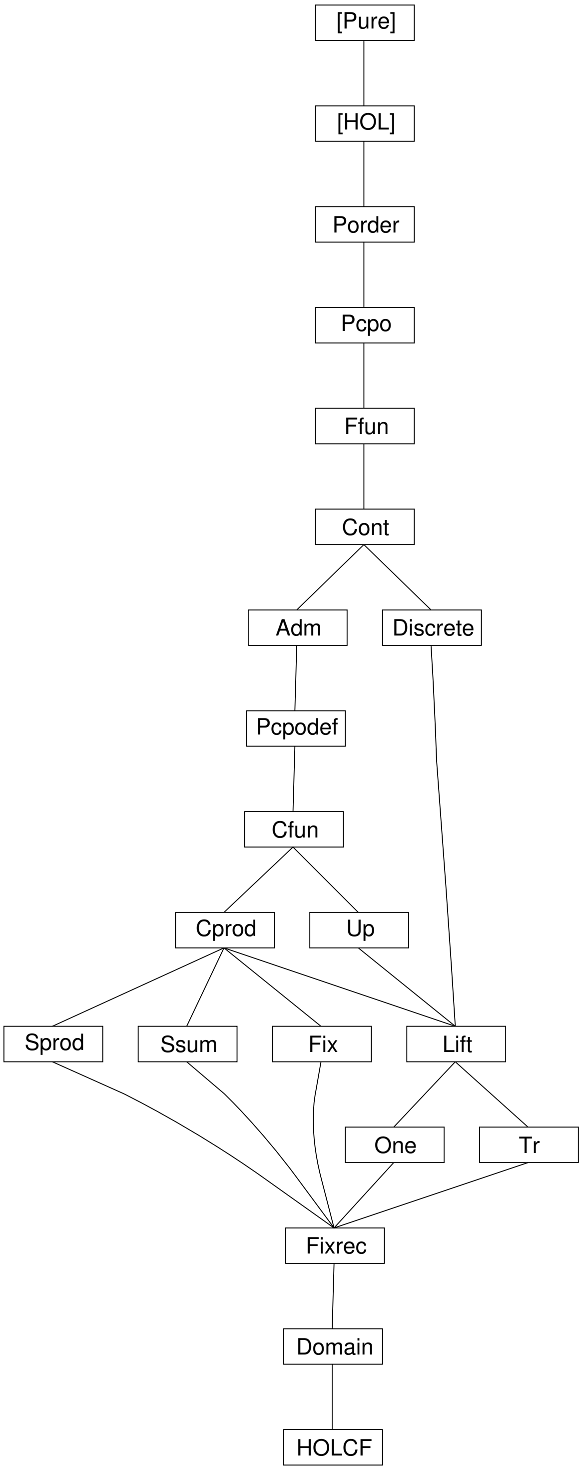
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

October 1, 2005

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 *Main*
begin

1.1 Type class for partial orders

— introduce a (syntactic) class for the constant <<
axclass *sq-ord* < *type*

— characteristic constant << for po
consts
<< :: [*'a*,*'a*::*sq-ord*] => *bool* (**infixl** *55*)

syntax (*xsymbols*)
op << :: [*'a*,*'a*::*sq-ord*] => *bool* (**infixl** $\sqsubseteq$ *55*)

axclass *po* < *sq-ord*
— class axioms:
refl-less [*iff*]: *x* << *x*
antisym-less: [|*x* << *y*; *y* << *x* |] ==> *x* = *y*
trans-less: [|*x* << *y*; *y* << *z* |] ==> *x* << *z*

minimal fixes least element

lemma *minimal2UU*[*OF allI*] : !*x*::*'a*::*po*. *uu*<<*x* ==> *uu*=(*THE u*.!*y*. *u*<<*y*)
by (*blast intro*: *theI2 antisym-less*)

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

lemma *antisym-less-inverse*: (*x*::*'a*::*po*)=*y* ==> *x* << *y* & *y* << *x*
apply *blast*
done

lemma *box-less*: [| (*a*::*'a*::*po*) << *b*; *c* << *a*; *b* << *d*|] ==> *c* << *d*
apply (*erule trans-less*)
apply (*erule trans-less*)
apply *assumption*
done

lemma *po-eq-conv*: ((*x*::*'a*::*po*)=*y*) = (*x* << *y* & *y* << *x*)
apply (*fast elim*!: *antisym-less-inverse intro*!: *antisym-less*)
done

1.2 Chains and least upper bounds

consts
<| :: [*'a set*,*'a*::*po*] => *bool* (**infixl** *55*)
<<| :: [*'a set*,*'a*::*po*] => *bool* (**infixl** *55*)
lub :: *'a set* => *'a*::*po*

tord :: 'a::po set => bool
chain :: (nat=>'a::po) => bool
max-in-chain :: [nat,nat=>'a::po]=>bool
finite-chain :: (nat=>'a::po)=>bool

syntax
@LUB :: ('b => 'a) => 'a (**binder** LUB 10)

translations
LUB x. t == lub(range(%x. t))

syntax (*xsymbols*)
LUB :: [idts, 'a] => 'a ((3⨆-./ -)[0,10] 10)

defs

— class definitions
is-ub-def: S <| x == ! y. y:S --> y<<x
is-lub-def: S <<| x == S <| x & (!u. S <| u --> x << u)

— Arbitrary chains are total orders
tord-def: tord S == !x y. x:S & y:S --> (x<<y | y<<x)

— Here we use countable chains and I prefer to code them as functions!
chain-def: chain F == !i. F i << F (Suc i)

— finite chains, needed for monotony of continouous functions
max-in-chain-def: max-in-chain i C == ! j. i <= j --> C(i) = C(j)
finite-chain-def: finite-chain C == chain(C) & (? i. max-in-chain i C)

lub-def: lub S == (THE x. S <<| x)

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 ==> x<y --> F x<<F y
apply (*unfold chain-def*)
apply (*induct-tac y*)
apply *auto*
prefer *2* **apply** (*blast intro*: *trans-less*)
apply (*blast elim*!: *less-SucE*)
done

lemma *chain-mono3*: [| chain F; x <= y |] ==> F x << 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*) ==> *tord*(*range*(*F*))
apply (*unfold tord-def*)
apply *safe*
apply (*rule nat-less-cases*)
apply (*fast intro*: *chain-mono*)+
done

technical lemmas about *lub* and *is-lub*

lemmas *lub* = *lub-def* [*THEN meta-eq-to-obj-eq*, *standard*]

lemma *lubI*[*OF exI*]: *EX x*. *M* <<| *x* ==> *M* <<| *lub*(*M*)
apply (*unfold lub-def*)
apply (*rule theI'*)
apply (*erule ex-ex1I*)
apply (*erule unique-lub*)
apply *assumption*
done

lemma *thelubI*: *M* <<| *l* ==> *lub*(*M*) = *l*
apply (*rule unique-lub*)
apply (*rule lubI*)
apply *assumption*
apply *assumption*
done

lemma *lub-singleton* [*simp*]: *lub*{*x*} = *x*
apply (*simp* (*no-asm*) *add*: *thelubI is-lub-def is-ub-def*)
done

access to some definition as inference rule

lemma *is-lubD1*: *S* <<| *x* ==> *S* <| *x*
apply (*unfold is-lub-def*)
apply *auto*
done

lemma *is-lub-lub*: [| *S* <<| *x*; *S* <| *u* |] ==> *x* << *u*
apply (*unfold is-lub-def*)
apply *auto*
done

lemma *is-lubI*:
    [| *S* <| *x*; !!*u*. *S* <| *u* ==> *x* << *u* |] ==> *S* <<| *x*
apply (*unfold is-lub-def*)
apply *blast*

done

lemma *chainE*: *chain F ==> F(i) << F(Suc(i))*
apply (*unfold chain-def*)
apply *auto*
done

lemma *chainI*: (!!*i. F i << F(Suc i)*) *==> chain F*
apply (*unfold chain-def*)
apply *blast*
done

lemma *chain-shift*: *chain Y ==> chain (%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 ==> S(i) << x*
apply (*unfold is-ub-def*)
apply *blast*
done

lemma *ub-rangeI*: (!!*i. S i << x*) *==> range S <| x*
apply (*unfold is-ub-def*)
apply *blast*
done

lemmas *is-ub-lub* = *is-lubD1* [*THEN ub-rangeD*, *standard*]
— *range ?S <<| ?x* $\Longrightarrow$ *?S ?i* $\sqsubseteq$ *?x*

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*|] ==> *range C* <<| *C i*
apply (*unfold max-in-chain-def*)
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*rule-tac m = i* **in** *nat-less-cases*)
apply (*rule antisym-less-inverse* [*THEN conjunct2*])
apply (*erule disjI1* [*THEN less-or-eq-imp-le*, *THEN rev-mp*])
apply (*erule spec*)
apply (*rule antisym-less-inverse* [*THEN conjunct2*])
apply (*erule disjI2* [*THEN less-or-eq-imp-le*, *THEN rev-mp*])
apply (*erule spec*)
apply (*erule chain-mono*)
apply *assumption*
apply (*erule ub-rangeD*)
done

lemma *lub-finch2*:
  finite-chain(*C*) ==> *range*(*C*) <<| *C*(*LEAST i*. *max-in-chain i C*)
apply (*unfold finite-chain-def*)
apply (*rule lub-finch1*)
prefer *2* **apply** (*best intro*: *LeastI*)
apply *blast*
done

lemma *bin-chain*: *x*<<*y* ==> *chain* (%*i*. *if i*=*0 then x else y*)
apply (*rule chainI*)
apply (*induct-tac i*)
apply *auto*
done

lemma *bin-chainmax*:
  x<<*y* ==> *max-in-chain* (*Suc 0*) (%*i*. *if* (*i*=*0*) *then x else y*)
apply (*unfold max-in-chain-def le-def*)
apply (*rule allI*)
apply (*induct-tac j*)
apply *auto*
done

lemma *lub-bin-chain*: *x* << *y* ==> *range*(%*i*::*nat*. *if* (*i*=*0*) *then x else y*) <<| *y*
apply (*rule-tac s* = *if* (*Suc 0*) = *0 then x else y* **in** *subst* , *rule-tac* [*2*] *lub-finch1*)
apply (*erule-tac* [*2*] *bin-chain*)
apply (*erule-tac* [*2*] *bin-chainmax*)
apply (*simp* (*no-asm*))
done

the maximal element in a chain is its lub

lemma *lub-chain-maxelem*: [| *Y i* = *c*; *ALL i*. *Y i*<<*c* |] ==> *lub*(*range Y*) = *c*
apply (*blast dest*: *ub-rangeD intro*: *thelubI is-lubI ub-rangeI*)

done

the lub of a constant chain is the constant

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

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

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

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* $(S::nat \Rightarrow 'a::cpo)$; *range* S <| $x]\!] \Longrightarrow (\bigsqcup i.\ S\ i) \sqsubseteq x$
by (*blast dest*: *cpo intro*: *lubI* [*THEN is-lub-lub*])

lemma *lub-range-mono*:
$[\![$*range* $X \subseteq$ *range* Y; *chain* Y; *chain* $(X::nat \Rightarrow 'a::cpo)]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
apply (*erule is-lub-thelub*)
apply (*rule ub-rangeI*)
apply (*subgoal-tac* $\exists j.\ X\ i = Y\ j$)
apply *clarsimp*

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

lemma *lub-range-shift*:
$chain\ (Y{::}nat \Rightarrow {'a}{::}cpo) \Longrightarrow (\bigsqcup i.\ Y\ (i + j)) = (\bigsqcup i.\ Y\ i)$
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 trans-less*)
apply (*rule-tac* [*2*] *is-ub-thelub*)
apply (*erule-tac* [*2*] *chain-shift*)
apply (*erule chain-mono3*)
apply (*rule le-add1*)
done

lemma *maxinch-is-thelub*:
$chain\ Y \Longrightarrow max\text{-}in\text{-}chain\ i\ Y = ((\bigsqcup i.\ Y\ i) = ((Y\ i){::}{'a}{::}cpo))$
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 $\sqsubseteq$ relation between two chains is preserved by their lubs

lemma *lub-mono*:
$[\![chain\ (X{::}nat \Rightarrow {'a}{::}cpo);\ chain\ Y;\ \forall k.\ X\ k \sqsubseteq Y\ k]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
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*:
$[\![chain\ (X{::}nat \Rightarrow {'a}{::}cpo);\ chain\ Y;\ \forall k.\ X\ k = Y\ k]\!]$
$\Longrightarrow (\bigsqcup i.\ X\ i) = (\bigsqcup i.\ Y\ i)$
by (*simp only*: *expand-fun-eq* [*symmetric*])

more results about mono and = of lubs of chains

lemma *lub-mono2*:
  $[\![\exists j{::}nat.\ \forall i{>}j.\ X\ i = Y\ i;\ chain\ (X{::}nat{=}{>}'a{::}cpo);\ chain\ Y]\!]$
    $\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
apply (*erule exE*)
apply (*rule is-lub-thelub*)
apply *assumption*
apply (*rule ub-rangeI*)
apply (*case-tac j < i*)
apply (*rule-tac s=Y i* **and** *t=X i* **in** *subst*)
apply *simp*
apply (*erule is-ub-thelub*)
apply (*rule-tac y = X (Suc j)* **in** *trans-less*)
apply (*erule chain-mono*)
apply (*erule not-less-eq [THEN iffD1]*)
apply (*rule-tac s=Y (Suc j)* **and** *t=X (Suc j)* **in** *subst*)
apply *simp*
apply (*erule is-ub-thelub*)
done

lemma *lub-equal2*:
  $[\![\exists j.\ \forall i{>}j.\ X\ i = Y\ i;\ chain\ (X{::}nat \Rightarrow 'a{::}cpo);\ chain\ Y]\!]$
    $\Longrightarrow (\bigsqcup i.\ X\ i) = (\bigsqcup i.\ Y\ i)$
by (*blast intro*: *antisym-less lub-mono2 sym*)

lemma *lub-mono3*:
  $[\![chain\ (Y{::}nat \Rightarrow 'a{::}cpo);\ chain\ X;\ \forall i.\ \exists j.\ Y\ i \sqsubseteq X\ j]\!]$
    $\Longrightarrow (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ X\ i)$
apply (*rule is-lub-thelub*)
apply *assumption*
apply (*rule ub-rangeI*)
apply (*erule allE*)
apply (*erule exE*)
apply (*erule trans-less*)
apply (*erule is-ub-thelub*)
done

lemma *ch2ch-lub*:
  fixes $Y :: nat \Rightarrow nat \Rightarrow 'a{::}cpo$
  assumes *1*: $\bigwedge j.\ chain\ (\lambda i.\ Y\ i\ j)$
  assumes *2*: $\bigwedge i.\ chain\ (\lambda j.\ Y\ i\ j)$
  shows $chain\ (\lambda i.\ \bigsqcup j.\ Y\ i\ j)$
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: $\exists\, x.\ \forall\, y.\ x \sqsubseteq y$

constdefs
$UU :: {'a}{::}pcpo$
$UU \equiv THE\ x.\ ALL\ y.\ x \sqsubseteq y$

syntax (*xsymbols*)
$UU :: {'a}{::}pcpo\ (\bot)$

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*]: $\perp \sqsubseteq x$
by (*rule UU-least* [*THEN spec*])

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

ML-setup ⟪
local
  val meta-UU-reorient = *thm UU-reorient RS eq-reflection*;
  fun is-UU (*Const* (*Pcpo.UU*,-)) = *true*
    | *is-UU* - = *false*;
  fun reorient-proc sg - (- \$ *t* \$ *u*) =
    if is-UU u then NONE else SOME meta-UU-reorient;
in
  val UU-reorient-simproc =
    Simplifier.simproc (*the-context* ())
      UU-reorient-simproc [*UU=x*] *reorient-proc*
end;

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

useful lemmas about $\perp$

lemma *eq-UU-iff*: $(x = \perp) = (x \sqsubseteq \perp)$
apply (*rule iffI*)
apply (*erule ssubst*)
apply (*rule refl-less*)
apply (*rule antisym-less*)
apply *assumption*
apply (*rule minimal*)
done

lemma *UU-I*: $x \sqsubseteq \perp \Longrightarrow x = \perp$
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*: $[\![chain\ Y;\ (\bigsqcup i.\ Y\ i) = \perp]\!] \Longrightarrow \forall i.\ Y\ i = \perp$
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 (*erule le-imp-less-or-eq* [*THEN disjE*])
apply *safe*
apply (*blast dest*: *chain-mono 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*: $(a::'a::flat) \neq \perp \Longrightarrow a \sqsubseteq b = (a = b)$
by (*safe dest*!: *ax-flat* [*rule-format*])

lemma *chfin2finch*: *chain* $(Y::nat \Rightarrow 'a::chfin) \Longrightarrow$ *finite-chain* Y
by (*simp add*: *chfin finite-chain-def*)

lemmata for improved admissibility introdution rule

lemma *infinite-chain-adm-lemma*:
$[\![chain\ Y;\ \forall i.\ P\ (Y\ i);$
$\bigwedge Y.\ [\![chain\ Y;\ \forall i.\ P\ (Y\ i);\ \neg\ finite\text{-}chain\ Y]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)]\!]$
$\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
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*:
$[\![chain\ Y;\ \ \forall i.\ P\ (Y\ i);\ \bigwedge Y.\ [\![chain\ Y;\ \forall i.\ P\ (Y\ i);$
$\forall i.\ \exists j{>}i.\ Y\ i \neq Y\ j \wedge Y\ i \sqsubseteq Y\ j]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)]\!]$
$\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
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 Type $'a \Rightarrow 'b$ 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 *less-fun*: $(f \sqsubseteq g) = (\forall x.\ f\ x \sqsubseteq g\ x)$
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 Type $'a \Rightarrow 'b$ is pointed

lemma *minimal-fun*: $(\lambda x.\ \bot) \sqsubseteq f$
by (*simp add*: *less-fun-def*)

lemma *least-fun*: $\exists x::'a \Rightarrow 'b::pcpo.\ \forall y.\ x \sqsubseteq y$
apply (*rule-tac* $x = \lambda x.\ \bot$ **in** *exI*)
apply (*rule minimal-fun* [*THEN allI*])
done

3.3 Type $'a \Rightarrow 'b$ 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-fun-rev*: $(\bigwedge x.\ chain\ (\lambda i.\ S\ i\ x)) \Longrightarrow chain\ S$
by (*simp add*: *chain-def less-fun-def*)

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

lemma *ub2ub-fun*:
$range\ (S{::}nat \Rightarrow {'a} \Rightarrow {'b}{::}po) <|\ u \Longrightarrow range\ (\lambda i.\ S\ i\ x) <|\ u\ x$
by (*auto simp add*: *is-ub-def less-fun-def*)

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

lemma *lub-fun*:
$chain\ (S{::}nat \Rightarrow {'a}{::}type \Rightarrow {'b}{::}cpo)$
$\Longrightarrow range\ S <<|\ (\lambda x.\ \bigsqcup i.\ S\ i\ x)$
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*:
$chain\ (S{::}nat \Rightarrow {'a}{::}type \Rightarrow {'b}{::}cpo)$
$\Longrightarrow lub\ (range\ S) = (\lambda x.\ \bigsqcup i.\ S\ i\ x)$
by (*rule lub-fun* [*THEN thelubI*])

lemma *cpo-fun*:
$chain\ (S{::}nat \Rightarrow {'a}{::}type \Rightarrow {'b}{::}cpo) \Longrightarrow \exists x.\ range\ S <<|\ x$
by (*rule exI*, *erule lub-fun*)

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

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

for compatibility with old HOLCF-Version

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

function application is strict in the left argument

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

constdefs
$monofun :: ('a \Rightarrow 'b) \Rightarrow bool$ — monotonicity
$monofun\ f \equiv \forall\, x\ y.\ x \sqsubseteq y \longrightarrow f\,x \sqsubseteq f\,y$

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

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

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

monotone functions map chains to chains

lemma *ch2ch-monofun*: $[\![monofun\ f;\ chain\ Y]\!] \Longrightarrow chain\ (\lambda i.\ f\ (Y\ i))$
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: $monofun\ f \wedge contlub\ f \Longrightarrow cont\ f$

lemma *monocontlub2cont*: $[\![monofun\ f;\ contlub\ f]\!] \Longrightarrow cont\ f$
apply (*rule contI*)
apply (*rule thelubE*)
apply (*erule ch2ch-monofun*)
apply *assumption*
apply (*erule contlubE* [*symmetric*])
apply *assumption*
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 binchain-cont*, *assumption*)
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 contE*)
apply *assumption*
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*:
$[\![chain\ (F::nat \Rightarrow {'a} \Rightarrow {'b}::cpo);\ \forall\, i.\ monofun\ (F\ i)]\!]$
$\Longrightarrow monofun\ (\bigsqcup i.\ F\ i)$
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*:
$[\![chain\ F;\ \forall\, i.\ cont\ (F\ i)]\!] \Longrightarrow contlub\ (\bigsqcup i.\ F\ i)$
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*:
$[\![chain\ F;\ \forall\, i.\ cont\ (F\ i)]\!] \Longrightarrow cont\ (\bigsqcup i.\ F\ i)$
apply (*rule monocontlub2cont*)
apply (*erule monofun-lub-fun*)
apply (*simp add*: *cont2mono*)
apply (*erule contlub-lub-fun*)
apply *assumption*
done

lemma *cont2cont-lub*:
$[\![chain\ F;\ \bigwedge i.\ cont\ (F\ i)]\!] \Longrightarrow cont\ (\lambda x.\ \bigsqcup i.\ F\ i\ x)$
by (*simp add*: *thelub-fun* [*symmetric*] *cont-lub-fun*)

lemma *mono2mono-MF1L*: $monofun\ f \Longrightarrow monofun\ (\lambda x.\ f\ x\ y)$
apply (*rule monofunI*)
apply (*erule* (*1*) *monofun-fun-arg* [*THEN monofun-fun-fun*])
done

lemma *cont2cont-CF1L*: $cont\ f \Longrightarrow cont\ (\lambda x.\ f\ x\ y)$
apply (*rule monocontlub2cont*)
apply (*erule cont2mono* [*THEN mono2mono-MF1L*])
apply (*rule contlubI*)
apply (*simp add*: *cont2contlubE*)
apply (*simp add*: *thelub-fun ch2ch-cont*)
done

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

lemma *mono2mono-MF1L-rev*: $\forall\, y.\ monofun\ (\lambda x.\ f\ x\ y) \Longrightarrow monofun\ f$
apply (*rule monofunI*)
apply (*rule less-fun* [*THEN iffD2*])
apply (*blast dest*: *monofunE*)
done

lemma *cont2cont-CF1L-rev*: $\forall\, y.\ cont\ (\lambda x.\ f\ x\ y) \Longrightarrow cont\ f$
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-MF1L-rev cont2mono*)
done

lemma *cont2cont-lambda*: $(\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)) \Longrightarrow cont\ (\lambda x.\ (\lambda y.\ f\ x\ y))$
apply (*rule cont2cont-CF1L-rev*)
apply *simp*
done

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

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 (*erule* (*1*) *cont2cont-CF1L-rev*)
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*; $\forall x$. *cont* (*f x*); *cont t*]] $\Longrightarrow$ *cont* (λx. (*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*]] $\Longrightarrow$ *cont* (λx. *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*]] $\Longrightarrow$ *finite-chain* (λn. *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*]] $\Longrightarrow$ *finite-chain* (λn. *f* (*Y n*))
by (*rule cont2mono* [*THEN monofun-finch2finch*])

lemma *chfindom-monofun2cont*: *monofun f* $\Longrightarrow$ *cont* (*f*::'*a*::*chfin* $\Rightarrow$ '*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* (λi. *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* $\perp = \perp \Longrightarrow$ *monofun* (*f*::'*a*::*flat* $\Rightarrow$ '*b*::*pcpo*)
apply (*rule monofunI*)
apply (*drule ax-flat* [*rule-format*])
apply *auto*
done

lemma *flatdom-strict2cont*: *f* $\perp = \perp \Longrightarrow$ *cont* (*f*::'*a*::*flat* $\Rightarrow$ '*b*::*pcpo*)
by (*rule flatdom-strict2mono* [*THEN chfindom-monofun2cont*])

end

5 Adm: Admissibility

theory *Adm*
imports *Cont*
begin

defaultsort *cpo*

5.1 Definitions

constdefs
$adm :: ('a::cpo \Rightarrow bool) \Rightarrow bool$
$adm\ P \equiv \forall\, Y.\ chain\ Y \longrightarrow (\forall\, i.\ P\ (Y\ i)) \longrightarrow P\ (\bigsqcup i.\ Y\ i)$

lemma *admI*:
$(\bigwedge Y.\ [\![chain\ Y;\ \forall\, i.\ P\ (Y\ i)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)) \Longrightarrow adm\ P$
apply (*unfold adm-def*)
apply *blast*
done

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

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

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

apply (*unfold adm-def*)
apply (*intro strip*)
apply (*drule spec*)
apply (*drule mp*)
apply *assumption*
apply (*erule exE*)
apply (*simp add*: *maxinch-is-thelub*)
done

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

5.3 Admissibility of special formulae and propagation

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

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

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 (*rule trans-less*)
prefer *2* **apply** (*assumption*)
apply (*erule cont2mono* [*THEN monofun-fun-arg*])
apply (*erule is-ub-thelub*)
done

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

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

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-ball2* = *adm-ball* [*rule-format*]

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-UU-not-less*: $adm\ (\lambda x.\ \neg \perp \sqsubseteq t\ x)$
by (*simp add*: *adm-not-free*)

lemma *adm-not-UU*: $cont\ t \Longrightarrow adm\ (\lambda x.\ \neg\ t\ x = \perp)$
by (*simp add*: *eq-UU-iff adm-not-less*)

lemma *adm-eq*: $[\![cont\ u;\ cont\ v]\!] \Longrightarrow adm\ (\lambda x.\ u\ x = v\ x)$
by (*simp add*: *po-eq-conv adm-conj adm-less*)

admissibility for disjunction is hard to prove. It takes 7 Lemmas

lemma *adm-disj-lemma1*:
$\forall n{::}nat.\ P\ n \vee Q\ n \Longrightarrow (\forall i.\ \exists j{\geq}i.\ P\ j) \vee (\forall i.\ \exists j{\geq}i.\ Q\ j)$
apply (*erule contrapos-pp*)
apply *clarsimp*
apply (*rule exI*)
apply (*rule conjI*)
apply (*drule spec*, *erule mp*)
apply (*rule le-maxI1*)
apply (*drule spec*, *erule mp*)
apply (*rule le-maxI2*)
done

lemma *adm-disj-lemma2*:
$[\![adm\ P;\ \exists X.\ chain\ X \wedge (\forall n.\ P\ (X\ n)) \wedge (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ X\ i)]\!]$
$\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
by (*force elim*: *admD*)

lemma *adm-disj-lemma3*:
$[\![chain\ (Y{::}nat \Rightarrow 'a{::}cpo);\ \forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!]$
$\Longrightarrow chain\ (\lambda m.\ Y\ (LEAST\ j.\ m \leq j \wedge P\ (Y\ j)))$
apply (*rule chainI*)
apply (*erule chain-mono3*)
apply (*rule Least-le*)
apply (*drule-tac x=Suc i* **in** *spec*)
apply (*rule conjI*)
apply (*rule Suc-leD*)
apply (*erule LeastI-ex* [*THEN conjunct1*])
apply (*erule LeastI-ex* [*THEN conjunct2*])
done

lemma *adm-disj-lemma4*:
$[\![\forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!] \Longrightarrow \forall m.\ P\ (Y\ (LEAST\ j{::}nat.\ m \leq j \wedge P\ (Y\ j)))$

apply (*rule allI*)
apply (*drule-tac x=m* **in** *spec*)
apply (*erule LeastI-ex* [*THEN conjunct2*])
done

lemma *adm-disj-lemma5*:
$[\![chain\ (Y::nat \Rightarrow 'a::cpo);\ \forall\, i.\ \exists\, j{\geq}i.\ P\ (Y\, j)]\!] \Longrightarrow$
$(\bigsqcup m.\ Y\ m) = (\bigsqcup m.\ Y\ (LEAST\, j.\ m \leq j \wedge P\ (Y\, j)))$
apply (*rule antisym-less*)
apply (*rule lub-mono*)
apply *assumption*
apply (*erule* (*1*) *adm-disj-lemma3*)
apply (*rule allI*)
apply (*erule chain-mono3*)
apply (*drule-tac x=k* **in** *spec*)
apply (*erule LeastI-ex* [*THEN conjunct1*])
apply (*rule lub-mono3*)
apply (*erule* (*1*) *adm-disj-lemma3*)
apply *assumption*
apply (*rule allI*)
apply (*rule exI*)
apply (*rule refl-less*)
done

lemma *adm-disj-lemma6*:
$[\![chain\ (Y::nat \Rightarrow 'a::cpo);\ \forall\, i.\ \exists\, j{\geq}i.\ P(Y\, j)]\!] \Longrightarrow$
$\exists\, X.\ chain\ X \wedge (\forall\, n.\ P\ (X\ n)) \wedge (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ X\ i)$
apply (*rule-tac* $x = \lambda m.\ Y\ (LEAST\, j.\ m \leq j \wedge P\ (Y\, j))$ **in** *exI*)
apply (*fast intro*!: *adm-disj-lemma3 adm-disj-lemma4 adm-disj-lemma5*)
done

lemma *adm-disj-lemma7*:
$[\![adm\ P;\ chain\ Y;\ \forall\, i.\ \exists\, j{\geq}i.\ P\ (Y\, j)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
apply (*erule adm-disj-lemma2*)
apply (*erule* (*1*) *adm-disj-lemma6*)
done

lemma *adm-disj*: $[\![adm\ P;\ adm\ Q]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \vee Q\ x)$
apply (*rule admI*)
apply (*erule adm-disj-lemma1* [*THEN disjE*])
apply (*rule disjI1*)
apply (*erule* (*2*) *adm-disj-lemma7*)
apply (*rule disjI2*)
apply (*erule* (*2*) *adm-disj-lemma7*)
done

lemma *adm-imp*: $[\![adm\ (\lambda x.\ \neg\ P\ x);\ adm\ Q]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \longrightarrow Q\ x)$
by (*subst imp-conv-disj*, *rule adm-disj*)

lemma *adm-iff*:
  $[\![adm\ (\lambda x.\ P\ x \longrightarrow Q\ x);\ adm\ (\lambda x.\ Q\ x \longrightarrow P\ x)]\!]$
    $\Longrightarrow adm\ (\lambda x.\ P\ x = Q\ x)$
by (*subst iff-conv-conj-imp*, *rule adm-conj*)

lemma *adm-not-conj*:
  $[\![adm\ (\lambda x.\ \neg\ P\ x);\ adm\ (\lambda x.\ \neg\ Q\ x)]\!] \Longrightarrow adm\ (\lambda x.\ \neg\ (P\ x \wedge Q\ x))$
by (*subst de-Morgan-conj*, *rule adm-disj*)

lemmas *adm-lemmas* =
  adm-less adm-conj adm-not-free adm-imp adm-disj adm-eq adm-not-UU
  adm-UU-not-less adm-all2 adm-not-less adm-not-conj adm-iff

declare *adm-lemmas* [*simp*]

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-UU-not-less = thm adm-UU-not-less;
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-lemma6 = thm adm-disj-lemma6;
val adm-disj-lemma7 = thm adm-disj-lemma7;
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;
```

$\rangle\!\rangle$

end

6 Pcpodef: Subtypes of pcpos

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

6.1 Proving a subtype is a partial order

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

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

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* $\Rightarrow$ $'b$::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq$ $\equiv$ $\lambda x\ y.$ *Rep x* $\sqsubseteq$ *Rep y*
and *adm*: *adm* ($\lambda x.\ x \in A$)
shows *OFCLASS*($'b$, *cpo-class*)
proof
fix *S*::*nat* $\Rightarrow$ $'b$ **assume** *chain S*
hence *range S* <<| *Abs* ($\bigsqcup i.$ *Rep* (*S i*))
by (*rule typedef-lub* [*OF type less adm*])
thus $\exists x.$ *range S* <<| *x* **..**
qed

6.2.1 Continuity of *Rep* and *Abs*

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

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

6.3 Proving a subtype is pointed

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

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

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

6.3.1 Strictness of *Rep* and *Abs*

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

theorem *typedef-Abs-strict*:
assumes *type*: *type-definition Rep Abs A*
and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\perp \in A$
shows $Abs\ \perp = \perp$
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*: $\perp \in A$
shows $Rep\ \perp = \perp$
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*: $\perp \in A$
shows $[\![x \neq \perp;\ x \in A]\!] \Longrightarrow Abs\ x \neq \perp$
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*: $\perp \in A$

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

6.4 HOLCF type definition package

use *pcpodef-package.ML*

end

7 Cfun: The type of continuous functions

theory *Cfun*
imports *Pcpodef*
uses (*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* => *'b*. *cont f*}
by (*simp add*: *Ex-cont adm-cont*)

syntax
Rep-CFun :: (*'a* −> *'b*) => (*'a* => *'b*) (-$- [*999*,*1000*] *999*)

Abs-CFun :: (*'a* => *'b*) => (*'a* −> *'b*) (**binder** *LAM* *10*)

syntax (*xsymbols*)
−> :: [*type*, *type*] => *type* ((- →/ -) [*1*,*0*]*0*)
LAM :: [*idts*, *'a* => *'b*] => (*'a* −> *'b*)
((*3*Λ-./ -) [*0*, *10*] *10*)
Rep-CFun :: (*'a* −> *'b*) => (*'a* => *'b*) ((-·-) [*999*,*1000*] *999*)

syntax (*HTML* **output**)
Rep-CFun :: (*'a* −> *'b*) => (*'a* => *'b*) ((-·-) [*999*,*1000*] *999*)

7.2 Class instances

lemma *UU-CFun*: $\perp \in$ *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*]

Additional lemma about the isomorphism between $'a \rightarrow 'b$ and *CFun*

lemma *Abs-CFun-inverse2*: *cont f* $\Longrightarrow$ *Rep-CFun* (*Abs-CFun f*) = *f*
by (*simp add*: *Abs-CFun-inverse CFun-def*)

Beta-equality for continuous functions

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 *ext-cfun*: $(\bigwedge x.\ f\cdot x = g\cdot x) \Longrightarrow f = g$
by (*simp add*: *Rep-CFun-inject* [*symmetric*] *ext*)

lemmas about application of continuous functions

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*

7.3 Continuity of application

lemma *cont-Rep-CFun1*: *cont* $(\lambda f.\ f\cdot x)$
by (*rule cont-Rep-CFun* [*THEN cont2cont-CF1L*])

lemma *cont-Rep-CFun2*: *cont* $(\lambda x.\ f\cdot x)$
apply (*rule-tac P* = *cont* **in** *CollectD*)
apply (*fold CFun-def*)
apply (*rule Rep-CFun*)

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*])

Extensionality wrt. $op \sqsubseteq$ in $'a \rightarrow 'b$

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

monotonicity of application

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

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 $'a \rightarrow 'b$

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

apply (*rule chainI*)
apply (*rule monofun-cfun*)
apply (*erule chainE*)
apply (*erule chainE*)
done

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

lemma *contlub-cfun*:
$[\![chain\ F;\ chain\ Y]\!] \Longrightarrow (\bigsqcup i.\ F\ i)\cdot(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ F\ i\cdot(Y\ i))$
apply (*simp only*: *contlub-cfun-fun*)
apply (*simp only*: *contlub-cfun-arg*)
apply (*rule diag-lub*)
apply (*erule monofun-Rep-CFun1* [*THEN ch2ch-monofun*])
apply (*erule monofun-Rep-CFun2* [*THEN ch2ch-monofun*])
done

lemma *cont-cfun*:
$[\![chain\ F;\ chain\ Y]\!] \Longrightarrow range\ (\lambda i.\ F\ i\cdot(Y\ i)) <<|\ (\bigsqcup i.\ F\ i)\cdot(\bigsqcup i.\ Y\ i)$
apply (*rule thelubE*)
apply (*simp only*: *ch2ch-Rep-CFun*)
apply (*simp only*: *contlub-cfun*)
done

strictness

lemma *strictI*: $f\cdot x = \bot \Longrightarrow f\cdot\bot = \bot$
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*: $chain\ F \Longrightarrow monofun\ (\lambda x.\ \bigsqcup i.\ F\ i\cdot x)$
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 $'a \rightarrow 'b$

lemma *ex-lub-cfun*:
$[\![chain\ F;\ chain\ Y]\!] \Longrightarrow (\bigsqcup j.\ \bigsqcup i.\ F\ j\cdot(Y\ i)) = (\bigsqcup i.\ \bigsqcup j.\ F\ j\cdot(Y\ i))$
by (*simp add*: *diag-lub ch2ch-Rep-CFunL ch2ch-Rep-CFunR*)

the lub of a chain of cont. functions is continuous

lemma *cont-lub-cfun*: $chain\ F \Longrightarrow cont\ (\lambda x.\ \bigsqcup i.\ F\ i\cdot x)$
apply (*rule cont2cont-lub*)
apply (*erule monofun-Rep-CFun* [*THEN ch2ch-monofun*])
apply (*rule cont-Rep-CFun2*)

done

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

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

7.4 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*)

for compatibility with old HOLCF-Version

lemma *inst-cfun-pcpo*: $\perp$ = (Λ *x*. $\perp$)
by (*simp add*: *inst-fun-pcpo* [*symmetric*] *Abs-CFun-strict*)

7.5 Continuity of application

cont2cont lemma for *Rep-CFun*

lemma *cont2cont-Rep-CFun*:
 $[\![$*cont f*; *cont t*$]\!]$ $\Longrightarrow$ *cont* (λ*x*. (*f x*)$\cdot$(*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-CF1L-rev*)
done

continuity simplification procedure

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

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

function application is strict in its first argument

lemma *Rep-CFun-strict1* [*simp*]: $\bot \cdot x = \bot$
by (*simp add*: *Rep-CFun-strict*)

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

7.6 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=⊥* **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

7.7 Identity and composition

consts
ID :: $'a \rightarrow 'a$
cfcomp :: $('b \rightarrow 'c) \rightarrow ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'c$

syntax @*oo* :: $['b \rightarrow 'c, 'a \rightarrow 'b] \Rightarrow 'a \rightarrow 'c$ (**infixr** *oo 100*)

translations *f1 oo f2* == *cfcomp*\$*f1*\$*f2*

defs
ID-def: $ID \equiv (\Lambda\ x.\ x)$
oo-def: $cfcomp \equiv (\Lambda\ f\ g\ x.\ f\cdot(g\cdot x))$

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

lemma *cfcomp1*: $(f\ oo\ g) = (\Lambda\ x.\ f\cdot(g\cdot x))$
by (*simp add*: *oo-def*)

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

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

lemma *ID2* [*simp*]: $f\ oo\ ID = f$
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*)

7.8 Strictified functions

defaultsort *pcpo*

consts
Istrictify :: $('a \rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
strictify :: $('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b$

defs
Istrictify-def: $Istrictify\ f\ x \equiv if\ x = \bot\ then\ \bot\ else\ f\cdot x$
strictify-def: $strictify \equiv (\Lambda\ f\ x.\ Istrictify\ f\ x)$

results about strictify

lemma *Istrictify1*: *Istrictify f* $\perp = \perp$
by (*simp add*: *Istrictify-def*)

lemma *Istrictify2*: $x \neq \perp \Longrightarrow$ *Istrictify f x* $= f \cdot x$
by (*simp add*: *Istrictify-def*)

lemma *cont-Istrictify1*: *cont* (λf. *Istrictify f x*)
apply (*case-tac* $x = \perp$)
apply (*simp add*: *Istrictify1*)
apply (*simp add*: *Istrictify2*)
done

lemma *monofun-Istrictify2*: *monofun* (λx. *Istrictify f x*)
apply (*rule monofunI*)
apply (*simp add*: *Istrictify-def monofun-cfun-arg*)
apply *clarify*
apply (*simp add*: *eq-UU-iff*)
done

lemma *contlub-Istrictify2*: *contlub* (λx. *Istrictify f x*)
apply (*rule contlubI*)
apply (*case-tac lub* (*range* Y) $= \perp$)
apply (*drule* (*1*) *chain-UU-I*)
apply (*simp add*: *Istrictify1 thelub-const*)
apply (*simp add*: *Istrictify2*)
apply (*simp add*: *contlub-cfun-arg*)
apply (*rule lub-equal2*)
apply (*rule chain-mono2* [*THEN exE*])
apply (*erule chain-UU-I-inverse2*)
apply (*assumption*)
apply (*blast intro*: *Istrictify2* [*symmetric*])
apply (*erule chain-monofun*)
apply (*erule monofun-Istrictify2* [*THEN ch2ch-monofun*])
done

lemmas *cont-Istrictify2* =
monocontlub2cont [*OF monofun-Istrictify2 contlub-Istrictify2*, *standard*]

lemma *strictify1* [*simp*]: *strictify*$\cdot f \cdot \perp = \perp$
apply (*unfold strictify-def*)
apply (*simp add*: *cont-Istrictify1 cont-Istrictify2*)
apply (*rule Istrictify1*)
done

lemma *strictify2* [*simp*]: $x \neq \perp \Longrightarrow$ *strictify*$\cdot f \cdot x = f \cdot x$
apply (*unfold strictify-def*)
apply (*simp add*: *cont-Istrictify1 cont-Istrictify2*)
apply (*erule Istrictify2*)
done

lemma *strictify-conv-if*: $strictify \cdot f \cdot x = (if\ x = \bot\ then\ \bot\ else\ f \cdot x)$
by *simp*

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*

8.2 Type $'a \times 'b$ is a partial order

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

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

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 Type $'a \times 'b$ 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* $\Rightarrow$ $'a$::*cpo* $*$ $'b$::*cpo*) $\Longrightarrow \exists x$. *range* S <<| x
by (*rule exI*, *erule lub-cprod*)

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

8.5 Type $'a \times 'b$ is pointed

lemma *minimal-cprod*: $(\bot, \bot) \sqsubseteq p$
by (*simp add*: *less-cprod-def*)

lemma *least-cprod*: *EX* x::$'a$::*pcpo* $*$ $'b$::*pcpo*. *ALL* y. $x \sqsubseteq y$
apply (*rule-tac* $x = (\bot, \bot)$ **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* (λx. (x, y))
apply (*rule contlubI*)
apply (*subst thelub-cprod*)
apply (*erule monofun-pair1* [*THEN ch2ch-monofun*])
apply (*simp add*: *thelub-const*)
done

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

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

done

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

consts
  cpair :: $'a \to 'b \to ('a * 'b)$
  cfst :: $('a * 'b) \to 'a$
  csnd :: $('a * 'b) \to 'b$
  csplit :: $('a \to 'b \to 'c) \to ('a * 'b) \to 'c$

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

translations
  <*x*, *y*, *z*> == <*x*, <*y*, *z*>>
  <*x*, *y*> == *cpair*\$*x*\$*y*

defs
  cpair-def: *cpair* $\equiv (\Lambda\ x\ y.\ (x,\ y))$
  cfst-def: *cfst* $\equiv (\Lambda\ p.\ fst\ p)$
  csnd-def: *csnd* $\equiv (\Lambda\ p.\ snd\ p)$

csplit-def: *csplit* ≡ (Λ *f p*. *f*·(*cfst*·*p*)·(*csnd*·*p*))

8.8 Syntax

syntax for *LAM* <*x*,*y*,*z*>.*e*

syntax
  -*LAM* :: [*patterns*, 'a ⇒ 'b] ⇒ ('a → 'b) ((*3LAM* <->./ -) [*0*, *10*] *10*)

translations
  LAM <*x*,*y*,*zs*>. *b* == *csplit*$(*LAM x*. *LAM* <*y*,*zs*>. *b*)
  LAM <*x*,*y*>. *LAM zs*. *b* <= *csplit*$(*LAM x y zs*. *b*)
  LAM <*x*,*y*>.*b* == *csplit*$(*LAM x y*. *b*)

syntax (*xsymbols*)
  -*LAM* :: [*patterns*, 'a => 'b] => ('a −> 'b) ((*3*Λ()<->./ -) [*0*, *10*] *10*)

syntax for Let

constdefs
  CLet :: 'a → ('a → 'b) → 'b
  CLet ≡ Λ *s f*. *f*·*s*

nonterminals
  Cletbinds *Cletbind*

syntax
  -*Cbind* :: [*pttrn*, 'a] => *Cletbind* ((*2*- =/ -) *10*)
  -*Cbindp* :: [*patterns*, 'a] => *Cletbind* ((*2*<-> =/ -) *10*)
     :: *Cletbind* => *Cletbinds* (-)
  -*Cbinds* :: [*Cletbind*, *Cletbinds*] => *Cletbinds* (-;/ -)
  -*CLet* :: [*Cletbinds*, 'a] => 'a ((*Let* (-)/ *in* (-)) *10*)

translations
  -*CLet* (-*Cbinds b bs*) *e* == -*CLet b* (-*CLet bs e*)
  Let x = *a in LAM ys*. *e* == *CLet*$*a*$(*LAM x ys*. *e*)
  Let x = *a in e* == *CLet*$*a*$(*LAM x*. *e*)
  Let <*xs*> = *a in e* == *CLet*$*a*$(*LAM* <*xs*>. *e*)

8.9 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*> ⟹ *a* = *aa* ∧ *b* = *ba*
by (*simp add*: *cpair-eq-pair*)

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

lemma *cpair-less*: (<*a*, *b*> ⊑ <*a'*, *b'*>) = (*a* ⊑ *a'* ∧ *b* ⊑ *b'*)

by (*simp add*: *cpair-eq-pair less-cprod-def*)

lemma *cpair-defined-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 add*: *cpair-defined-iff*)

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 add*: *inst-cprod-pcpo cpair-eq-pair*)

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

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

lemma *spair-lemma*:
  $<strictify\cdot(\Lambda\ b.\ a)\cdot b,\ strictify\cdot(\Lambda\ a.\ b)\cdot a> \in Sprod$
by (*simp add*: *Sprod-def strictify-conv-if cpair-strict*)

9.2 Definitions of constants

consts
  $sfst :: ('a ** 'b) \to 'a$
  $ssnd :: ('a ** 'b) \to 'b$
  $spair :: 'a \to 'b \to ('a ** 'b)$
  $ssplit :: ('a \to 'b \to 'c) \to ('a ** 'b) \to 'c$

defs
  sfst-def: $sfst \equiv \Lambda\ p.\ cfst\cdot(Rep\text{-}Sprod\ p)$

ssnd-def: $ssnd \equiv \Lambda\ p.\ csnd \cdot (Rep\text{-}Sprod\ p)$
spair-def: $spair \equiv \Lambda\ a\ b.\ Abs\text{-}Sprod$
$\quad <strictify \cdot (\Lambda\ b.\ a) \cdot b,\ strictify \cdot (\Lambda\ a.\ b) \cdot a>$
ssplit-def: $ssplit \equiv \Lambda\ f.\ strictify \cdot (\Lambda\ p.\ f \cdot (sfst \cdot p) \cdot (ssnd \cdot p))$

syntax
@*stuple* :: $['a,\ args] => 'a ** 'b$ $((1'(:\text{-},/\ \text{-}:')))$

translations
$(:x,\ y,\ z:)$ == $(:x,\ (:y,\ z:):)$
$(:x,\ y:)$ == $spair\$x\$y$

9.3 Case analysis

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

lemma *Exh-Sprod2*:
$z = \bot \lor (\exists\, a\ b.\ z = (:a,\ b:) \land a \neq \bot \land b \neq \bot)$
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* $x{=}cfst \cdot y$ **in** *exI*)
apply (*rule-tac* $x{=}csnd \cdot y$ **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*: $x = \bot \lor y = \bot \Longrightarrow (:x,\ y:) = \bot$
by *auto*

lemma *spair-strict-rev*: $(:x,\ y:) \neq \bot \Longrightarrow x \neq \bot \land y \neq \bot$
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 cpair-defined-iff 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*

9.5 Properties of *sfst* and *ssnd*

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

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

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

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

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

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

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

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

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* × '*b*. *cfst*·*p* = ⊥ ∨ *csnd*·*p* = ⊥}
by *simp*

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

10.2 Definitions of constructors

constdefs
 sinl :: '*a* → ('*a* ++ '*b*)
 sinl ≡ Λ *a*. *Abs-Ssum* <*a*, ⊥>

 sinr :: '*b* → ('*a* ++ '*b*)
 sinr ≡ Λ *b*. *Abs-Ssum* <⊥, *b*>

10.3 Properties of *sinl* and *sinr*

lemma *sinl-Abs-Ssum*: *sinl*·*a* = *Abs-Ssum* <*a*, ⊥>
by (*unfold sinl-def*, *simp add*: *cont-Abs-Ssum Ssum-def*)

lemma *sinr-Abs-Ssum*: *sinr*·*b* = *Abs-Ssum* <⊥, *b*>
by (*unfold sinr-def*, *simp add*: *cont-Abs-Ssum Ssum-def*)

lemma *Rep-Ssum-sinl*: *Rep-Ssum* (*sinl*·*a*) = <*a*, ⊥>
by (*unfold sinl-def*, *simp add*: *cont-Abs-Ssum Abs-Ssum-inverse Ssum-def*)

lemma *Rep-Ssum-sinr*: *Rep-Ssum* (*sinr*·*b*) = <⊥, *b*>
by (*unfold sinr-def*, *simp add*: *cont-Abs-Ssum Abs-Ssum-inverse Ssum-def*)

lemma *sinl-strict* [*simp*]: *sinl*·⊥ = ⊥
by (*simp add*: *sinl-Abs-Ssum Abs-Ssum-strict cpair-strict*)

lemma *sinr-strict* [*simp*]: *sinr*·⊥ = ⊥
by (*simp add*: *sinr-Abs-Ssum Abs-Ssum-strict cpair-strict*)

lemma *sinl-eq* [*simp*]: (*sinl*·*x* = *sinl*·*y*) = (*x* = *y*)
by (*simp add*: *sinl-Abs-Ssum Abs-Ssum-inject Ssum-def*)

lemma *sinr-eq* [*simp*]: (*sinr*·*x* = *sinr*·*y*) = (*x* = *y*)
by (*simp add*: *sinr-Abs-Ssum Abs-Ssum-inject Ssum-def*)

lemma *sinl-inject*: *sinl*·*x* = *sinl*·*y* ⟹ *x* = *y*
by (*rule sinl-eq* [*THEN iffD1*])

lemma *sinr-inject*: *sinr*·*x* = *sinr*·*y* ⟹ *x* = *y*

by (*rule sinr-eq* [*THEN iffD1*])

lemma *sinl-defined-iff* [*simp*]: $(sinl{\cdot}x = \bot) = (x = \bot)$
apply (*rule sinl-strict* [*THEN subst*])
apply (*rule sinl-eq*)
done

lemma *sinr-defined-iff* [*simp*]: $(sinr{\cdot}x = \bot) = (x = \bot)$
apply (*rule sinr-strict* [*THEN subst*])
apply (*rule sinr-eq*)
done

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

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

10.4 Case analysis

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

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

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 cpair-less eq-UU-iff*)

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 cpair-less eq-UU-iff*)

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

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

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*=$\bot$ **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*=$\bot$ **in** *exI*, *simp*)
apply *simp*
apply *simp*
done

lemma *ssum-chain-lemma*:
$chain\ Y \Longrightarrow (\exists\, A.\ chain\ A \wedge Y = (\lambda i.\ sinl{\cdot}(A\ i))) \vee$
$(\exists\, B.\ chain\ B \wedge Y = (\lambda i.\ sinr{\cdot}(B\ i)))$
apply (*rule-tac p=lub* (*range Y*) **in** *ssumE2*)
apply (*rule disjI1*)
apply (*rule-tac x*=$\lambda i.\ cfst{\cdot}(Rep\text{-}Ssum\ (Y\ i))$ **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*=$\lambda i.\ csnd{\cdot}(Rep\text{-}Ssum\ (Y\ i))$ **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

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

rewrites for *Iwhen*

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

constdefs
$sscase :: ('a \to 'c) \to ('b \to 'c) \to ('a ++ 'b) \to 'c$
$sscase \equiv \Lambda\ f\ g\ s.\ Iwhen\ f\ g\ s$

translations
case s of sinl$x => t1 | sinr$y => t2 == sscase$(LAM x. t1)$(LAM y. t2)$s

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 Sum-Type Datatype*
begin

defaultsort *cpo*

11.1 Definition of new type for lifting

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

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

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

11.2 Ordering on type $'a\ u$

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

11.3 Type $'a\ u$ is a partial order

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

lemma *antisym-less-up*: $[\![(x::'a\ u) \sqsubseteq y;\ y \sqsubseteq x]\!] \Longrightarrow x = y$
apply (*simp add*: *less-up-def split*: *u.split-asm*)
apply (*erule* (*1*) *antisym-less*)
done

lemma *trans-less-up*: $[\![(x::'a\ u) \sqsubseteq y;\ y \sqsubseteq z]\!] \Longrightarrow x \sqsubseteq z$
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 Type $'a\ u$ is a cpo

lemma *is-lub-Iup*:
$range\ S <<|\ x \Longrightarrow range\ (\lambda i.\ Iup\ (S\ i)) <<|\ Iup\ x$
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 $'a\ u$ elements

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

by (*case-tac z*, *simp-all*)

lemma *up-lemma2*:
 $[\![chain\ Y;\ Y\,j \neq Ibottom]\!] \Longrightarrow Y\ (i + j) \neq Ibottom$
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*:
 $[\![chain\ Y;\ Y\,j \neq Ibottom]\!] \Longrightarrow Iup\ (THE\ a.\ Iup\ a = Y\ (i + j)) = Y\ (i + j)$
by (*rule up-lemma1* [*OF up-lemma2*])

lemma *up-lemma4*:
 $[\![chain\ Y;\ Y\,j \neq Ibottom]\!] \Longrightarrow chain\ (\lambda i.\ THE\ a.\ Iup\ a = Y\ (i + j))$
apply (*rule chainI*)
apply (*rule Iup-less* [*THEN iffD1*])
apply (*subst up-lemma3*, *assumption*+)+
apply (*simp add*: *chainE*)
done

lemma *up-lemma5*:
 $[\![chain\ Y;\ Y\,j \neq Ibottom]\!] \Longrightarrow$
  $(\lambda i.\ Y\ (i + j)) = (\lambda i.\ Iup\ (THE\ a.\ Iup\ a = Y\ (i + j)))$
by (*rule ext*, *rule up-lemma3* [*symmetric*])

lemma *up-lemma6*:
 $[\![chain\ Y;\ Y\,j \neq Ibottom]\!]$
   $\Longrightarrow range\ Y <<|\ Iup\ (\bigsqcup i.\ THE\ a.\ Iup\ a = Y(i + j))$
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-cases*:
 $chain\ Y \Longrightarrow$
 $(\exists A.\ chain\ A \wedge lub\ (range\ Y) = Iup\ (lub\ (range\ A)) \wedge$
 $(\exists j.\ \forall i.\ Y\ (i + j) = Iup\ (A\ i))) \vee (Y = (\lambda i.\ Ibottom))$
apply (*rule disjCI*)
apply (*simp add*: *expand-fun-eq*)
apply (*erule exE*, *rename-tac j*)
apply (*rule-tac x=λi. 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* $\Rightarrow$ $'a$ u) $\Longrightarrow$ $\exists\, x$. *range* Y <<| x
apply (*frule up-chain-cases*, *safe*)
apply (*rule-tac* x=*Iup* (*lub* (*range* A)) **in** *exI*)
apply (*erule-tac* *j1*=j **in** *is-lub-range-shift* [*THEN iffD1*])
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 Type $'a$ u is pointed

lemma *least-up*: $\exists\, x$::$'a$ u. $\forall\, y$. $x \sqsubseteq y$
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*: $\perp$ = *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* (λf. *Ifup f x*)
by (*induct x*, *simp-all*)

lemma *monofun-Ifup2*: *monofun* (λx. *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* ($\lambda x.$ *Ifup f x*)
apply (*rule contI*)
apply (*frule up-chain-cases*, *safe*)
apply (*rule-tac j1=j* **in** *is-lub-range-shift* [*THEN iffD1*])
apply (*erule monofun-Ifup2* [*THEN ch2ch-monofun*])
apply (*simp add*: *cont-cfun-arg*)
apply (*simp add*: *thelub-const lub-const*)
done

11.7 Continuous versions of constants

constdefs
  $up :: 'a \to 'a\ u$
  $up \equiv \Lambda\ x.\ Iup\ x$

  $fup :: ('a \to 'b{::}pcpo) \to 'a\ u \to 'b$
  $fup \equiv \Lambda\ f\ p.\ Ifup\ f\ p$

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

continuous versions of lemmas for $'a\ u$

lemma *Exh-Up*: $z = \bot \vee (\exists\, x.\ z = up{\cdot}x)$
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*]: $up{\cdot}x \neq \bot$
by (*simp add*: *up-def cont-Iup inst-up-pcpo*)

lemma *not-up-less-UU* [*simp*]: $\neg\ up{\cdot}x \sqsubseteq \bot$
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*: $[\![p = \bot \Longrightarrow Q;\ \bigwedge x.\ p = up{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*case-tac p*)
apply (*simp add*: *inst-up-pcpo*)
apply (*simp add*: *up-def cont-Iup*)

done

lemma *fup1* [*simp*]: *fup*·*f*·⊥ = ⊥
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 Datatype*
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*

constdefs
  undiscr :: (*'a*::*type*)*discr* => *'a*
  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*]

constdefs

$Def :: 'a \Rightarrow 'a\ lift$
$Def\ x \equiv Abs\text{-}lift\ (up \cdot (Discr\ x))$

13.1 Lift as a datatype

lemma *lift-distinct1*: $\perp \neq Def\ x$
by (*simp add*: *Def-def Abs-lift-inject lift-def inst-lift-pcpo*)

lemma *lift-distinct2*: $Def\ x \neq \perp$
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*: $[\![P\ \perp;\ \bigwedge x.\ P\ (Def\ x)]\!] \Longrightarrow P\ y$
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*

$\perp$ and *Def*

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

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

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

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

For $x \neq \perp$ 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;
```

$\rangle\rangle$

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 add*: *eq-UU-iff*)
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-CF1L*])

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-CF1L*])

13.3 Further operations

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

  flift2 :: $('a \Rightarrow 'b) \Rightarrow ('a\ lift \rightarrow 'b\ lift)$
  flift2 $f \equiv FLIFT\ x.\ Def\ (f\ x)$

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

13.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

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

apply (*induct x*)
apply *simp*
apply *simp*
apply (*rule cont-id* [*THEN cont2cont-CF1L*])
done

lemma *cont-lift-case2*: *cont* (λx. *lift-case* $\perp$ *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*:
$[\![\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)]\!] \Longrightarrow cont\ (\lambda x.\ FLIFT\ y.\ f\ x\ y)$
apply (*rule cont-flift1* [*THEN cont2cont-app3*])
apply (*simp add*: *cont2cont-lambda*)
done

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

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

lemma *flift1-strict* [*simp*]: *flift1 f*$\cdot\perp = \perp$
by (*simp add*: *flift1-def cont-lift-case2*)

lemma *flift2-strict* [*simp*]: *flift2 f*$\cdot\perp = \perp$
by (*simp add*: *flift2-def*)

lemma *flift2-defined* [*simp*]: $x \neq \perp \Longrightarrow (flift2\ f)\cdot x \neq \perp$
by (*erule lift-definedE*, *simp*)

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 ⟪
```
val cont-lemmas2 = cont-lemmas1 @ thms cont-lemmas-ext;

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

local val flift1-def = thm flift1-def
in 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*

constdefs
ONE :: *one*
ONE ≡ *Def* ()

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

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* $\neq \perp$ $\perp \neq$ *ONE*
apply (*unfold ONE-def*)
apply *simp-all*
done

end

15 Tr: The type of lifted booleans

theory *Tr*
imports *Lift*
begin

defaultsort *pcpo*

types
tr = *bool lift*

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

consts
TT :: *tr*
FF :: *tr*
Icifte :: *tr* −> ′*c* −> ′*c* −> ′*c*
trand :: *tr* −> *tr* −> *tr*
tror :: *tr* −> *tr* −> *tr*
neg :: *tr* −> *tr*
If2 :: *tr*=>′*c*=>′*c*=>′*c*

syntax @*cifte* :: *tr*=>′*c*=>′*c*=>′*c* ((*3If* -/ (*then* -/ *else* -) *fi*) *60*)
@*andalso* :: *tr* => *tr* => *tr* (- *andalso* - [*36*,*35*] *35*)
@*orelse* :: *tr* => *tr* => *tr* (- *orelse* - [*31*,*30*] *30*)

translations
x andalso y == *trand*$*x*$*y*
x orelse y == *tror*$*x*$*y*
If b then e1 else e2 fi == *Icifte*$*b*$*e1*$*e2*
defs
TT-def: *TT*==*Def True*
FF-def: *FF*==*Def False*
neg-def: *neg* == *flift2 Not*
ifte-def: *Icifte* == (*LAM b t e*. *flift1*(%*b*. *if b then t else e*)$*b*)
andalso-def: *trand* == (*LAM x y*. *If x then y else FF fi*)

orelse-def: *tror == (LAM x y. If x then TT else y fi)*
If2-def: *If2 Q x y == If Q then x else y fi*

Exhaustion and Elimination for type *tr*

lemma *Exh-tr*: *t=UU | t = TT | t = FF*
apply (*unfold FF-def TT-def*)
apply (*induct-tac t*)
apply *fast*
apply *fast*
done

lemma *trE*: [| *p=UU ==> 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 << UU ~FF << UU ~TT << FF ~FF << TT*
by (*simp-all add*: *tr-defs*)

lemma *dist-eq-tr* [*simp*]: *TT~=UU FF~=UU TT~=FF UU~=TT UU~=FF FF~=TT*
by (*simp-all add*: *tr-defs*)

lemmas about andalso, orelse, neg and if

lemma *ifte-thms* [*simp*]:
If UU then e1 else e2 fi = UU
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*
(*UU andalso y*) = *UU*
(*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*
  (*UU orelse y*) = *UU*
  (*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$*UU* = *UU*
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*=*UU* --> *P UU*) & (*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* [*symmetric* (*thm If2-def*)])
    THEN' (*split-tac* [*thm split-If2*])
⟫

15.1 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*:
!!*t*.[|*t*~=*UU*|]==> ((*t andalso s*)=*FF*)=(*t*=*FF* | *s*=*FF*)
apply (*rule-tac p* = *t* **in** *trE*)
apply *simp-all*
done

lemma *andalso-and*: [|*t*~=*UU*|]==> ((*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* ~= *FF*) = *x*
by (*simp add*: *FF-def*)

lemma *Def-bool2* [*simp*]: (*Def x* = *FF*) = (~*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* ~= *TT*) = (~*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 admissibility

The following rewrite rules for admissibility should in the future be replaced by a more general admissibility test that also checks chain-finiteness, of which these lemmata are specific examples

lemma *adm-trick-1*: (*x*~=*FF*) = (*x*=*TT*|*x*=*UU*)
apply (*rule-tac p* = *x* **in** *trE*)
apply (*simp-all*)
done

lemma *adm-trick-2*: (*x*~=*TT*) = (*x*=*FF*|*x*=*UU*)
apply (*rule-tac p* = *x* **in** *trE*)
apply (*simp-all*)
done

lemmas *adm-tricks* = *adm-trick-1 adm-trick-2*

lemma *adm-nTT* [*simp*]: *cont*(*f*) ==> *adm* (%*x*. (*f x*)~=*TT*)
by (*simp add*: *adm-tricks*)

lemma *adm-nFF* [*simp*]: *cont*(*f*) ==> *adm* (%*x*. (*f x*)~=*FF*)
by (*simp add*: *adm-tricks*)

end

16 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun Cprod Adm*
begin

defaultsort *pcpo*

16.1 Definitions

consts
$iterate :: nat \Rightarrow ('a \to 'a) \Rightarrow 'a \Rightarrow 'a$
$Ifix :: ('a \to 'a) \Rightarrow 'a$
$fix :: ('a \to 'a) \to 'a$
$admw :: ('a \Rightarrow bool) \Rightarrow bool$

primrec
$iterate\text{-}0$: $iterate\ 0\ F\ x = x$
$iterate\text{-}Suc$: $iterate\ (Suc\ n)\ F\ x = F{\cdot}(iterate\ n\ F\ x)$

defs
$Ifix\text{-}def$: $Ifix \equiv \lambda F.\ \bigsqcup i.\ iterate\ i\ F\ \bot$
$fix\text{-}def$: $fix \equiv \Lambda\ F.\ Ifix\ F$

$admw\text{-}def$: $admw\ P \equiv \forall F.\ (\forall n.\ P\ (iterate\ n\ F\ \bot)) \longrightarrow P\ (\bigsqcup i.\ iterate\ i\ F\ \bot)$

16.2 Binder syntax for *fix*

syntax
@*FIX* :: ('*a* => '*a*) => '*a* (**binder** *FIX* *10*)
@*FIXP* :: [*patterns*, '*a*] => '*a* ((*3FIX* <->./ -) [*0*, *10*] *10*)

syntax (*xsymbols*)
FIX :: [*idt*, '*a*] => '*a* ((*3*μ-./ -) [*0*, *10*] *10*)
@*FIXP* :: [*patterns*, '*a*] => '*a* ((*3*μ()<->./ -) [*0*, *10*] *10*)

translations
FIX x. LAM y. t == *fix*·(*LAM x y. t*)
FIX x. t == *fix*·(*LAM x. t*)
FIX <*xs*>. *t* == *fix*·(*LAM* <*xs*>. *t*)

16.3 Properties of *iterate* and *fix*

derive inductive properties of iterate from primitive recursion

lemma $iterate\text{-}Suc2$: $iterate\ (Suc\ n)\ F\ x = iterate\ n\ F\ (F{\cdot}x)$
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 $chain\text{-}iterate2$: $x \sqsubseteq F{\cdot}x \Longrightarrow chain\ (\lambda i.\ iterate\ i\ F\ x)$
by (*rule chainI*, *induct-tac i*, *auto elim*: *monofun-cfun-arg*)

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

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

lemma *Ifix-eq*: *Ifix F* = *F*·(*Ifix F*)
apply (*unfold Ifix-def*)
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 *Ifix-least*: *F*·*x* = *x* $\Longrightarrow$ *Ifix F* $\sqsubseteq$ *x*
apply (*unfold Ifix-def*)
apply (*rule is-lub-thelub*)
apply (*rule chain-iterate*)
apply (*rule ub-rangeI*)
apply (*induct-tac i*)
apply *simp*
apply *simp*
apply (*erule subst*)
apply (*erule monofun-cfun-arg*)
done

continuity of *iterate*

lemma *cont-iterate1*: *cont* (λ*F*. *iterate n F x*)
by (*induct-tac n*, *simp-all*)

lemma *cont-iterate2*: *cont* (λ*x*. *iterate n F x*)
by (*induct-tac n*, *simp-all*)

lemma *cont-iterate*: *cont* (*iterate n*)
by (*rule cont-iterate1* [*THEN cont2cont-lambda*])

lemmas *monofun-iterate2* = *cont-iterate2* [*THEN cont2mono*, *standard*]
lemmas *contlub-iterate2* = *cont-iterate2* [*THEN cont2contlub*, *standard*]

continuity of *Ifix*

lemma *cont-Ifix*: *cont Ifix*
apply (*unfold Ifix-def*)
apply (*rule cont2cont-lub*)
apply (*rule ch2ch-fun-rev*)
apply (*rule chain-iterate*)
apply (*rule cont-iterate1*)
done

propagate properties of *Ifix* to its continuous counterpart

lemma *fix-eq*: *fix*·*F* = *F*·(*fix*·*F*)
apply (*unfold fix-def*)
apply (*simp add*: *cont-Ifix*)
apply (*rule Ifix-eq*)
done

lemma *fix-least*: $F \cdot x = x \Longrightarrow fix \cdot F \sqsubseteq x$
apply (*unfold fix-def*)
apply (*simp add*: *cont-Ifix*)
apply (*erule Ifix-least*)
done

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 (*erule allE*)
apply (*erule mp*)
apply (*rule 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*])

direct connection between *fix* and iteration without *Ifix*

lemma *fix-def2*: $fix \cdot F = (\bigsqcup i.\ iterate\ i\ F\ \bot)$
apply (*unfold fix-def*)
apply (*simp add*: *cont-Ifix*)
apply (*simp add*: *Ifix-def*)
done

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 (*rule fix-eq* [*THEN trans*], *simp*)

16.4 Admissibility and fixed point induction

an admissible formula is also weak admissible

lemma *adm-impl-admw*: $adm\ P \Longrightarrow admw\ P$
apply (*unfold admw-def*)
apply (*intro strip*)
apply (*erule admD*)
apply (*rule chain-iterate*)
apply *assumption*
done

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

lemma *adm-chfindom*: $adm\ (\lambda(u{::}'a{::}cpo \rightarrow 'b{::}chfin).\ P(u{\cdot}s))$
apply (*unfold adm-def*)
apply (*intro strip*)
apply (*drule chfin-Rep-CFunR*)
apply (*erule-tac* $x = s$ **in** *allE*)
apply *clarsimp*
done

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*)
apply (*rule chain-iterate*)
apply (*rule allI*)
apply (*induct-tac i*)
apply *simp*
apply *simp*
done

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

computational induction for weak admissible formulae

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

by (*simp add*: *fix-def2 admw-def*)

lemma *def-wfix-ind*:
$[\![f \equiv fix \cdot F;\ admw\ P;\ \forall\, n.\ P\ (iterate\ n\ F\ \bot)]\!] \Longrightarrow P\ f$
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 (*fixrec-package.ML*)
begin

17.1 Maybe monad type

defaultsort *cpo*

types $'a\ maybe = one\ {+}{+}\ 'a\ u$

constdefs
$fail :: 'a\ maybe$
$fail \equiv sinl \cdot ONE$

$return :: 'a \to 'a\ maybe$
$return \equiv sinr\ oo\ up$

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 (*rule-tac p=p* **in** *ssumE*, *simp*)
apply (*rule-tac p=x* **in** *oneE*, *simp*, *simp*)
apply (*rule-tac p=y* **in** *upE*, *simp*, *simp*)
done

17.2 Monadic bind operator

constdefs
$bind :: 'a\ maybe \to ('a \to 'b\ maybe) \to 'b\ maybe$
$bind \equiv \Lambda\ m\ f.\ sscase \cdot sinl \cdot (fup \cdot f) \cdot m$

syntax
$\text{-}bind :: 'a\ maybe \Rightarrow ('a \to 'b\ maybe) \Rightarrow 'b\ maybe$
((- >>= -) [*50*, *51*] *50*)

translations $m >>= k == bind \cdot m \cdot k$

nonterminals
maybebind maybebinds

syntax
-MBIND :: *pttrn* ⇒ *'a maybe* ⇒ *maybebind* ((*2*- <−/ -) *10*)
:: *maybebind* ⇒ *maybebinds* (-)

-MBINDS :: [*maybebind*, *maybebinds*] ⇒ *maybebinds* (-;/ -)
-MDO :: [*maybebinds*, *'a maybe*] ⇒ *'a maybe* ((*do* -;/ (-)) *10*)

translations
-MDO (*-MBINDS b bs*) *e* == *-MDO b* (*-MDO bs e*)
do (*x*,*y*) <− *m*; *e* == *m* >>= (*LAM* <*x*,*y*>. *e*)
do x <− *m*; *e* == *m* >>= (*LAM x*. *e*)

monad laws

lemma *bind-strict* [*simp*]: *UU* >>= *f* = *UU*
by (*simp add*: *bind-def*)

lemma *bind-fail* [*simp*]: *fail* >>= *f* = *fail*
by (*simp add*: *bind-def fail-def*)

lemma *left-unit* [*simp*]: (*return*·*a*) >>= *k* = *k*·*a*
by (*simp add*: *bind-def return-def*)

lemma *right-unit* [*simp*]: *m* >>= *return* = *m*
by (*rule-tac p*=*m* **in** *maybeE*, *simp-all*)

lemma *bind-assoc* [*simp*]:
(*do b* <− (*do a* <− *m*; *k*·*a*); *h*·*b*) = (*do a* <− *m*; *b* <− *k*·*a*; *h*·*b*)
by (*rule-tac p*=*m* **in** *maybeE*, *simp-all*)

17.3 Run operator

constdefs
run:: *'a*::*pcpo maybe* → *'a*
run ≡ *sscase*·⊥·(*fup*·*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 fail-def*)

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

17.4 Monad plus operator

constdefs
  $mplus$ :: $'a\ maybe \rightarrow 'a\ maybe \rightarrow 'a\ maybe$
  $mplus \equiv \Lambda\ m1\ m2.\ sscase\cdot(\Lambda\ x.\ m2)\cdot(fup\cdot return)\cdot m1$

syntax $+++$ :: $'a\ maybe \Rightarrow 'a\ maybe \Rightarrow 'a\ maybe$ (**infixr** 65)
translations $x +++ y == mplus\cdot x\cdot y$

rewrite rules for mplus

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

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

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

17.5 Match functions for built-in types

defaultsort *pcpo*

constdefs
  $match\text{-}UU$ :: $'a \rightarrow unit\ maybe$
  $match\text{-}UU \equiv \Lambda\ x.\ fail$

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

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

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

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

  $match\text{-}up$ :: $'a::cpo\ u \rightarrow 'a\ maybe$
  $match\text{-}up \equiv fup\cdot return$

  $match\text{-}ONE$ :: $one \rightarrow unit\ maybe$
  $match\text{-}ONE \equiv flift1\ (\lambda u.\ return\cdot())$

$match\text{-}TT :: tr \rightarrow unit\ maybe$
$match\text{-}TT \equiv flift1\ (\lambda b.\ if\ b\ then\ return\cdot()\ else\ fail)$

$match\text{-}FF :: tr \rightarrow unit\ maybe$
$match\text{-}FF \equiv flift1\ (\lambda b.\ if\ b\ then\ fail\ else\ return\cdot())$

lemma *match-UU-simps* [*simp*]:
$match\text{-}UU\cdot x = fail$
by (*simp add*: *match-UU-def*)

lemma *match-cpair-simps* [*simp*]:
$match\text{-}cpair\cdot{<}x,y{>} = return\cdot{<}x,y{>}$
by (*simp add*: *match-cpair-def*)

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

lemma *match-sinl-simps* [*simp*]:
$x \neq \bot \Longrightarrow match\text{-}sinl\cdot(sinl\cdot x) = return\cdot x$
$x \neq \bot \Longrightarrow match\text{-}sinl\cdot(sinr\cdot x) = fail$
$match\text{-}sinl\cdot\bot = \bot$
by (*simp-all add*: *match-sinl-def*)

lemma *match-sinr-simps* [*simp*]:
$x \neq \bot \Longrightarrow match\text{-}sinr\cdot(sinr\cdot x) = return\cdot x$
$x \neq \bot \Longrightarrow match\text{-}sinr\cdot(sinl\cdot x) = fail$
$match\text{-}sinr\cdot\bot = \bot$
by (*simp-all add*: *match-sinr-def*)

lemma *match-up-simps* [*simp*]:
$match\text{-}up\cdot(up\cdot x) = return\cdot x$
$match\text{-}up\cdot\bot = \bot$
by (*simp-all add*: *match-up-def*)

lemma *match-ONE-simps* [*simp*]:
$match\text{-}ONE\cdot ONE = return\cdot()$
$match\text{-}ONE\cdot\bot = \bot$
by (*simp-all add*: *ONE-def match-ONE-def*)

lemma *match-TT-simps* [*simp*]:
$match\text{-}TT\cdot TT = return\cdot()$
$match\text{-}TT\cdot FF = fail$
$match\text{-}TT\cdot\bot = \bot$
by (*simp-all add*: *TT-def FF-def match-TT-def*)

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

$match\text{-}FF \cdot FF = return \cdot ()$
$match\text{-}FF \cdot TT = fail$
$match\text{-}FF \cdot \bot = \bot$
by (*simp-all add*: *TT-def FF-def match-FF-def*)

17.6 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*

ML ⟪
val cpair-equalI = *thm cpair-equalI*;
val cpair-eqD1 = *thm cpair-eqD1*;
val cpair-eqD2 = *thm cpair-eqD2*;
val ssubst-lhs = *thm ssubst-lhs*;
⟫

17.7 Initializing the fixrec package

use *fixrec-package.ML*

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*]: $abs\cdot(rep\cdot y) = y$

lemma (**in** *iso*) *swap*: *iso rep abs*
by (*rule iso.intro* [*OF rep-iso abs-iso*])

lemma (**in** *iso*) *abs-strict*: $abs\cdot\bot = \bot$
proof −
  have $\bot \sqsubseteq rep\cdot\bot$ **..**
  hence $abs\cdot\bot \sqsubseteq abs\cdot(rep\cdot\bot)$ **by** (*rule monofun-cfun-arg*)
  hence $abs\cdot\bot \sqsubseteq \bot$ **by** *simp*
  thus *?thesis* **by** (*rule UU-I*)
qed

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

lemma (**in** *iso*) *abs-defin'*: $abs\cdot z = \bot \Longrightarrow z = \bot$
proof −
  assume A: $abs\cdot z = \bot$
  have $z = rep\cdot(abs\cdot z)$ **by** *simp*
  also have $\ldots = rep\cdot\bot$ **by** (*simp only*: A)
  also note *rep-strict*
  finally show $z = \bot$ **.**
qed

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

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

lemma (**in** *iso*) *rep-defined*: $z \neq \bot \Longrightarrow rep\cdot z \neq \bot$
by (*erule contrapos-nn*, *erule rep-defin'*)

lemma (**in** *iso*) *iso-swap*: $(x = abs\cdot y) = (rep\cdot x = y)$
proof
  assume $x = abs\cdot y$
  hence $rep\cdot x = rep\cdot(abs\cdot y)$ **by** *simp*
  thus $rep\cdot x = y$ **by** *simp*
next
  assume $rep\cdot x = y$
  hence $abs\cdot(rep\cdot x) = abs\cdot y$ **by** *simp*
  thus $x = abs\cdot y$ **by** *simp*
qed

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 intro*: *sinl-defined*)
apply (*force intro*: *sinr-defined*)
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-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*
uses
holcf-logic.ML
cont-consts.ML
domain/library.ML
domain/syntax.ML
domain/axioms.ML
domain/theorems.ML
domain/extender.ML
domain/interface.ML
adm-tac.ML

begin

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

end