f\ g\ (sinr{\cdot}y) = g{\cdot}y$
by (*simp add*: *Iwhen-def Rep-Ssum-sinr*)

lemma *Iwhen4*: $Iwhen\ f\ g\ (sinl{\cdot}x) = strictify{\cdot}f{\cdot}x$
by (*simp add*: *strictify-conv-if*)

lemma *Iwhen5*: $Iwhen\ f\ g\ (sinr{\cdot}y) = strictify{\cdot}g{\cdot}y$
by (*simp add*: *strictify-conv-if*)

10.8 Continuity of *Iwhen*

Iwhen is continuous in all arguments

lemma *cont-Iwhen1*: $cont\ (\lambda f.\ Iwhen\ f\ g\ s)$
by (*rule-tac p=s* **in** *ssumE*, *simp-all*)

lemma *cont-Iwhen2*: $cont\ (\lambda g.\ Iwhen\ f\ g\ s)$
by (*rule-tac p=s* **in** *ssumE*, *simp-all*)

lemma *cont-Iwhen3*: $cont\ (\lambda s.\ Iwhen\ f\ g\ s)$
apply (*rule contI*)
apply (*drule ssum-chain-lemma*, *safe*)
apply (*simp add*: *contlub-cfun-arg* [*symmetric*])
apply (*simp add*: *Iwhen4 cont-cfun-arg*)
apply (*simp add*: *contlub-cfun-arg* [*symmetric*])
apply (*simp add*: *Iwhen5 cont-cfun-arg*)
done

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* | *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*: $sscase{\cdot}f{\cdot}g{\cdot}s = Iwhen\ f\ g\ s$
by (*simp add*: *sscase-def cont-Iwhen1 cont-Iwhen2 cont-Iwhen3*)

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

by (*simp add*: *beta-sscase*)

lemma *sscase2* [*simp*]: $x \neq \bot \Longrightarrow sscase{\cdot}f{\cdot}g{\cdot}(sinl{\cdot}x) = f{\cdot}x$
by (*simp add*: *beta-sscase*)

lemma *sscase3* [*simp*]: $y \neq \bot \Longrightarrow sscase{\cdot}f{\cdot}g{\cdot}(sinr{\cdot}y) = g{\cdot}y$
by (*simp add*: *beta-sscase*)

lemma *sscase4* [*simp*]: $sscase{\cdot}sinl{\cdot}sinr{\cdot}z = z$
by (*rule-tac p*=*z* **in** *ssumE*, *simp-all*)

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

primrec
$Ifup\ f\ Ibottom = \bot$
$Ifup\ f\ (Iup\ x) = f{\cdot}x$

11.2 Ordering on lifted cpo

instance $u :: (sq\text{-}ord)\ sq\text{-}ord$ **..**

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*]: $Ibottom \sqsubseteq z$
by (*simp add*: *less-up-def*)

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

by (*simp add*: *less-up-def*)

lemma *Iup-less* [*iff*]: (*Iup x* ⊑ *Iup y*) = (*x* ⊑ *y*)
by (*simp add*: *less-up-def*)

11.3 Lifted cpo is a partial order

lemma *refl-less-up*: (*x*::'*a u*) ⊑ *x*
by (*simp add*: *less-up-def split*: *u.split*)

lemma *antisym-less-up*: ⟦(*x*::'*a u*) ⊑ *y*; *y* ⊑ *x*⟧ ⟹ *x* = *y*
apply (*simp add*: *less-up-def split*: *u.split-asm*)
apply (*erule* (*1*) *antisym-less*)
done

lemma *trans-less-up*: ⟦(*x*::'*a u*) ⊑ *y*; *y* ⊑ *z*⟧ ⟹ *x* ⊑ *z*
apply (*simp add*: *less-up-def split*: *u.split-asm*)
apply (*erule* (*1*) *trans-less*)
done

instance *u* :: (*cpo*) *po*
by *intro-classes*
(*assumption* | *rule refl-less-up antisym-less-up trans-less-up*)+

11.4 Lifted cpo is a cpo

lemma *is-lub-Iup*:
range S <<| *x* ⟹ *range* (λ*i*. *Iup* (*S i*)) <<| *Iup x*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*subst Iup-less*)
apply (*erule is-ub-lub*)
apply (*case-tac u*)
apply (*drule ub-rangeD*)
apply *simp*
apply *simp*
apply (*erule is-lub-lub*)
apply (*rule ub-rangeI*)
apply (*drule-tac i*=*i* **in** *ub-rangeD*)
apply *simp*
done

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

lemma *up-lemma1*: *z* ≠ *Ibottom* ⟹ *Iup* (*THE a*. *Iup a* = *z*) = *z*
by (*case-tac z*, *simp-all*)

lemma *up-lemma2*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹ *Y* (*i* + *j*) ≠ *Ibottom*
apply (*erule contrapos-nn*)

apply (*drule-tac x=j* **and** *y=i + j* **in** *chain-mono3*)
apply (*rule le-add2*)
apply (*case-tac Y j*)
apply *assumption*
apply *simp*
done

lemma *up-lemma3*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹ *Iup* (*THE a*. *Iup a* = *Y* (*i* + *j*)) = *Y* (*i* + *j*)
by (*rule up-lemma1* [*OF up-lemma2*])

lemma *up-lemma4*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹ *chain* (λ*i*. *THE a*. *Iup a* = *Y* (*i* + *j*))
apply (*rule chainI*)
apply (*rule Iup-less* [*THEN iffD1*])
apply (*subst up-lemma3*, *assumption*+)+
apply (*simp add*: *chainE*)
done

lemma *up-lemma5*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹
(λ*i*. *Y* (*i* + *j*)) = (λ*i*. *Iup* (*THE a*. *Iup a* = *Y* (*i* + *j*)))
by (*rule ext*, *rule up-lemma3* [*symmetric*])

lemma *up-lemma6*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧
⟹ *range Y* <<| *Iup* (⨆ *i*. *THE a*. *Iup a* = *Y*(*i* + *j*))
apply (*rule-tac j1* = *j* **in** *is-lub-range-shift* [*THEN iffD1*])
apply *assumption*
apply (*subst up-lemma5*, *assumption*+)
apply (*rule is-lub-Iup*)
apply (*rule thelubE* [*OF - refl*])
apply (*erule* (*1*) *up-lemma4*)
done

lemma *up-chain-lemma*:
chain Y ⟹
(∃ *A*. *chain A* ∧ *lub* (*range Y*) = *Iup* (*lub* (*range A*)) ∧
(∃ *j*. ∀ *i*. *Y* (*i* + *j*) = *Iup* (*A i*))) ∨ (*Y* = (λ*i*. *Ibottom*))
apply (*rule disjCI*)
apply (*simp add*: *expand-fun-eq*)
apply (*erule exE*, *rename-tac j*)
apply (*rule-tac x=*λ*i*. *THE a*. *Iup a* = *Y* (*i* + *j*) **in** *exI*)
apply (*simp add*: *up-lemma4*)
apply (*simp add*: *up-lemma6* [*THEN thelubI*])
apply (*rule-tac x=j* **in** *exI*)
apply (*simp add*: *up-lemma3*)
done

lemma *cpo-up*: *chain* (*Y*::*nat* ⇒ '*a u*) ⟹ ∃ *x*. *range Y* <<| *x*
apply (*frule up-chain-lemma*, *safe*)
apply (*rule-tac x*=*Iup* (*lub* (*range A*)) **in** *exI*)
apply (*erule-tac j*=*j* **in** *is-lub-range-shift* [*THEN iffD1*, *standard*])
apply (*simp add*: *is-lub-Iup thelubE*)
apply (*rule exI*, *rule lub-const*)
done

instance *u* :: (*cpo*) *cpo*
by *intro-classes* (*rule cpo-up*)

11.5 Lifted cpo is pointed

lemma *least-up*: ∃ *x*::'*a u*. ∀ *y*. *x* ⊑ *y*
apply (*rule-tac x* = *Ibottom* **in** *exI*)
apply (*rule minimal-up* [*THEN allI*])
done

instance *u* :: (*cpo*) *pcpo*
by *intro-classes* (*rule least-up*)

for compatibility with old HOLCF-Version

lemma *inst-up-pcpo*: ⊥ = *Ibottom*
by (*rule minimal-up* [*THEN UU-I*, *symmetric*])

11.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

lemma *cont-Iup*: *cont Iup*
apply (*rule contI*)
apply (*rule is-lub-Iup*)
apply (*erule thelubE* [*OF - refl*])
done

continuity for *Ifup*

lemma *cont-Ifup1*: *cont* (λ*f*. *Ifup f x*)
by (*induct x*, *simp-all*)

lemma *monofun-Ifup2*: *monofun* (λ*x*. *Ifup f x*)
apply (*rule monofunI*)
apply (*case-tac x*, *simp*)
apply (*case-tac y*, *simp*)
apply (*simp add*: *monofun-cfun-arg*)
done

lemma *cont-Ifup2*: *cont* (λ*x*. *Ifup f x*)
apply (*rule contI*)
apply (*frule up-chain-lemma*, *safe*)

apply (*rule-tac j=j* **in** *is-lub-range-shift* [*THEN iffD1*, *standard*])
apply (*erule monofun-Ifup2* [*THEN ch2ch-monofun*])
apply (*simp add*: *cont-cfun-arg*)
apply (*simp add*: *lub-const*)
done

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*: $z = \perp \vee (\exists\, x.\ z = up\cdot x)$
apply (*induct z*)
apply (*simp add*: *inst-up-pcpo*)
apply (*simp add*: *up-def cont-Iup*)
done

lemma *up-eq* [*simp*]: $(up\cdot x = up\cdot y) = (x = y)$
by (*simp add*: *up-def cont-Iup*)

lemma *up-inject*: $up\cdot x = up\cdot y \Longrightarrow x = y$
by *simp*

lemma *up-defined* [*simp*]: $up\cdot x \neq \perp$
by (*simp add*: *up-def cont-Iup inst-up-pcpo*)

lemma *not-up-less-UU* [*simp*]: $\neg\ up\cdot x \sqsubseteq \perp$
by (*simp add*: *eq-UU-iff* [*symmetric*])

lemma *up-less* [*simp*]: $(up\cdot x \sqsubseteq up\cdot y) = (x \sqsubseteq y)$
by (*simp add*: *up-def cont-Iup*)

lemma *upE*: $[\![p = \perp \Longrightarrow Q;\ \bigwedge x.\ p = up\cdot x \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*case-tac p*)
apply (*simp add*: *inst-up-pcpo*)
apply (*simp add*: *up-def cont-Iup*)
done

lemma *up-chain-cases*:

chain Y $\Longrightarrow$
($\exists A.$ *chain A* $\wedge$ ($\bigsqcup i.$ *Y i*) = *up*·($\bigsqcup i.$ *A i*) $\wedge$
($\exists j. \forall i.$ *Y* (*i* + *j*) = *up*·(*A i*))) $\vee$ *Y* = ($\lambda i. \bot$)
by (*simp add*: *inst-up-pcpo up-def cont-Iup up-chain-lemma*)

lemma *compact-up* [*simp*]: *compact x* $\Longrightarrow$ *compact* (*up*·*x*)
apply (*unfold compact-def*)
apply (*rule admI*)
apply (*drule up-chain-cases*)
apply (*elim disjE exE conjE*)
apply *simp*
apply (*erule* (*1*) *admD*)
apply (*rule allI*, *drule-tac x*=*i* + *j* **in** *spec*)
apply *simp*
apply *simp*
done

properties of fup

lemma *fup1* [*simp*]: *fup*·*f*·$\bot$ = $\bot$
by (*simp add*: *fup-def cont-Ifup1 cont-Ifup2 inst-up-pcpo*)

lemma *fup2* [*simp*]: *fup*·*f*·(*up*·*x*) = *f*·*x*
by (*simp add*: *up-def fup-def cont-Iup cont-Ifup1 cont-Ifup2*)

lemma *fup3* [*simp*]: *fup*·*up*·*x* = *x*
by (*rule-tac p*=*x* **in** *upE*, *simp-all*)

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* **..**

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*)
by (*unfold less-discr-def*) (*rule refl*)

instance *discr* :: (*type*) *po*

proof
fix $x\ y\ z :: 'a\ discr$
show $x << x$ **by** *simp*
{ assume $x << y$ **and** $y << x$ **thus** $x = y$ **by** *simp* **}**
{ assume $x << y$ **and** $y << z$ **thus** $x << z$ **by** *simp* **}**
qed

12.2 Type *'a discr* is a cpo

lemma *discr-chain0*:
$!!S{::}nat{=>}('a{::}type)discr.\ chain\ S ==> S\ i = S\ 0$
apply (*unfold chain-def*)
apply (*induct-tac i*)
apply (*rule refl*)
apply (*erule subst*)
apply (*rule sym*)
apply *fast*
done

lemma *discr-chain-range0* [*simp*]:
$!!S{::}nat{=>}('a{::}type)discr.\ chain(S) ==> range(S) = \{S\ 0\}$
by (*fast elim*: *discr-chain0*)

lemma *discr-cpo*:
$!!S.\ chain\ S ==> ?\ x{::}('a{::}type)discr.\ range(S) <<|\ x$
by (*unfold is-lub-def is-ub-def*) *simp*

instance *discr* :: (*type*) *cpo*
by *intro-classes* (*rule discr-cpo*)

12.3 *undiscr*

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

lemma *undiscr-Discr* [*simp*]: $undiscr(Discr\ x) = x$
by (*simp add*: *undiscr-def*)

lemma *discr-chain-f-range0*:
$!!S{::}nat{=>}('a{::}type)discr.\ chain(S) ==> range(\%i.\ f(S\ i)) = \{f(S\ 0)\}$
by (*fast dest*: *discr-chain0 elim*: *arg-cong*)

lemma *cont-discr* [*iff*]: $cont(\%x{::}('a{::}type)discr.\ f\ x)$
apply (*unfold cont-def is-lub-def is-ub-def*)
apply (*simp add*: *discr-chain-f-range0*)
done

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*
by *simp*

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*: $\bot \neq Def\ x$
by (*simp add*: *Def-def Abs-lift-inject lift-def inst-lift-pcpo*)

lemma *lift-distinct2*: $Def\ x \neq \bot$
by (*simp add*: *Def-def Abs-lift-inject lift-def inst-lift-pcpo*)

lemma *Def-inject*: $(Def\ x = Def\ y) = (x = y)$
by (*simp add*: *Def-def Abs-lift-inject lift-def*)

lemma *lift-induct*: $[\![P\ \bot;\ \bigwedge x.\ P\ (Def\ x)]\!] \Longrightarrow P\ y$
apply (*induct y*)
apply (*rule-tac p=y* **in** *upE*)
apply (*simp add*: *Abs-lift-strict*)
apply (*case-tac x*)
apply (*simp add*: *Def-def*)
done

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

lemma *Def-not-UU*: $Def\ a \neq UU$
by *simp*

$\bot$ and *Def*

lemma *Lift-exhaust*: $x = \bot \vee (\exists y.\ x = Def\ y)$
by (*induct x*) *simp-all*

lemma *Lift-cases*: $[\![x = \bot \Longrightarrow P;\ \exists a.\ x = Def\ a \Longrightarrow P]\!] \Longrightarrow P$
by (*insert Lift-exhaust*) *blast*

lemma *not-Undef-is-Def*: $(x \neq \bot) = (\exists\, y.\ x = Def\ y)$
by (*cases x*) *simp-all*

lemma *lift-definedE*: $[\![x \neq \bot;\ \bigwedge a.\ x = Def\ a \Longrightarrow R]\!] \Longrightarrow R$
by (*cases x*) *simp-all*

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

ML ⟪
local val lift-definedE = thm lift-definedE
in val def-tac = SIMPSET' (fn ss =>
etac lift-definedE THEN' asm-simp-tac ss)
end;
⟫

lemma *DefE*: $Def\ x = \bot \Longrightarrow R$
by *simp*

lemma *DefE2*: $[\![x = Def\ s;\ x = \bot]\!] \Longrightarrow R$
by *simp*

lemma *Def-inject-less-eq*: $Def\ x \sqsubseteq Def\ y = (x = y)$
by (*simp add*: *less-lift-def Def-def Abs-lift-inverse lift-def*)

lemma *Def-less-is-eq* [*simp*]: $Def\ x \sqsubseteq y = (Def\ x = y)$
apply (*induct y*)
apply *simp*
apply (*simp add*: *Def-inject-less-eq*)
done

13.2 Lift is flat

lemma *less-lift*: $(x::'a\ lift) \sqsubseteq y = (x = y \vee x = \bot)$
by (*induct x*, *simp-all*)

instance *lift* :: (*type*) *flat*
by (*intro-classes*, *simp add*: *less-lift*)

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

lemma *cont-Rep-CFun-app*: $[\![cont\ g;\ cont\ f]\!] \Longrightarrow cont(\lambda x.\ ((f\ x)\cdot(g\ x))\ s)$
by (*rule cont2cont-Rep-CFun* [*THEN cont2cont-fun*])

lemma *cont-Rep-CFun-app-app*: $[\![cont\ g;\ cont\ f]\!] \Longrightarrow cont(\lambda x.\ ((f\ x)\cdot(g\ x))\ s\ t)$
by (*rule cont-Rep-CFun-app* [*THEN cont2cont-fun*])

13.3 Further operations

definition
flift1 :: $('a \Rightarrow 'b::pcpo) \Rightarrow ('a\ lift \to 'b)$ (**binder** *FLIFT* *10*) **where**

flift1 = $(\lambda f.\ (\Lambda\ x.\ \textit{lift-case}\ \bot\ f\ x))$

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

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

13.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

lemma *cont-lift-case1*: *cont* $(\lambda f.\ \textit{lift-case}\ a\ f\ x)$
apply (*induct x*)
apply *simp*
apply *simp*
apply (*rule cont-id* [*THEN cont2cont-fun*])
done

lemma *cont-lift-case2*: *cont* $(\lambda x.\ \textit{lift-case}\ \bot\ f\ x)$
apply (*rule flatdom-strict2cont*)
apply *simp*
done

lemma *cont-flift1*: *cont flift1*
apply (*unfold flift1-def*)
apply (*rule cont2cont-LAM*)
apply (*rule cont-lift-case2*)
apply (*rule cont-lift-case1*)
done

lemma *cont2cont-flift1*:
$[\![\bigwedge y.\ \textit{cont}\ (\lambda x.\ f\ x\ y)]\!] \Longrightarrow \textit{cont}\ (\lambda x.\ \textit{FLIFT}\ y.\ f\ x\ y)$
apply (*rule cont-flift1* [*THEN cont2cont-app3*])
apply (*simp add*: *cont2cont-lambda*)
done

lemma *cont2cont-lift-case*:
$[\![\bigwedge y.\ \textit{cont}\ (\lambda x.\ f\ x\ y);\ \textit{cont}\ g]\!] \Longrightarrow \textit{cont}\ (\lambda x.\ \textit{lift-case}\ \textit{UU}\ (f\ x)\ (g\ x))$
apply (*subgoal-tac cont* $(\lambda x.\ (\textit{FLIFT}\ y.\ f\ x\ y)\cdot(g\ x)))$
apply (*simp add*: *flift1-def cont-lift-case2*)
apply (*simp add*: *cont2cont-flift1*)
done

rewrites for *flift1*, *flift2*

lemma *flift1-Def* [*simp*]: *flift1* $f\cdot(\textit{Def}\ x) = (f\ x)$
by (*simp add*: *flift1-def cont-lift-case2*)

lemma *flift2-Def* [*simp*]: *flift2 f*·(*Def x*) = *Def* (*f x*)
by (*simp add*: *flift2-def*)

lemma *flift1-strict* [*simp*]: *flift1 f*·⊥ = ⊥
by (*simp add*: *flift1-def cont-lift-case2*)

lemma *flift2-strict* [*simp*]: *flift2 f*·⊥ = ⊥
by (*simp add*: *flift2-def*)

lemma *flift2-defined* [*simp*]: *x* ≠ ⊥ ⟹ (*flift2 f*)·*x* ≠ ⊥
by (*erule lift-definedE*, *simp*)

lemma *flift2-defined-iff* [*simp*]: (*flift2 f*·*x* = ⊥) = (*x* = ⊥)
by (*cases x*, *simp-all*)

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 ⟪
local
val cont-lemmas2 = *thms cont-lemmas1* @ *thms cont-lemmas-ext*;
val flift1-def = *thm flift1-def*;
in

fun cont-tac *i* = *resolve-tac cont-lemmas2 i*;
fun cont-tacR i = *REPEAT* (*cont-tac i*);

fun cont-tacRs ss i =
simp-tac ss i THEN
REPEAT (*cont-tac i*)
end;
⟫

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*: $t = \bot \vee t = ONE$
apply (*unfold ONE-def*)
apply (*induct t*)
apply *simp*
apply *simp*
done

lemma *oneE*: $[\![p = \bot \Longrightarrow Q;\ p = ONE \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*rule Exh-one* [*THEN disjE*])
apply *fast*
apply *fast*
done

lemma *dist-less-one* [*simp*]: $\neg\ ONE \sqsubseteq \bot$
apply (*unfold ONE-def*)
apply *simp*
done

lemma *dist-eq-one* [*simp*]: $ONE \neq \bot\ \bot \neq ONE$
apply (*unfold ONE-def*)
apply *simp-all*
done

lemma *compact-ONE* [*simp*]: *compact ONE*
by (*rule compact-chfin*)

Case analysis function for type *one*

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

translations
case x of CONST ONE $\Rightarrow$ *t* == *CONST one-when*·*t*·*x*
Λ (*CONST ONE*). *t* == *CONST one-when*·*t*

lemma *one-when1* [*simp*]: (*case* $\bot$ *of ONE* $\Rightarrow$ *t*) = $\bot$
by (*simp add*: *one-when-def*)

lemma *one-when2* [*simp*]: (*case ONE of ONE* $\Rightarrow$ *t*) = *t*
by (*simp add*: *one-when-def*)

lemma *one-when3* [*simp*]: (*case x of ONE* $\Rightarrow$ *ONE*) = *x*
by (*rule-tac p*=*x* **in** *oneE*, *simp-all*)

end

15 Tr: The type of lifted booleans

theory *Tr*
imports *Lift*
begin

defaultsort *pcpo*

types
tr = bool lift

translations
tr <= (type) bool lift

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

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

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

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

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

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

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

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

Exhaustion and Elimination for type *tr*

lemma *Exh-tr*: *t* = ⊥ ∨ *t* = *TT* ∨ *t* = *FF*
apply (*unfold FF-def TT-def*)
apply (*induct t*)
apply *fast*
apply *fast*
done

lemma *trE*: ⟦*p* = ⊥ ⟹ *Q*; *p* = *TT* ⟹ *Q*; *p* = *FF* ⟹ *Q*⟧ ⟹ *Q*
apply (*rule Exh-tr* [*THEN disjE*])
apply *fast*
apply (*erule disjE*)
apply *fast*
apply *fast*
done

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*]:
¬ *TT* ⊑ ⊥ ¬ *FF* ⊑ ⊥ ¬ *TT* ⊑ *FF* ¬ *FF* ⊑ *TT*
by (*simp-all add*: *tr-defs*)

lemma *dist-eq-tr* [*simp*]:
TT ≠ ⊥ *FF* ≠ ⊥ *TT* ≠ *FF* ⊥ ≠ *TT* ⊥ ≠ *FF FF* ≠ *TT*
by (*simp-all add*: *tr-defs*)

lemmas about andalso, orelse, neg and if

lemma *ifte-thms* [*simp*]:
If ⊥ *then e1 else e2 fi* = ⊥
If FF then e1 else e2 fi = *e2*
If TT then e1 else e2 fi = *e1*
by (*simp-all add*: *ifte-def TT-def FF-def*)

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

(*y andalso TT*) = *y*
(*y andalso y*) = *y*
apply (*unfold andalso-def*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)
done

lemma *orelse-thms* [*simp*]:
(*TT orelse y*) = *TT*
(*FF orelse y*) = *y*
(⊥ *orelse y*) = ⊥
(*y orelse FF*) = *y*
(*y orelse y*) = *y*
apply (*unfold orelse-def*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)
done

lemma *neg-thms* [*simp*]:
neg·*TT* = *FF*
neg·*FF* = *TT*
neg·⊥ = ⊥
by (*simp-all add*: *neg-def TT-def FF-def*)

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

lemma *split-If2*:
P (*If2 Q x y*) = ((*Q* = ⊥ ⟶ *P* ⊥) ∧ (*Q* = *TT* ⟶ *P x*) ∧ (*Q* = *FF* ⟶ *P y*))
apply (*unfold If2-def*)
apply (*rule-tac p* = *Q* **in** *trE*)
apply (*simp-all*)
done

ML ⟪
val split-If-tac =
simp-tac (*HOL-basic-ss addsimps* [@{*thm If2-def*} *RS sym*])
THEN' (*split-tac* [@{*thm split-If2*}])
⟫

15.1 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*:
t ≠ ⊥ ⟹ ((*t andalso s*) = *FF*) = (*t* = *FF* ∨ *s* = *FF*)
apply (*rule-tac p* = *t* **in** *trE*)
apply *simp-all*
done

lemma *andalso-and*:
t ≠ ⊥ ⟹ ((*t andalso s*) ≠ *FF*) = (*t* ≠ *FF* ∧ *s* ≠ *FF*)

apply (*rule-tac* $p = t$ **in** *trE*)
apply *simp-all*
done

lemma *Def-bool1* [*simp*]: $(Def\ x \neq FF) = x$
by (*simp add*: *FF-def*)

lemma *Def-bool2* [*simp*]: $(Def\ x = FF) = (\neg\ x)$
by (*simp add*: *FF-def*)

lemma *Def-bool3* [*simp*]: $(Def\ x = TT) = x$
by (*simp add*: *TT-def*)

lemma *Def-bool4* [*simp*]: $(Def\ x \neq TT) = (\neg\ x)$
by (*simp add*: *TT-def*)

lemma *If-and-if*:
$(If\ Def\ P\ then\ A\ else\ B\ fi) = (if\ P\ then\ A\ else\ B)$
apply (*rule-tac* $p = Def\ P$ **in** *trE*)
apply (*auto simp add*: *TT-def*[*symmetric*] *FF-def*[*symmetric*])
done

15.2 Compactness

lemma *compact-TT* [*simp*]: *compact TT*
by (*rule compact-chfin*)

lemma *compact-FF* [*simp*]: *compact FF*
by (*rule compact-chfin*)

end

16 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun Cprod Adm*
begin

defaultsort *pcpo*

16.1 Iteration

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

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* $0{\cdot}F{\cdot}x = x$
by *simp*

lemma *iterate-Suc* [*simp*]: *iterate* $(Suc\ n){\cdot}F{\cdot}x = F{\cdot}(iterate\ n{\cdot}F{\cdot}x)$
by *simp*

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

lemma *iterate-Suc2*: *iterate* $(Suc\ n){\cdot}F{\cdot}x = iterate\ n{\cdot}F{\cdot}(F{\cdot}x)$
by (*induct-tac n*, *auto*)

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

lemma *chain-iterate2*: $x \sqsubseteq F{\cdot}x \Longrightarrow chain\ (\lambda i.\ iterate\ i{\cdot}F{\cdot}x)$
by (*rule chainI*, *induct-tac i*, *auto elim*: *monofun-cfun-arg*)

lemma *chain-iterate* [*simp*]: *chain* $(\lambda i.\ iterate\ i{\cdot}F{\cdot}\bot)$
by (*rule chain-iterate2* [*OF minimal*])

16.2 Least fixed point operator

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

Binder syntax for *fix*

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

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

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

Properties of *fix*

direct connection between *fix* and iteration

lemma *fix-def2*: $fix{\cdot}F = (\bigsqcup i.\ iterate\ i{\cdot}F{\cdot}\bot)$
apply (*unfold fix-def*)
apply (*rule beta-cfun*)
apply (*rule cont2cont-lub*)
apply (*rule ch2ch-lambda*)
apply (*rule chain-iterate*)
apply *simp*
done

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

lemma *fix-eq*: $fix{\cdot}F = F{\cdot}(fix{\cdot}F)$
apply (*simp add*: *fix-def2*)
apply (*subst lub-range-shift* [*of - 1*, *symmetric*])
apply (*rule chain-iterate*)
apply (*subst contlub-cfun-arg*)
apply (*rule chain-iterate*)
apply *simp*
done

lemma *fix-least-less*: $F{\cdot}x \sqsubseteq x \Longrightarrow fix{\cdot}F \sqsubseteq x$
apply (*simp add*: *fix-def2*)
apply (*rule is-lub-thelub*)
apply (*rule chain-iterate*)
apply (*rule ub-rangeI*)
apply (*induct-tac i*)
apply *simp*
apply *simp*
apply (*erule rev-trans-less*)
apply (*erule monofun-cfun-arg*)
done

lemma *fix-least*: $F{\cdot}x = x \Longrightarrow fix{\cdot}F \sqsubseteq x$
by (*rule fix-least-less*, *simp*)

lemma *fix-eqI*: $[\![F{\cdot}x = x; \forall z.\ F{\cdot}z = z \longrightarrow x \sqsubseteq z]\!] \Longrightarrow x = fix{\cdot}F$
apply (*rule antisym-less*)
apply (*simp add*: *fix-eq* [*symmetric*])
apply (*erule fix-least*)
done

lemma *fix-eq2*: $f \equiv fix{\cdot}F \Longrightarrow f = F{\cdot}f$
by (*simp add*: *fix-eq* [*symmetric*])

lemma *fix-eq3*: $f \equiv fix{\cdot}F \Longrightarrow f{\cdot}x = F{\cdot}f{\cdot}x$
by (*erule fix-eq2* [*THEN cfun-fun-cong*])

lemma *fix-eq4*: $f = fix{\cdot}F \Longrightarrow f = F{\cdot}f$
apply (*erule ssubst*)
apply (*rule fix-eq*)
done

lemma *fix-eq5*: $f = fix{\cdot}F \Longrightarrow f{\cdot}x = F{\cdot}f{\cdot}x$
by (*erule fix-eq4* [*THEN cfun-fun-cong*])

strictness of *fix*

lemma *fix-defined-iff*: $(fix{\cdot}F = \bot) = (F{\cdot}\bot = \bot)$
apply (*rule iffI*)

apply (*erule subst*)
apply (*rule fix-eq* [*symmetric*])
apply (*erule fix-least* [*THEN UU-I*])
done

lemma *fix-strict*: $F{\cdot}\bot = \bot \Longrightarrow fix{\cdot}F = \bot$
by (*simp add*: *fix-defined-iff*)

lemma *fix-defined*: $F{\cdot}\bot \neq \bot \Longrightarrow fix{\cdot}F \neq \bot$
by (*simp add*: *fix-defined-iff*)

fix applied to identity and constant functions

lemma *fix-id*: $(\mu\ x.\ x) = \bot$
by (*simp add*: *fix-strict*)

lemma *fix-const*: $(\mu\ x.\ c) = c$
by (*subst fix-eq*, *simp*)

16.3 Fixed point induction

lemma *fix-ind*: $[\![adm\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F{\cdot}x)]\!] \Longrightarrow P\ (fix{\cdot}F)$
apply (*subst fix-def2*)
apply (*erule admD* [*rule-format*])
apply (*rule chain-iterate*)
apply (*induct-tac i*, *simp-all*)
done

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

16.4 Recursive let bindings

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

nonterminals
recbinds recbindt recbind

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

translations

(recbindt) x = a, ⟨y,ys⟩ = ⟨b,bs⟩ == (recbindt) ⟨x,y,ys⟩ = ⟨a,b,bs⟩
(recbindt) x = a, y = b == (recbindt) ⟨x,y⟩ = ⟨a,b⟩

translations
-Letrec (-recbinds b bs) e == -Letrec b (-Letrec bs e)
Letrec xs = a in ⟨e,es⟩ == CONST CLetrec·(Λ xs. ⟨a,e,es⟩)
Letrec xs = a in e == CONST CLetrec·(Λ xs. ⟨a,e⟩)

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

lemma *fix-cprod*:
fix·(F::'a × 'b → 'a × 'b) =
⟨μ x. cfst·(F·⟨x, μ y. csnd·(F·⟨x, y⟩)⟩),
μ y. csnd·(F·⟨μ x. cfst·(F·⟨x, μ y. csnd·(F·⟨x, y⟩)⟩), y⟩)⟩
(**is** *fix·F = ⟨?x, ?y⟩*)
proof (*rule fix-eqI* [*rule-format*, *symmetric*])
have *1*: *cfst·(F·⟨?x, ?y⟩) = ?x*
by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
have *2*: *csnd·(F·⟨?x, ?y⟩) = ?y*
by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
from *1 2* **show** *F·⟨?x, ?y⟩ = ⟨?x, ?y⟩* **by** (*simp add*: *eq-cprod*)
next
fix *z* **assume** *F-z*: *F·z = z*
then obtain *x y* **where** *z*: *z = ⟨x,y⟩* **by** (*rule-tac p=z* **in** *cprodE*)
from *F-z z* **have** *F-x*: *cfst·(F·⟨x, y⟩) = x* **by** *simp*
from *F-z z* **have** *F-y*: *csnd·(F·⟨x, y⟩) = y* **by** *simp*
let *?y1 = μ y. csnd·(F·⟨x, y⟩)*
have *?y1 ⊑ y* **by** (*rule fix-least*, *simp add*: *F-y*)
hence *cfst·(F·⟨x, ?y1⟩) ⊑ cfst·(F·⟨x, y⟩)* **by** (*simp add*: *monofun-cfun*)
hence *cfst·(F·⟨x, ?y1⟩) ⊑ x* **using** *F-x* **by** *simp*
hence *1*: *?x ⊑ x* **by** (*simp add*: *fix-least-less*)
hence *csnd·(F·⟨?x, y⟩) ⊑ csnd·(F·⟨x, y⟩)* **by** (*simp add*: *monofun-cfun*)
hence *csnd·(F·⟨?x, y⟩) ⊑ y* **using** *F-y* **by** *simp*
hence *2*: *?y ⊑ y* **by** (*simp add*: *fix-least-less*)
show *⟨?x, ?y⟩ ⊑ z* **using** *z 1 2* **by** *simp*
qed

16.5 Weak admissibility

definition
admw :: (*'a* ⇒ *bool*) ⇒ *bool* **where**
admw P = (∀ F. (∀ n. P (iterate n·F·⊥)) ⟶ P (⨆ i. iterate i·F·⊥))

an admissible formula is also weak admissible

lemma *adm-impl-admw*: *adm P ⟹ admw P*
apply (*unfold admw-def*)
apply (*intro strip*)
apply (*erule admD*)
apply (*rule chain-iterate*)

apply *assumption*
done

computational induction for weak admissible formulae

lemma *wfix-ind*: $[\![adm\!w\ P; \forall\, n.\ P\ (iterate\ n{\cdot}F{\cdot}\bot)]\!] \Longrightarrow P\ (fix{\cdot}F)$
by (*simp add*: *fix-def2 admw-def*)

lemma *def-wfix-ind*:
$[\![f \equiv fix{\cdot}F;\ adm\!w\ P; \forall\, n.\ P\ (iterate\ n{\cdot}F{\cdot}\bot)]\!] \Longrightarrow P\ f$
by (*simp*, *rule wfix-ind*)

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*
by *simp*

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

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

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

lemma *maybeE*:
$[\![p = \bot \Longrightarrow Q;\ p = fail \Longrightarrow Q;\ \bigwedge x.\ p = return{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*unfold fail-def return-def*)
apply (*cases p*, *rename-tac r*)
apply (*rule-tac p*=*r* **in** *ssumE*, *simp add*: *Abs-maybe-strict*)
apply (*rule-tac p*=*x* **in** *oneE*, *simp*, *simp*)
apply (*rule-tac p*=*y* **in** *upE*, *simp*, *simp add*: *cont-Abs-maybe*)

done

lemma *return-defined* [*simp*]: $return{\cdot}x \neq \bot$
by (*simp add*: *return-def cont-Abs-maybe Abs-maybe-defined*)

lemma *fail-defined* [*simp*]: $fail \neq \bot$
by (*simp add*: *fail-def Abs-maybe-defined*)

lemma *return-eq* [*simp*]: $(return{\cdot}x = return{\cdot}y) = (x = y)$
by (*simp add*: *return-def cont-Abs-maybe Abs-maybe-inject*)

lemma *return-neq-fail* [*simp*]:
$return{\cdot}x \neq fail\ fail \neq return{\cdot}x$
by (*simp-all add*: *return-def fail-def cont-Abs-maybe Abs-maybe-inject*)

lemma *maybe-when-rews* [*simp*]:
$maybe\text{-}when{\cdot}f{\cdot}r{\cdot}\bot = \bot$
$maybe\text{-}when{\cdot}f{\cdot}r{\cdot}fail = f$
$maybe\text{-}when{\cdot}f{\cdot}r{\cdot}(return{\cdot}x) = r{\cdot}x$
by (*simp-all add*: *return-def fail-def maybe-when-def cont-Rep-maybe*
cont-Abs-maybe Abs-maybe-inverse Rep-maybe-strict)

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*]: $bind{\cdot}\bot{\cdot}f = \bot$
by (*simp add*: *bind-def*)

lemma *bind-fail* [*simp*]: $bind{\cdot}fail{\cdot}f = fail$
by (*simp add*: *bind-def*)

lemma *left-unit* [*simp*]: $bind{\cdot}(return{\cdot}a){\cdot}k = k{\cdot}a$
by (*simp add*: *bind-def*)

lemma *right-unit* [*simp*]: $bind{\cdot}m{\cdot}return = m$
by (*rule-tac p=m* **in** *maybeE*, *simp-all*)

lemma *bind-assoc*:
$bind{\cdot}(bind{\cdot}m{\cdot}k){\cdot}h = bind{\cdot}m{\cdot}(\Lambda\ a.\ bind{\cdot}(k{\cdot}a){\cdot}h)$
by (*rule-tac p=m* **in** *maybeE*, *simp-all*)

17.1.2 Run operator

definition
run:: *'a maybe* → *'a*::*pcpo* **where**
run = *maybe-when*·⊥·*ID*

rewrite rules for run

lemma *run-strict* [*simp*]: *run*·⊥ = ⊥
by (*simp add*: *run-def*)

lemma *run-fail* [*simp*]: *run*·*fail* = ⊥
by (*simp add*: *run-def*)

lemma *run-return* [*simp*]: *run*·(*return*·*x*) = *x*
by (*simp add*: *run-def*)

17.1.3 Monad plus operator

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

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

rewrite rules for mplus

lemma *mplus-strict* [*simp*]: ⊥ +++ *m* = ⊥
by (*simp add*: *mplus-def*)

lemma *mplus-fail* [*simp*]: *fail* +++ *m* = *m*
by (*simp add*: *mplus-def*)

lemma *mplus-return* [*simp*]: *return*·*x* +++ *m* = *return*·*x*
by (*simp add*: *mplus-def*)

lemma *mplus-fail2* [*simp*]: *m* +++ *fail* = *m*
by (*rule-tac p*=*m* **in** *maybeE*, *simp-all*)

lemma *mplus-assoc*: (*x* +++ *y*) +++ *z* = *x* +++ (*y* +++ *z*)
by (*rule-tac p*=*x* **in** *maybeE*, *simp-all*)

17.1.4 Fatbar combinator

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

abbreviation

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

lemma *fatbar1*: *m*·*x* = $\bot$ $\Longrightarrow$ (*m* $\parallel$ *ms*)·*x* = $\bot$
by (*simp add*: *fatbar-def*)

lemma *fatbar2*: *m*·*x* = *fail* $\Longrightarrow$ (*m* $\parallel$ *ms*)·*x* = *ms*·*x*
by (*simp add*: *fatbar-def*)

lemma *fatbar3*: *m*·*x* = *return*·*y* $\Longrightarrow$ (*m* $\parallel$ *ms*)·*x* = *return*·*y*
by (*simp add*: *fatbar-def*)

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

lemma *run-fatbar1*: *m*·*x* = $\bot$ $\Longrightarrow$ *run*·((*m* $\parallel$ *ms*)·*x*) = $\bot$
by (*simp add*: *fatbar-def*)

lemma *run-fatbar2*: *m*·*x* = *fail* $\Longrightarrow$ *run*·((*m* $\parallel$ *ms*)·*x*) = *run*·(*ms*·*x*)
by (*simp add*: *fatbar-def*)

lemma *run-fatbar3*: *m*·*x* = *return*·*y* $\Longrightarrow$ *run*·((*m* $\parallel$ *ms*)·*x*) = *y*
by (*simp add*: *fatbar-def*)

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 $\equiv$ Λ *r x*. *bind*·(*p*·*x*)·(Λ *y*. *return*·(*r*·*y*))

lemma *branch-rews*:
p·*x* = $\bot$ $\Longrightarrow$ *branch p*·*r*·*x* = $\bot$
p·*x* = *fail* $\Longrightarrow$ *branch p*·*r*·*x* = *fail*
p·*x* = *return*·*y* $\Longrightarrow$ *branch p*·*r*·*x* = *return*·(*r*·*y*)
by (*simp-all add*: *branch-def*)

lemma *branch-return* [*simp*]: *branch return*·*r*·*x* = *return*·(*r*·*x*)
by (*simp add*: *branch-def*)

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

parse-translation ⟪
(* rewrites (-pat x) => (return) *)
(* rewrites (-var x t) => (Abs-CFun (%x. t)) *)
  [(-pat, K (Syntax.const Fixrec.return)),
   mk-binder-tr (-var, Abs-CFun)];
⟫
```

Printing Case expressions

```
syntax
  -match :: 'a

print-translation ⟪
  let
    fun dest-LAM (Const (@{const-syntax Rep-CFun},-) $ Const (@{const-syntax
unit-when},-) $ t) =
          (Syntax.const @{const-syntax Unity}, t)
    |   dest-LAM (Const (@{const-syntax Rep-CFun},-) $ Const (@{const-syntax
csplit},-) $ t) =
          let
            val (v1, t1) = dest-LAM t;
            val (v2, t2) = dest-LAM t1;
          in (Syntax.const -args $ v1 $ v2, t2) end
    |   dest-LAM (Const (@{const-syntax Abs-CFun},-) $ t) =
          let
            val abs = case t of Abs abs => abs
                | - => (x, dummyT, incr-boundvars 1 t $ Bound 0);
            val (x, t') = atomic-abs-tr' abs;
          in (Syntax.const -var $ x, t') end
    |   dest-LAM - = raise Match; (* too few vars: abort translation *)
```

```
    fun Case1-tr' [Const(@{const-syntax branch},-) $ p, r] =
          let val (v, t) = dest-LAM r;
          in Syntax.const -Case1 $ (Syntax.const -match $ p $ v) $ t end;

  in [(@{const-syntax Rep-CFun}, Case1-tr')] end;
⟫
```

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

17.4 Pattern combinators for data constructors

types *('a, 'b) pat = 'a → 'b maybe*

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

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

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

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

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

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

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

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

Parse translations (patterns)

translations
-pat (XCONST cpair·x·y) => 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⟩ = ⊥
apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p=p·x* **in** *maybeE*, *simp-all*)
done

lemma *cpair-pat2*:
branch p·r·x = fail ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = fail
apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p=p·x* **in** *maybeE*, *simp-all*)
done

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

apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p*=$p{\cdot}x$ **in** *maybeE*, *simp-all*)
apply (*rule-tac p*=$q{\cdot}y$ **in** *maybeE*, *simp-all*)
done

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

lemma *spair-pat* [*simp*]:
branch (*spair-pat p1 p2*)$\cdot r\cdot\bot = \bot$
$[\![x \neq \bot;\ y \neq \bot]\!]$
$\Longrightarrow$ *branch* (*spair-pat p1 p2*)$\cdot r\cdot(:x,\ y:) =$
branch (*cpair-pat p1 p2*)$\cdot r\cdot\langle x,\ y\rangle$
by (*simp-all add*: *branch-def spair-pat-def*)

lemma *sinl-pat* [*simp*]:
branch (*sinl-pat p*)$\cdot r\cdot\bot = \bot$
$x \neq \bot \Longrightarrow$ *branch* (*sinl-pat p*)$\cdot r\cdot($*sinl*$\cdot x) =$ *branch* $p{\cdot}r{\cdot}x$
$y \neq \bot \Longrightarrow$ *branch* (*sinl-pat p*)$\cdot r\cdot($*sinr*$\cdot y) =$ *fail*
by (*simp-all add*: *branch-def sinl-pat-def*)

lemma *sinr-pat* [*simp*]:
branch (*sinr-pat p*)$\cdot r\cdot\bot = \bot$
$x \neq \bot \Longrightarrow$ *branch* (*sinr-pat p*)$\cdot r\cdot($*sinl*$\cdot x) =$ *fail*
$y \neq \bot \Longrightarrow$ *branch* (*sinr-pat p*)$\cdot r\cdot($*sinr*$\cdot y) =$ *branch* $p{\cdot}r{\cdot}y$
by (*simp-all add*: *branch-def sinr-pat-def*)

lemma *up-pat* [*simp*]:
branch (*up-pat p*)$\cdot r\cdot\bot = \bot$
branch (*up-pat p*)$\cdot r\cdot($*up*$\cdot x) =$ *branch* $p{\cdot}r{\cdot}x$
by (*simp-all add*: *branch-def up-pat-def*)

lemma *TT-pat* [*simp*]:
branch TT-pat$\cdot($*unit-when*$\cdot r)\cdot\bot = \bot$
branch TT-pat$\cdot($*unit-when*$\cdot r)\cdot TT =$ *return*$\cdot r$
branch TT-pat$\cdot($*unit-when*$\cdot r)\cdot FF =$ *fail*
by (*simp-all add*: *branch-def TT-pat-def*)

lemma *FF-pat* [*simp*]:
branch FF-pat$\cdot($*unit-when*$\cdot r)\cdot\bot = \bot$
branch FF-pat$\cdot($*unit-when*$\cdot r)\cdot TT =$ *fail*
branch FF-pat$\cdot($*unit-when*$\cdot r)\cdot FF =$ *return*$\cdot r$
by (*simp-all add*: *branch-def FF-pat-def*)

lemma *ONE-pat* [*simp*]:
branch ONE-pat$\cdot($*unit-when*$\cdot r)\cdot\bot = \bot$
branch ONE-pat$\cdot($*unit-when*$\cdot r)\cdot ONE =$ *return*$\cdot r$
by (*simp-all add*: *branch-def ONE-pat-def*)

17.5 Wildcards, as-patterns, and lazy patterns

syntax
-as-pat :: [*idt*, *'a*] ⇒ *'a* (**infixr** *as 10*)
-lazy-pat :: *'a* ⇒ *'a* (∼ - [*1000*] *1000*)

definition
wild-pat :: *'a* → *unit maybe* **where**
wild-pat = (Λ *x*. *return*·())

definition
as-pat :: (*'a* → *'b maybe*) ⇒ *'a* → (*'a* × *'b*) *maybe* **where**
as-pat p = (Λ *x*. *bind*·(*p*·*x*)·(Λ *a*. *return*·⟨*x*, *a*⟩))

definition
lazy-pat :: (*'a* → *'b*::*pcpo maybe*) ⇒ (*'a* → *'b maybe*) **where**
lazy-pat p = (Λ *x*. *return*·(*run*·(*p*·*x*)))

Parse translations (patterns)

translations
-pat - => *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-pat*·(*unit-when*·*r*)·*x* = *return*·*r*
by (*simp add*: *branch-def wild-pat-def*)

lemma *as-pat* [*simp*]:
branch (*as-pat p*)·(*csplit*·*r*)·*x* = *branch p*·(*r*·*x*)·*x*
apply (*simp add*: *branch-def as-pat-def*)
apply (*rule-tac p*=*p*·*x* **in** *maybeE*, *simp-all*)
done

lemma *lazy-pat* [*simp*]:
branch p·r·x = ⊥ ⟹ branch (lazy-pat p)·r·x = return·(r·⊥)
branch p·r·x = fail ⟹ branch (lazy-pat p)·r·x = return·(r·⊥)
branch p·r·x = return·s ⟹ branch (lazy-pat p)·r·x = return·s
apply (*simp-all add*: *branch-def lazy-pat-def*)
apply (*rule-tac* [!] *p=p·x* **in** *maybeE*, *simp-all*)
done

17.6 Match functions for built-in types

defaultsort *pcpo*

definition
match-UU :: *'a → unit maybe* **where**
match-UU = (Λ *x*. *fail*)

definition
match-cpair :: *'a*::*cpo* × *'b*::*cpo* → (*'a* × *'b*) *maybe* **where**
match-cpair = *csplit*·(Λ *x y*. *return*·<*x*,*y*>)

definition
match-spair :: *'a* ⊗ *'b* → (*'a* × *'b*) *maybe* **where**
match-spair = *ssplit*·(Λ *x y*. *return*·<*x*,*y*>)

definition
match-sinl :: *'a* ⊕ *'b* → *'a maybe* **where**
match-sinl = *sscase*·*return*·(Λ *y*. *fail*)

definition
match-sinr :: *'a* ⊕ *'b* → *'b maybe* **where**
match-sinr = *sscase*·(Λ *x*. *fail*)·*return*

definition
match-up :: *'a*::*cpo u* → *'a maybe* **where**
match-up = *fup*·*return*

definition
match-ONE :: *one* → *unit maybe* **where**
match-ONE = (Λ *ONE*. *return*·())

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

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

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

match-UU·x = fail
by (*simp add*: *match-UU-def*)

lemma *match-cpair-simps* [*simp*]:
match-cpair·<x,y> = return·<x,y>
by (*simp add*: *match-cpair-def*)

lemma *match-spair-simps* [*simp*]:
⟦x ≠ ⊥; y ≠ ⊥⟧ ⟹ match-spair·(:x,y:) = return·<x,y>
match-spair·⊥ = ⊥
by (*simp-all add*: *match-spair-def*)

lemma *match-sinl-simps* [*simp*]:
x ≠ ⊥ ⟹ match-sinl·(sinl·x) = return·x
x ≠ ⊥ ⟹ match-sinl·(sinr·x) = fail
match-sinl·⊥ = ⊥
by (*simp-all add*: *match-sinl-def*)

lemma *match-sinr-simps* [*simp*]:
x ≠ ⊥ ⟹ match-sinr·(sinr·x) = return·x
x ≠ ⊥ ⟹ match-sinr·(sinl·x) = fail
match-sinr·⊥ = ⊥
by (*simp-all add*: *match-sinr-def*)

lemma *match-up-simps* [*simp*]:
match-up·(up·x) = return·x
match-up·⊥ = ⊥
by (*simp-all add*: *match-up-def*)

lemma *match-ONE-simps* [*simp*]:
match-ONE·ONE = return·()
match-ONE·⊥ = ⊥
by (*simp-all add*: *match-ONE-def*)

lemma *match-TT-simps* [*simp*]:
match-TT·TT = return·()
match-TT·FF = fail
match-TT·⊥ = ⊥
by (*simp-all add*: *match-TT-def*)

lemma *match-FF-simps* [*simp*]:
match-FF·FF = return·()
match-FF·TT = fail
match-FF·⊥ = ⊥
by (*simp-all add*: *match-FF-def*)

17.7 Mutual recursion

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

lemma *cpair-equalI*: $[\![x \equiv cfst{\cdot}p;\ y \equiv csnd{\cdot}p]\!] \Longrightarrow <x,y> \equiv p$
by (*simp add*: *surjective-pairing-Cprod2*)

lemma *cpair-eqD1*: $<x,y> = <x',y'> \Longrightarrow x = x'$
by *simp*

lemma *cpair-eqD2*: $<x,y> = <x',y'> \Longrightarrow y = y'$
by *simp*

lemma for proving rewrite rules

lemma *ssubst-lhs*: $[\![t = s;\ P\ s = Q]\!] \Longrightarrow P\ t = Q$
by *simp*

17.8 Initializing the fixrec package

use *Tools/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*
  by (*rule iso.intro* [*OF rep-iso abs-iso*])

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

proof
  assume $abs{\cdot}x \sqsubseteq abs{\cdot}y$
  then have $rep{\cdot}(abs{\cdot}x) \sqsubseteq rep{\cdot}(abs{\cdot}y)$ **by** (*rule monofun-cfun-arg*)
  then show $x \sqsubseteq y$ **by** *simp*
next
  assume $x \sqsubseteq y$
  then show $abs{\cdot}x \sqsubseteq abs{\cdot}y$ **by** (*rule monofun-cfun-arg*)
qed

lemma *rep-less*: $(rep{\cdot}x \sqsubseteq rep{\cdot}y) = (x \sqsubseteq y)$
  by (*rule iso.abs-less* [*OF swap*])

lemma *abs-eq*: $(abs{\cdot}x = abs{\cdot}y) = (x = y)$
  by (*simp add*: *po-eq-conv abs-less*)

lemma *rep-eq*: $(rep{\cdot}x = rep{\cdot}y) = (x = y)$
  by (*rule iso.abs-eq* [*OF swap*])

lemma *abs-strict*: $abs{\cdot}\bot = \bot$
proof −
  have $\bot \sqsubseteq rep{\cdot}\bot$ **..**
  then have $abs{\cdot}\bot \sqsubseteq abs{\cdot}(rep{\cdot}\bot)$ **by** (*rule monofun-cfun-arg*)
  then have $abs{\cdot}\bot \sqsubseteq \bot$ **by** *simp*
  then show *?thesis* **by** (*rule UU-I*)
qed

lemma *rep-strict*: $rep{\cdot}\bot = \bot$
  by (*rule iso.abs-strict* [*OF swap*])

lemma *abs-defin'*: $abs{\cdot}x = \bot \Longrightarrow x = \bot$
proof −
  have $x = rep{\cdot}(abs{\cdot}x)$ **by** *simp*
  also assume $abs{\cdot}x = \bot$
  also note *rep-strict*
  finally show $x = \bot$ **.**
qed

lemma *rep-defin'*: $rep{\cdot}z = \bot \Longrightarrow z = \bot$
  by (*rule iso.abs-defin'* [*OF swap*])

lemma *abs-defined*: $z \neq \bot \Longrightarrow abs{\cdot}z \neq \bot$
  by (*erule contrapos-nn*, *erule abs-defin'*)

lemma *rep-defined*: $z \neq \bot \Longrightarrow rep{\cdot}z \neq \bot$
  by (*rule iso.abs-defined* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *abs-defined-iff*: $(abs{\cdot}x = \bot) = (x = \bot)$
  by (*auto elim*: *abs-defin' intro*: *abs-strict*)

lemma *rep-defined-iff*: $(rep{\cdot}x = \bot) = (x = \bot)$
by (*rule iso.abs-defined-iff* [*OF iso.swap*]) (*rule iso-axioms*)

lemma (**in** *iso*) *compact-abs-rev*: *compact* $(abs{\cdot}x) \Longrightarrow$ *compact* x
proof (*unfold compact-def*)
assume *adm* $(\lambda y.\ \neg\ abs{\cdot}x \sqsubseteq y)$
with *cont-Rep-CFun2*
have *adm* $(\lambda y.\ \neg\ abs{\cdot}x \sqsubseteq abs{\cdot}y)$ **by** (*rule adm-subst*)
then show *adm* $(\lambda y.\ \neg\ x \sqsubseteq y)$ **using** *abs-less* **by** *simp*
qed

lemma *compact-rep-rev*: *compact* $(rep{\cdot}x) \Longrightarrow$ *compact* x
by (*rule iso.compact-abs-rev* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *compact-abs*: *compact* $x \Longrightarrow$ *compact* $(abs{\cdot}x)$
by (*rule compact-rep-rev*) *simp*

lemma *compact-rep*: *compact* $x \Longrightarrow$ *compact* $(rep{\cdot}x)$
by (*rule iso.compact-abs* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *iso-swap*: $(x = abs{\cdot}y) = (rep{\cdot}x = y)$
proof
assume $x = abs{\cdot}y$
then have $rep{\cdot}x = rep{\cdot}(abs{\cdot}y)$ **by** *simp*
then show $rep{\cdot}x = y$ **by** *simp*
next
assume $rep{\cdot}x = y$
then have $abs{\cdot}(rep{\cdot}x) = abs{\cdot}y$ **by** *simp*
then show $x = abs{\cdot}y$ **by** *simp*
qed

end

18.2 Casedist

lemma *ex-one-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) = P\ ONE$
apply *safe*
apply (*rule-tac p=x* **in** *oneE*)
apply *simp*
apply *simp*
apply *force*
done

lemma *ex-up-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) = (\exists\, x.\ P\ (up{\cdot}x))$
apply *safe*
apply (*rule-tac p=x* **in** *upE*)
apply *simp*

apply *fast*
apply (*force intro*!: *up-defined*)
done

lemma *ex-sprod-defined-iff*:
$(\exists\, y.\ P\ y \wedge y \neq \bot) =$
$(\exists\, x\ y.\ (P\ (:x,\ y:) \wedge x \neq \bot) \wedge y \neq \bot)$
apply *safe*
apply (*rule-tac p=y* **in** *sprodE*)
apply *simp*
apply *fast*
apply (*force intro*!: *spair-defined*)
done

lemma *ex-sprod-up-defined-iff*:
$(\exists\, y.\ P\ y \wedge y \neq \bot) =$
$(\exists\, x\ y.\ P\ (:up{\cdot}x,\ y:) \wedge y \neq \bot)$
apply *safe*
apply (*rule-tac p=y* **in** *sprodE*)
apply *simp*
apply (*rule-tac p=x* **in** *upE*)
apply *simp*
apply *fast*
apply (*force intro*!: *spair-defined*)
done

lemma *ex-ssum-defined-iff*:
$(\exists\, x.\ P\ x \wedge x \neq \bot) =$
$((\exists\, x.\ P\ (sinl{\cdot}x) \wedge x \neq \bot) \vee$
$(\exists\, x.\ P\ (sinr{\cdot}x) \wedge x \neq \bot))$
apply (*rule iffI*)
apply (*erule exE*)
apply (*erule conjE*)
apply (*rule-tac p=x* **in** *ssumE*)
apply *simp*
apply (*rule disjI1*, *fast*)
apply (*rule disjI2*, *fast*)
apply (*erule disjE*)
apply *force*
apply *force*
done

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

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*: $[\![R;\ R \Longrightarrow P]\!] \Longrightarrow P$
by *auto*

lemma *exh-casedist1*: $((P \vee Q \Longrightarrow R) \Longrightarrow S) \equiv ([\![P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow S)$
by *rule auto*

lemma *exh-casedist2*: $(\exists x.\ P\ x \Longrightarrow Q) \equiv (\bigwedge x.\ P\ x \Longrightarrow Q)$
by *rule auto*

lemma *exh-casedist3*: $(P \wedge Q \Longrightarrow R) \equiv (P \Longrightarrow Q \Longrightarrow R)$
by *rule auto*

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

18.3 Setting up the package

ML ⟪
val iso-intro = *thm iso.intro*;
val iso-abs-iso = *thm iso.abs-iso*;
val iso-rep-iso = *thm iso.rep-iso*;
val iso-abs-strict = *thm iso.abs-strict*;
val iso-rep-strict = *thm iso.rep-strict*;
val iso-abs-defin' = *thm iso.abs-defin'*;
val iso-rep-defin' = *thm iso.rep-defin'*;
val iso-abs-defined = *thm iso.abs-defined*;
val iso-rep-defined = *thm iso.rep-defined*;
val iso-compact-abs = *thm iso.compact-abs*;
val iso-compact-rep = *thm iso.compact-rep*;
val iso-iso-swap = *thm iso.iso-swap*;

val exh-start = *thm exh-start*;
val ex-defined-iffs = *thms ex-defined-iffs*;
val exh-casedist0 = *thm exh-casedist0*;
val exh-casedists = *thms exh-casedists*;
⟫

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-setup ⟪
change-simpset (*fn simpset => simpset addSolver*
(*mk-solver' adm-tac* (*fn ss =>*
adm-tac (*cut-facts-tac* (*Simplifier.prems-of-ss ss*) *THEN' cont-tacRs ss*))));
⟫

end