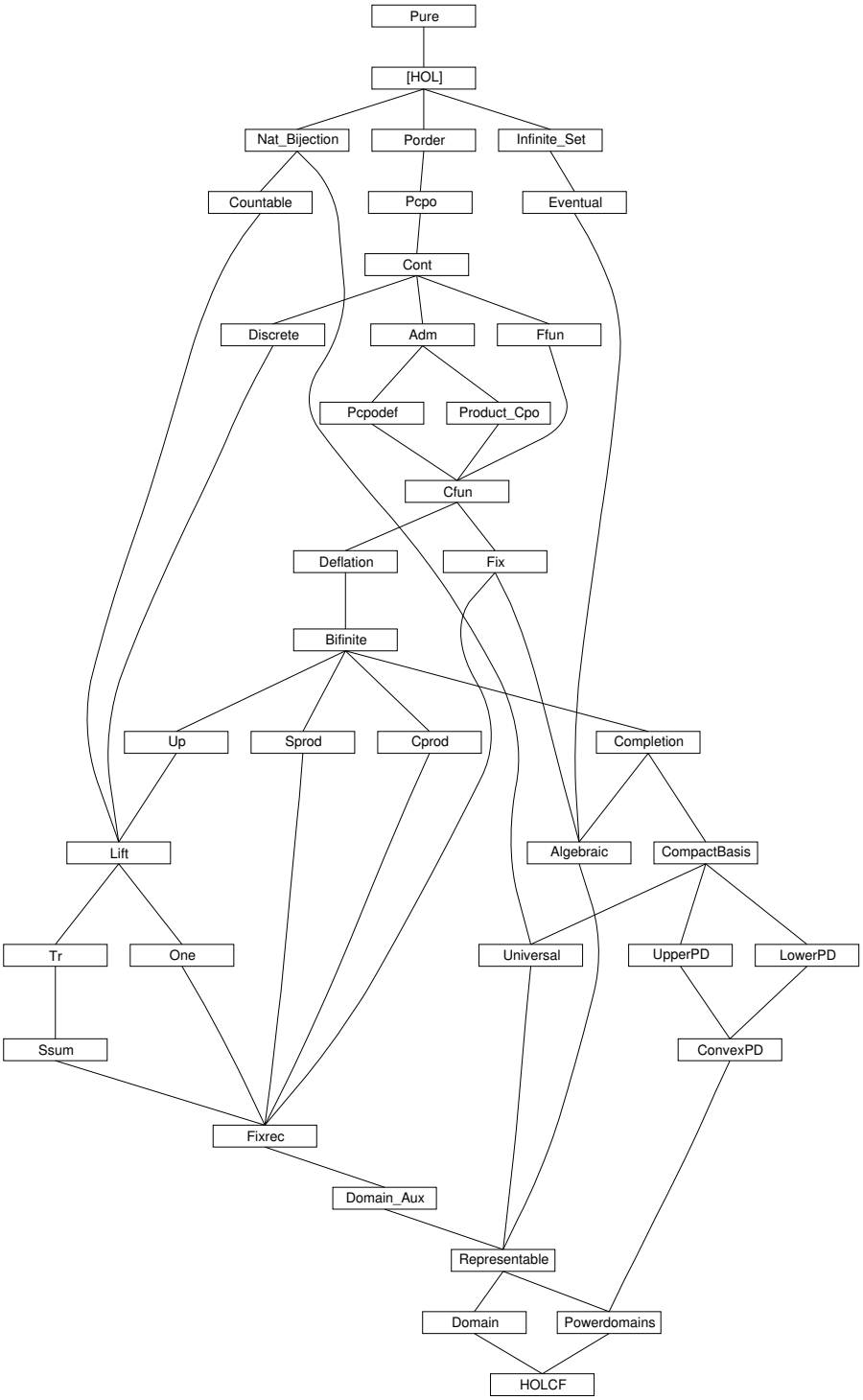
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

June 21, 2010

Contents


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

1 Porder: Partial orders

theory *Porder*
imports *Main*
begin

1.1 Type class for partial orders

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

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

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

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

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

end

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

minimal fixes least element

lemma *minimal2UU*[*OF allI*] : $\forall x.\ uu \sqsubseteq x \Longrightarrow uu = (THE\ u.\ \forall y.\ u \sqsubseteq y)$
  by (*blast intro*: *theI2 below-antisym*)

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

lemma *below-antisym-inverse*: $x = y \Longrightarrow x \sqsubseteq y \wedge y \sqsubseteq x$
  by *simp*

lemma *box-below*: $a \sqsubseteq b \Longrightarrow c \sqsubseteq a \Longrightarrow b \sqsubseteq d \Longrightarrow c \sqsubseteq d$
  by (*rule below-trans* [*OF below-trans*])

lemma *po-eq-conv*: $x = y \longleftrightarrow x \sqsubseteq y \wedge y \sqsubseteq x$
  by (*fast intro*!: *below-antisym*)

lemma *rev-below-trans*: $y \sqsubseteq z \Longrightarrow x \sqsubseteq y \Longrightarrow x \sqsubseteq z$
  by (*rule below-trans*)

lemma *not-below2not-eq*: $\neg\ x \sqsubseteq y \Longrightarrow x \neq y$
by *auto*

end

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

context *po*
begin

1.2 Upper bounds

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

lemma *is-ubI*: $(\bigwedge x.\ x \in S \Longrightarrow x \sqsubseteq u) \Longrightarrow S <| u$
by (*simp add*: *is-ub-def*)

lemma *is-ubD*: $[\![S <| u; x \in S]\!] \Longrightarrow x \sqsubseteq u$
by (*simp add*: *is-ub-def*)

lemma *ub-imageI*: $(\bigwedge x.\ x \in S \Longrightarrow f\,x \sqsubseteq u) \Longrightarrow (\lambda x.\ f\,x)\ `\ S <| u$
unfolding *is-ub-def* **by** *fast*

lemma *ub-imageD*: $[\![f\ `\ S <| u; x \in S]\!] \Longrightarrow f\,x \sqsubseteq u$
unfolding *is-ub-def* **by** *fast*

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

lemma *ub-rangeD*: $range\ S <| x \Longrightarrow S\ i \sqsubseteq x$
unfolding *is-ub-def* **by** *fast*

lemma *is-ub-empty* [*simp*]: $\{\} <| u$
unfolding *is-ub-def* **by** *fast*

lemma *is-ub-insert* [*simp*]: $(insert\ x\ A) <| y = (x \sqsubseteq y \wedge A <| y)$
unfolding *is-ub-def* **by** *fast*

lemma *is-ub-upward*: $[\![S <| x; x \sqsubseteq y]\!] \Longrightarrow S <| y$
unfolding *is-ub-def* **by** (*fast intro*: *below-trans*)

1.3 Least upper bounds

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

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

end

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

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

translations
LUB x:*A*. *t* == *CONST lub* ((%*x*. *t*) ' *A*)

context *po*
begin

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

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

access to some definition as inference rule

lemma *is-lubD1*: *S* <<| *x* ⟹ *S* <| *x*
unfolding *is-lub-def* **by** *fast*

lemma *is-lub-lub*: ⟦*S* <<| *x*; *S* <| *u*⟧ ⟹ *x* ⊑ *u*
unfolding *is-lub-def* **by** *fast*

lemma *is-lubI*: ⟦*S* <| *x*; ⋀*u*. *S* <| *u* ⟹ *x* ⊑ *u*⟧ ⟹ *S* <<| *x*
unfolding *is-lub-def* **by** *fast*

lubs are unique

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

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

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

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

lemma *is-lub-singleton*: $\{x\} <<| x$
by (*simp add*: *is-lub-def*)

lemma *lub-singleton* [*simp*]: $lub\ \{x\} = x$
by (*rule thelubI* [*OF is-lub-singleton*])

lemma *is-lub-bin*: $x \sqsubseteq y \Longrightarrow \{x, y\} <<| y$
by (*simp add*: *is-lub-def*)

lemma *lub-bin*: $x \sqsubseteq y \Longrightarrow lub\ \{x, y\} = y$
by (*rule is-lub-bin* [*THEN thelubI*])

lemma *is-lub-maximal*: $[\![S <| x; x \in S]\!] \Longrightarrow S <<| x$
by (*erule is-lubI*, *erule* (*1*) *is-ubD*)

lemma *lub-maximal*: $[\![S <| x; x \in S]\!] \Longrightarrow lub\ S = x$
by (*rule is-lub-maximal* [*THEN thelubI*])

1.4 Countable chains

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

lemma *chainI*: $(\bigwedge i.\ Y\ i \sqsubseteq Y\ (Suc\ i)) \Longrightarrow chain\ Y$
unfolding *chain-def* **by** *fast*

lemma *chainE*: $chain\ Y \Longrightarrow Y\ i \sqsubseteq Y\ (Suc\ i)$
unfolding *chain-def* **by** *fast*

chains are monotone functions

lemma *chain-mono-less*: $[\![chain\ Y; i < j]\!] \Longrightarrow Y\ i \sqsubseteq Y\ j$
by (*erule less-Suc-induct*, *erule chainE*, *erule below-trans*)

lemma *chain-mono*: $[\![chain\ Y; i \leq j]\!] \Longrightarrow Y\ i \sqsubseteq Y\ j$
by (*cases* $i = j$, *simp*, *simp add*: *chain-mono-less*)

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

technical lemmas about (least) upper bounds of chains

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

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

apply (*rule iffI*)
apply (*rule ub-rangeI*)
apply (*rule-tac y=S* (*i* + *j*) **in** *below-trans*)
apply (*erule chain-mono*)
apply (*rule le-add1*)
apply (*erule ub-rangeD*)
apply (*rule ub-rangeI*)
apply (*erule ub-rangeD*)
done

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

the lub of a constant chain is the constant

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

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

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

1.5 Finite chains

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

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

results about finite chains

lemma *max-in-chainI*: (⋀*j*. *i* ≤ *j* ⟹ *Y i* = *Y j*) ⟹ *max-in-chain i Y*
unfolding *max-in-chain-def* **by** *fast*

lemma *max-in-chainD*: ⟦*max-in-chain i Y*; *i* ≤ *j*⟧ ⟹ *Y i* = *Y j*
unfolding *max-in-chain-def* **by** *fast*

lemma *finite-chainI*:
⟦*chain C*; *max-in-chain i C*⟧ ⟹ *finite-chain C*
unfolding *finite-chain-def* **by** *fast*

lemma *finite-chainE*:
⟦*finite-chain C*; ⋀*i*. ⟦*chain C*; *max-in-chain i C*⟧ ⟹ *R*⟧ ⟹ *R*
unfolding *finite-chain-def* **by** *fast*

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

apply (*rule is-lubI*)
apply (*rule ub-rangeI*, *rename-tac j*)
apply (*rule-tac x=i* **and** *y=j* **in** *linorder-le-cases*)
apply (*drule* (*1*) *max-in-chainD*, *simp*)
apply (*erule* (*1*) *chain-mono*)
apply (*erule ub-rangeD*)
done

lemma *lub-finch2*:
finite-chain C ⟹ *range C* <<| *C* (*LEAST i. max-in-chain i C*)
apply (*erule finite-chainE*)
apply (*erule LeastI2* [**where** *Q*=λ*i. range C* <<| *C i*])
apply (*erule* (*1*) *lub-finch1*)
done

lemma *finch-imp-finite-range*: *finite-chain Y* ⟹ *finite* (*range Y*)
apply (*erule finite-chainE*)
apply (*rule-tac B=Y* ‘ {*..i*} **in** *finite-subset*)
apply (*rule subsetI*)
apply (*erule rangeE*, *rename-tac j*)
apply (*rule-tac x=i* **and** *y=j* **in** *linorder-le-cases*)
apply (*subgoal-tac Y j* = *Y i*, *simp*)
apply (*simp add*: *max-in-chain-def*)
apply *simp*
apply *simp*
done

lemma *finite-range-has-max*:
fixes *f* :: *nat* ⇒ *'a* **and** *r* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *mono*: ⋀*i j. i* ≤ *j* ⟹ *r* (*f i*) (*f j*)
assumes *finite-range*: *finite* (*range f*)
shows ∃ *k*. ∀ *i*. *r* (*f i*) (*f k*)
proof (*intro exI allI*)
fix *i* :: *nat*
let *?j* = *LEAST k. f k* = *f i*
let *?k* = *Max* ((λ*x. LEAST k. f k* = *x*) ‘ *range f*)
have *?j* ≤ *?k*
proof (*rule Max-ge*)
show *finite* ((λ*x. LEAST k. f k* = *x*) ‘ *range f*)
using *finite-range* **by** (*rule finite-imageI*)
show *?j* ∈ (λ*x. LEAST k. f k* = *x*) ‘ *range f*
by (*intro imageI rangeI*)
qed
hence *r* (*f ?j*) (*f ?k*)
by (*rule mono*)
also have *f ?j* = *f i*
by (*rule LeastI*, *rule refl*)
finally show *r* (*f i*) (*f ?k*) **.**
qed

lemma *finite-range-imp-finch*:
 ⟦*chain Y*; *finite* (*range Y*)⟧ ⟹ *finite-chain Y*
apply (*subgoal-tac* ∃ *k*. ∀ *i*. *Y i* ⊑ *Y k*)
apply (*erule exE*)
apply (*rule finite-chainI*, *assumption*)
apply (*rule max-in-chainI*)
apply (*rule below-antisym*)
apply (*erule* (*1*) *chain-mono*)
apply (*erule spec*)
apply (*rule finite-range-has-max*)
apply (*erule* (*1*) *chain-mono*)
apply *assumption*
done

lemma *bin-chain*: *x* ⊑ *y* ⟹ *chain* (λ*i*. *if i=0 then x else y*)
by (*rule chainI*, *simp*)

lemma *bin-chainmax*:
x ⊑ *y* ⟹ *max-in-chain* (*Suc 0*) (λ*i*. *if i=0 then x else y*)
unfolding *max-in-chain-def* **by** *simp*

lemma *lub-bin-chain*:
x ⊑ *y* ⟹ *range* (λ*i*::*nat*. *if i=0 then x else y*) <<| *y*
apply (*frule bin-chain*)
apply (*drule bin-chainmax*)
apply (*drule* (*1*) *lub-finch1*)
apply *simp*
done

the maximal element in a chain is its lub

lemma *lub-chain-maxelem*: ⟦*Y i* = *c*; ∀ *i*. *Y i* ⊑ *c*⟧ ⟹ *lub* (*range Y*) = *c*
by (*blast dest*: *ub-rangeD intro*: *thelubI is-lubI ub-rangeI*)

1.6 Directed sets

definition *directed* :: *'a set* ⇒ *bool* **where**
directed S ⟷ (∃ *x*. *x* ∈ *S*) ∧ (∀ *x*∈*S*. ∀ *y*∈*S*. ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*)

lemma *directedI*:
assumes *1*: ∃ *z*. *z* ∈ *S*
assumes *2*: ⋀*x y*. ⟦*x* ∈ *S*; *y* ∈ *S*⟧ ⟹ ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*
shows *directed S*
unfolding *directed-def* **using** *prems* **by** *fast*

lemma *directedD1*: *directed S* ⟹ ∃ *z*. *z* ∈ *S*
unfolding *directed-def* **by** *fast*

lemma *directedD2*: ⟦*directed S*; *x* ∈ *S*; *y* ∈ *S*⟧ ⟹ ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*

unfolding *directed-def* **by** *fast*

lemma *directedE1*:
assumes S: *directed* S
obtains z **where** $z \in S$
by (*insert directedD1* [*OF* S], *fast*)

lemma *directedE2*:
assumes S: *directed* S
assumes x: $x \in S$ **and** y: $y \in S$
obtains z **where** $z \in S$ $x \sqsubseteq z$ $y \sqsubseteq z$
by (*insert directedD2* [*OF* S x y], *fast*)

lemma *directed-finiteI*:
assumes U: $\bigwedge U.\ [\![\mathit{finite}\ U;\ U \subseteq S]\!] \Longrightarrow \exists z{\in}S.\ U <| z$
shows *directed* S
proof (*rule directedI*)
have *finite* $\{\}$ **and** $\{\} \subseteq S$ **by** *simp-all*
hence $\exists z{\in}S.\ \{\} <| z$ **by** (*rule* U)
thus $\exists z.\ z \in S$ **by** *simp*
next
fix x y
assume $x \in S$ **and** $y \in S$
hence *finite* $\{x, y\}$ **and** $\{x, y\} \subseteq S$ **by** *simp-all*
hence $\exists z{\in}S.\ \{x, y\} <| z$ **by** (*rule* U)
thus $\exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$ **by** *simp*
qed

lemma *directed-finiteD*:
assumes S: *directed* S
shows $[\![\mathit{finite}\ U;\ U \subseteq S]\!] \Longrightarrow \exists z{\in}S.\ U <| z$
proof (*induct* U *set*: *finite*)
case *empty*
from S **have** $\exists z.\ z \in S$ **by** (*rule directedD1*)
thus $\exists z{\in}S.\ \{\} <| z$ **by** *simp*
next
case (*insert* x F)
from ‹*insert* x $F \subseteq S$›
have xS: $x \in S$ **and** FS: $F \subseteq S$ **by** *simp-all*
from FS **have** $\exists y{\in}S.\ F <| y$ **by** *fact*
then obtain y **where** yS: $y \in S$ **and** Fy: $F <| y$ **..**
obtain z **where** zS: $z \in S$ **and** xz: $x \sqsubseteq z$ **and** yz: $y \sqsubseteq z$
using S xS yS **by** (*rule directedE2*)
from Fy yz **have** $F <| z$ **by** (*rule is-ub-upward*)
with xz **have** *insert* x $F <| z$ **by** *simp*
with zS **show** $\exists z{\in}S.$ *insert* x $F <| z$ **..**
qed

lemma *not-directed-empty* [*simp*]: $\neg$ *directed* $\{\}$

by (*rule notI*, *drule directedD1*, *simp*)

lemma *directed-singleton*: *directed* $\{x\}$
by (*rule directedI*, *auto*)

lemma *directed-bin*: $x \sqsubseteq y \Longrightarrow$ *directed* $\{x, y\}$
by (*rule directedI*, *auto*)

lemma *directed-chain*: *chain* $S \Longrightarrow$ *directed* (*range* S)
apply (*rule directedI*)
apply (*rule-tac* x=S *0* **in** *exI*, *simp*)
apply (*clarify*, *rename-tac* m n)
apply (*rule-tac* x=S (*max* m n) **in** *bexI*)
apply (*simp add*: *chain-mono*)
apply *simp*
done

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-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-less*)
done

end

end

2 Pcpo: Classes cpo and pcpo

theory *Pcpo*
imports *Porder*
begin

2.1 Complete partial orders

The class cpo of chain complete partial orders

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

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

lemma *cpo-lubI*: *chain S* $\Longrightarrow$ *range S* <<| $(\bigsqcup i.\ S\ i)$
  by (*fast dest*: *cpo elim*: *lubI*)

lemma *thelubE*: $[\![$*chain S*; $(\bigsqcup i.\ S\ i) = l]\!] \Longrightarrow$ *range S* <<| *l*
  by (*blast dest*: *cpo intro*: *lubI*)

Properties of the lub

lemma *is-ub-thelub*: *chain S* $\Longrightarrow S\ x \sqsubseteq (\bigsqcup i.\ S\ i)$
  by (*blast dest*: *cpo intro*: *lubI* [*THEN is-ub-lub*])

lemma *is-lub-thelub*:
  $[\![$*chain S*; *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*$]\!]$
    $\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
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 $\Longrightarrow (\bigsqcup i.\ Y\ (i + j)) = (\bigsqcup i.\ Y\ i)$
apply (*rule below-antisym*)
apply (*rule lub-range-mono*)
apply *fast*
apply *assumption*
apply (*erule chain-shift*)
apply (*rule is-lub-thelub*)
apply *assumption*

apply (*rule ub-rangeI*)
apply (*rule-tac y=Y (i + j)* **in** *below-trans*)
apply (*erule chain-mono*)
apply (*rule le-add1*)
apply (*rule is-ub-thelub*)
apply (*erule chain-shift*)
done

lemma *maxinch-is-thelub*:
chain Y $\Longrightarrow$ *max-in-chain i Y* = ((⨆ *i*. *Y i*) = *Y i*)
apply (*rule iffI*)
apply (*fast intro*!: *thelubI lub-finch1*)
apply (*unfold max-in-chain-def*)
apply (*safe intro*!: *below-antisym*)
apply (*fast elim*!: *chain-mono*)
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*; *chain Y*; ⋀*i*. *X i* $\sqsubseteq$ *Y i*⟧
$\Longrightarrow$ (⨆ *i*. *X i*) $\sqsubseteq$ (⨆ *i*. *Y i*)
apply (*erule is-lub-thelub*)
apply (*rule ub-rangeI*)
apply (*rule below-trans*)
apply (*erule meta-spec*)
apply (*erule is-ub-thelub*)
done

the = relation between two chains is preserved by their lubs

lemma *lub-equal*:
⟦*chain X*; *chain Y*; ∀*k*. *X k* = *Y k*⟧
$\Longrightarrow$ (⨆ *i*. *X i*) = (⨆ *i*. *Y i*)
by (*simp only*: *expand-fun-eq* [*symmetric*])

lemma *lub-eq*:
(⋀*i*. *X i* = *Y i*) $\Longrightarrow$ (⨆ *i*. *X i*) = (⨆ *i*. *Y i*)
by *simp*

more results about mono and = of lubs of chains

lemma *lub-mono2*:
⟦∃*j*. ∀*i*>*j*. *X i* = *Y i*; *chain X*; *chain Y*⟧
$\Longrightarrow$ (⨆ *i*. *X i*) $\sqsubseteq$ (⨆ *i*. *Y i*)
apply (*erule exE*)
apply (*subgoal-tac* (⨆ *i*. *X* (*i* + *Suc j*)) $\sqsubseteq$ (⨆ *i*. *Y* (*i* + *Suc j*)))
apply (*thin-tac* ∀*i*>*j*. *X i* = *Y i*)
apply (*simp only*: *lub-range-shift*)
apply *simp*

done

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

lemma *lub-mono3*:
  $[\![chain\ Y;\ chain\ X;\ \forall i.\ \exists j.\ Y\ i \sqsubseteq X\ j]\!]$
    $\Longrightarrow (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ X\ i)$
apply (*erule is-lub-thelub*)
apply (*rule ub-rangeI*)
apply (*erule allE*)
apply (*erule exE*)
apply (*erule below-trans*)
apply (*erule is-ub-thelub*)
done

lemma *ch2ch-lub*:
  assumes *1*: $\bigwedge j.\ chain\ (\lambda i.\ Y\ i\ j)$
  assumes *2*: $\bigwedge i.\ chain\ (\lambda j.\ Y\ i\ j)$
  shows *chain* $(\lambda i.\ \bigsqcup j.\ Y\ i\ j)$
apply (*rule chainI*)
apply (*rule lub-mono* [*OF 2 2*])
apply (*rule chainE* [*OF 1*])
done

lemma *diag-lub*:
  assumes *1*: $\bigwedge j.\ chain\ (\lambda i.\ Y\ i\ j)$
  assumes *2*: $\bigwedge i.\ chain\ (\lambda j.\ Y\ i\ j)$
  shows $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) = (\bigsqcup i.\ Y\ i\ i)$
proof (*rule below-antisym*)
  have *3*: *chain* $(\lambda i.\ Y\ i\ i)$
    apply (*rule chainI*)
    apply (*rule below-trans*)
    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 below-trans*)
    apply (*rule chain-mono* [*OF 1 le-maxI1*])
    apply (*rule chain-mono* [*OF 2 le-maxI2*])
    done

show $(\bigsqcup i.\ Y\ i\ i) \sqsubseteq (\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j)$
apply (*rule lub-mono* [*OF 3 4*])
apply (*rule is-ub-thelub* [*OF 2*])
done
qed

lemma *ex-lub*:
assumes *1*: $\bigwedge j.$ *chain* $(\lambda i.\ Y\ i\ j)$
assumes *2*: $\bigwedge i.$ *chain* $(\lambda j.\ Y\ i\ j)$
shows $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) = (\bigsqcup j.\ \bigsqcup i.\ Y\ i\ j)$
by (*simp add*: *diag-lub 1 2*)

end

2.2 Pointed cpos

The class pcpo of pointed cpos

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

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

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

derive the old rule minimal

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

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

end

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

setup ⟪
Reorient-Proc.add
(*fn Const*(@{*const-name UU*}, -) => *true* | - => *false*)
⟫

simproc-setup *reorient-bottom* $(\bot = x)$ = *Reorient-Proc.proc*

context *pcpo*
begin

useful lemmas about $\bot$

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

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

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

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

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

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

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

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

end

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

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

subclass *cpo*
apply *default*
apply (*frule chfin*)
apply (*blast intro*: *lub-finch1*)

done

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

end

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

subclass *chfin*
apply *default*
apply (*drule finite-range-imp-finch*)
apply (*rule finite*)
apply (*simp add*: *finite-chain-def*)
done

end

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

subclass *chfin*
apply *default*
apply (*unfold max-in-chain-def*)
apply (*case-tac* $\forall i.\ Y\ i = \bot$)
apply *simp*
apply *simp*
apply (*erule exE*)
apply (*rule-tac* $x{=}i$ **in** *exI*)
apply *clarify*
apply (*blast dest*: *chain-mono ax-flat*)
done

lemma *flat-below-iff*:
shows $(x \sqsubseteq y) = (x = \bot \vee x = y)$
by (*safe dest*!: *ax-flat*)

lemma *flat-eq*: $a \neq \bot \Longrightarrow a \sqsubseteq b = (a = b)$
by (*safe dest*!: *ax-flat*)

end

Discrete cpos

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

subclass *po*
proof qed *simp-all*

In a discrete cpo, every chain is constant

lemma *discrete-chain-const*:
  assumes S: *chain S*
  shows $\exists x.\ S = (\lambda i.\ x)$
proof (*intro exI ext*)
  fix i :: *nat*
  have $S\ 0 \sqsubseteq S\ i$ **using** S *le0* **by** (*rule chain-mono*)
  hence $S\ 0 = S\ i$ **by** *simp*
  thus $S\ i = S\ 0$ **by** (*rule sym*)
qed

subclass *cpo*
proof
  fix S :: *nat* $\Rightarrow$ $'a$
  assume S: *chain S*
  hence $\exists x.\ S = (\lambda i.\ x)$
    by (*rule discrete-chain-const*)
  thus $\exists x.$ *range* S <<| x
    by (*fast intro*: *lub-const*)
qed

end

end

3 Cont: Continuity and monotonicity

theory *Cont*
imports *Pcpo*
begin

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

default-sort *po*

3.1 Definitions

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

definition

cont :: ('*a*::*cpo* ⇒ '*b*::*cpo*) ⇒ *bool*
where
cont f = (∀ *Y*. *chain Y* ⟶ *range* (λ*i*. *f* (*Y i*)) <<| *f* (⨆ *i*. *Y i*))

lemma *contI*:
⟦⋀*Y*. *chain Y* ⟹ *range* (λ*i*. *f* (*Y i*)) <<| *f* (⨆ *i*. *Y i*)⟧ ⟹ *cont f*
by (*simp add*: *cont-def*)

lemma *contE*:
⟦*cont f*; *chain Y*⟧ ⟹ *range* (λ*i*. *f* (*Y i*)) <<| *f* (⨆ *i*. *Y i*)
by (*simp add*: *cont-def*)

lemma *monofunI*:
⟦⋀*x y*. *x* ⊑ *y* ⟹ *f x* ⊑ *f y*⟧ ⟹ *monofun f*
by (*simp add*: *monofun-def*)

lemma *monofunE*:
⟦*monofun f*; *x* ⊑ *y*⟧ ⟹ *f x* ⊑ *f y*
by (*simp add*: *monofun-def*)

3.2 Equivalence of alternate definition

monotone functions map chains to chains

lemma *ch2ch-monofun*: ⟦*monofun f*; *chain Y*⟧ ⟹ *chain* (λ*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*⟧ ⟹ *range* (λ*i*. *f* (*Y i*)) <| *f u*
apply (*rule ub-rangeI*)
apply (*erule monofunE*)
apply (*erule ub-rangeD*)
done

a lemma about binary chains

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

continuity implies monotonicity

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

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

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

continuity implies preservation of lubs

lemma *cont2contlubE*:
  $[\![cont\ f;\ chain\ Y]\!] \Longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = (\bigsqcup\ i.\ f\ (Y\ i))$
apply (*rule thelubI* [*symmetric*])
apply (*erule* (*1*) *contE*)
done

lemma *contI2*:
  assumes *mono*: *monofun f*
  assumes *below*: $\bigwedge Y.\ [\![chain\ Y;\ chain\ (\lambda i.\ f\ (Y\ i))]\!]$
    $\Longrightarrow f\ (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ f\ (Y\ i))$
  shows *cont f*
apply (*rule contI*)
apply (*rule thelubE*)
apply (*erule ch2ch-monofun* [*OF mono*])
apply (*rule below-antisym*)
apply (*rule is-lub-thelub*)
apply (*erule ch2ch-monofun* [*OF mono*])
apply (*rule ub2ub-monofun* [*OF mono*])
apply (*rule is-lubD1*)
apply (*erule cpo-lubI*)
apply (*rule below*, *assumption*)
apply (*erule ch2ch-monofun* [*OF mono*])
done

3.3 Collection of continuity rules

ML ⟪
structure Cont2ContData = Named-Thms
(
  val name = cont2cont
  val description = continuity intro rule
)
⟫

setup *Cont2ContData.setup*

3.4 Continuity of basic functions

The identity function is continuous

lemma *cont-id* [*simp*, *cont2cont*]: *cont* ($\lambda x.\ x$)
apply (*rule contI*)
apply (*erule cpo-lubI*)
done

constant functions are continuous

lemma *cont-const* [*simp*, *cont2cont*]: *cont* ($\lambda x.\ c$)
apply (*rule contI*)
apply (*rule lub-const*)
done

application of functions is continuous

lemma *cont-apply*:
  fixes $f :: 'a{::}cpo \Rightarrow 'b{::}cpo \Rightarrow 'c{::}cpo$ **and** $t :: 'a \Rightarrow 'b$
  assumes *1*: *cont* ($\lambda x.\ t\ x$)
  assumes *2*: $\bigwedge x.$ *cont* ($\lambda y.\ f\ x\ y$)
  assumes *3*: $\bigwedge y.$ *cont* ($\lambda x.\ f\ x\ y$)
  shows *cont* ($\lambda x.\ (f\ x)\ (t\ x)$)
proof (*rule contI2* [*OF monofunI*])
  fix $x\ y :: 'a$ **assume** $x \sqsubseteq y$
  then show $f\ x\ (t\ x) \sqsubseteq f\ y\ (t\ y)$
    by (*auto intro*: *cont2monofunE* [*OF 1*]
                cont2monofunE [*OF 2*]
                cont2monofunE [*OF 3*]
                below-trans)
next
  fix $Y :: nat \Rightarrow 'a$ **assume** *chain Y*
  then show $f\ (\bigsqcup i.\ Y\ i)\ (t\ (\bigsqcup i.\ Y\ i)) \sqsubseteq (\bigsqcup i.\ f\ (Y\ i)\ (t\ (Y\ i)))$
    by (*simp only*: *cont2contlubE* [*OF 1*] *ch2ch-cont* [*OF 1*]
                cont2contlubE [*OF 2*] *ch2ch-cont* [*OF 2*]
                cont2contlubE [*OF 3*] *ch2ch-cont* [*OF 3*]
                diag-lub below-refl)
qed

lemma *cont-compose*:
  $[\![$*cont c*; *cont* ($\lambda x.\ f\ x$)$]\!] \Longrightarrow$ *cont* ($\lambda x.\ c\ (f\ x)$)
by (*rule cont-apply* [*OF - - cont-const*])

if-then-else is continuous

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

3.5 Finite chains and flat pcpos

monotone functions map finite chains to finite chains

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

The same holds for continuous functions

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

lemma *chfindom-monofun2cont*: *monofun f* ⟹ *cont* (*f*::'*a*::*chfin* ⇒ '*b*::*cpo*)
apply (*erule contI2*)
apply (*frule chfin2finch*)
apply (*clarsimp simp add*: *finite-chain-def*)
apply (*subgoal-tac max-in-chain i* (λ*i*. *f* (*Y i*)))
apply (*simp add*: *maxinch-is-thelub ch2ch-monofun*)
apply (*force simp add*: *max-in-chain-def*)
done

some properties of flat

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

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

functions with discrete domain

lemma *cont-discrete-cpo* [*simp*, *cont2cont*]: *cont* (*f*::'*a*::*discrete-cpo* ⇒ '*b*::*cpo*)
apply (*rule contI*)
apply (*drule discrete-chain-const*, *clarify*)
apply (*simp add*: *lub-const*)
done

end

4 Discrete: Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

4.1 Discrete ordering

instantiation *discr* :: (*type*) *below*
begin

definition
below-discr-def:
(*op* $\sqsubseteq$ :: *'a discr* $\Rightarrow$ *'a discr* $\Rightarrow$ *bool*) = (*op* =)

instance ..
end

4.2 Discrete cpo class instance

instance *discr* :: (*type*) *discrete-cpo*
by *intro-classes* (*simp add*: *below-discr-def*)

lemma *discr-below-eq* [*iff*]: ((*x*::('*a*::*type*)*discr*) << *y*) = (*x* = *y*)
by *simp*

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*)

instance *discr* :: (*finite*) *finite-po*
proof
have *finite* (*Discr* ' (*UNIV* :: *'a set*))
by (*rule finite-imageI* [*OF finite*])
also have (*Discr* ' (*UNIV* :: *'a set*)) = *UNIV*
by (*auto*, *case-tac x*, *auto*)
finally show *finite* (*UNIV* :: *'a discr set*) .
qed

instance *discr* :: (*type*) *chfin*
apply *intro-classes*
apply (*rule-tac x=0* **in** *exI*)
apply (*unfold max-in-chain-def*)
apply (*clarify*, *erule discr-chain0* [*symmetric*])
done

4.3 *undiscr*

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

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

lemma *Discr-undiscr* [*simp*]: *Discr* (*undiscr y*) = *y*
by (*induct y*) *simp*

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*)
by (*rule cont-discrete-cpo*)

end

5 Adm: Admissibility and compactness

theory *Adm*
imports *Cont*
begin

default-sort *cpo*

5.1 Definitions

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

lemma *admI*:
(⋀*Y*. ⟦*chain Y*; ∀ *i*. *P* (*Y i*)⟧ ⟹ *P* (⨆ *i*. *Y i*)) ⟹ *adm P*
unfolding *adm-def* **by** *fast*

lemma *admD*: ⟦*adm P*; *chain Y*; ⋀*i*. *P* (*Y i*)⟧ ⟹ *P* (⨆ *i*. *Y i*)
unfolding *adm-def* **by** *fast*

lemma *admD2*: ⟦*adm* (λ*x*. ¬ *P x*); *chain Y*; *P* (⨆ *i*. *Y i*)⟧ ⟹ ∃ *i*. *P* (*Y i*)
unfolding *adm-def* **by** *fast*

lemma *triv-admI*: ∀ *x*. *P x* ⟹ *adm P*
by (*rule admI*, *erule spec*)

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-chfin*: $adm\ (P{::}'a{::}chfin \Rightarrow bool)$
by (*rule admI*, *frule chfin*, *auto simp add*: *maxinch-is-thelub*)

5.3 Admissibility of special formulae and propagation

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

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

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

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

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

lemma *adm-disj-lemma1*:
$[\![chain\ (Y{::}nat \Rightarrow {'a}{::}cpo);\ \forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!]$
$\Longrightarrow chain\ (\lambda i.\ Y\ (LEAST\ j.\ i \leq j \wedge P\ (Y\ j)))$
apply (*rule chainI*)
apply (*erule chain-mono*)
apply (*rule Least-le*)
apply (*rule LeastI2-ex*)
apply *simp-all*
done

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

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

apply (*rule lub-range-mono*, *fast*, *assumption+*)
done

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

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

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

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

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

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

admissibility and continuity

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

lemma *adm-eq*: ⟦*cont u*; *cont v*⟧ ⟹ *adm* (λ*x*. *u x* = *v x*)

by (*simp add*: *po-eq-conv adm-conj adm-below*)

lemma *adm-subst*: ⟦*cont t*; *adm P*⟧ ⟹ *adm* (λ*x*. *P* (*t x*))
apply (*rule admI*)
apply (*simp add*: *cont2contlubE*)
apply (*erule admD*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule spec*)
done

lemma *adm-not-below*: *cont t* ⟹ *adm* (λ*x*. ¬ *t x* ⊑ *u*)
apply (*rule admI*)
apply (*drule-tac x=0* **in** *spec*)
apply (*erule contrapos-nn*)
apply (*erule rev-below-trans*)
apply (*erule cont2mono* [*THEN monofunE*])
apply (*erule is-ub-thelub*)
done

5.4 Compactness

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

lemma *compactI*: *adm* (λ*x*. ¬ *k* ⊑ *x*) ⟹ *compact k*
unfolding *compact-def* **.**

lemma *compactD*: *compact k* ⟹ *adm* (λ*x*. ¬ *k* ⊑ *x*)
unfolding *compact-def* **.**

lemma *compactI2*:
(⋀*Y*. ⟦*chain Y*; *x* ⊑ (⨆ *i*. *Y i*)⟧ ⟹ ∃ *i*. *x* ⊑ *Y i*) ⟹ *compact x*
unfolding *compact-def adm-def* **by** *fast*

lemma *compactD2*:
⟦*compact x*; *chain Y*; *x* ⊑ (⨆ *i*. *Y i*)⟧ ⟹ ∃ *i*. *x* ⊑ *Y i*
unfolding *compact-def adm-def* **by** *fast*

lemma *compact-chfin* [*simp*]: *compact* (*x*::'*a*::*chfin*)
by (*rule compactI* [*OF adm-chfin*])

lemma *compact-imp-max-in-chain*:
⟦*chain Y*; *compact* (⨆ *i*. *Y i*)⟧ ⟹ ∃ *i*. *max-in-chain i Y*
apply (*drule* (*1*) *compactD2*, *simp*)
apply (*erule exE*, *rule-tac x=i* **in** *exI*)
apply (*rule max-in-chainI*)
apply (*rule below-antisym*)
apply (*erule* (*1*) *chain-mono*)

apply (*erule* (*1*) *below-trans* [*OF is-ub-thelub*])
done

admissibility and compactness

lemma *adm-compact-not-below*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. ¬ *k* ⊑ *t x*)
unfolding *compact-def* **by** (*rule adm-subst*)

lemma *adm-neq-compact*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. *t x* ≠ *k*)
by (*simp add*: *po-eq-conv adm-imp adm-not-below adm-compact-not-below*)

lemma *adm-compact-neq*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. *k* ≠ *t x*)
by (*simp add*: *po-eq-conv adm-imp adm-not-below adm-compact-not-below*)

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

lemma *adm-not-UU*: *cont t* ⟹ *adm* (λ*x*. *t x* ≠ ⊥)
by (*simp add*: *adm-neq-compact*)

Any upward-closed predicate is admissible.

lemma *adm-upward*:
assumes *P*: ⋀*x y*. ⟦*P x*; *x* ⊑ *y*⟧ ⟹ *P y*
shows *adm P*
by (*rule admI*, *drule spec*, *erule P*, *erule is-ub-thelub*)

lemmas *adm-lemmas* [*simp*] =
adm-not-free adm-conj adm-all adm-ball adm-disj adm-imp adm-iff
adm-below adm-eq adm-not-below
adm-compact-not-below adm-compact-neq adm-neq-compact adm-not-UU

end

6 Pcpodef: Subtypes of pcpos

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

6.1 Proving a subtype is a partial order

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

setup ⟪ *Sign.add-const-constraint* (@{*const-name Porder.below*}, *NONE*) ⟫

theorem *typedef-po*:
fixes *Abs* :: *'a*::*po* ⇒ *'b*::*type*

assumes *type*: *type-definition Rep Abs A*
and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows *OFCLASS*('*b*, *po-class*)
apply (*intro-classes*, *unfold below*)
apply (*rule below-refl*)
apply (*erule* (*1*) *below-trans*)
apply (*rule type-definition.Rep-inject* [*OF type*, *THEN iffD1*])
apply (*erule* (*1*) *below-antisym*)
done

setup ⟪ *Sign.add-const-constraint* (@{*const-name Porder.below*},
SOME @{*typ* '*a*::*below* ⇒ '*a*::*below* ⇒ *bool*}) ⟫

6.2 Proving a subtype is finite

lemma *typedef-finite-UNIV*:
fixes *Abs* :: '*a*::*type* ⇒ '*b*::*type*
assumes *type*: *type-definition Rep Abs A*
shows *finite A* ⟹ *finite* (*UNIV* :: '*b set*)
proof −
assume *finite A*
hence *finite* (*Abs* ‘ *A*) **by** (*rule finite-imageI*)
thus *finite* (*UNIV* :: '*b set*)
by (*simp only*: *type-definition.Abs-image* [*OF type*])
qed

theorem *typedef-finite-po*:
fixes *Abs* :: '*a*::*finite-po* ⇒ '*b*::*po*
assumes *type*: *type-definition Rep Abs A*
shows *OFCLASS*('*b*, *finite-po-class*)
apply (*intro-classes*)
apply (*rule typedef-finite-UNIV* [*OF type*])
apply (*rule finite*)
done

6.3 Proving a subtype is chain-finite

lemma *monofun-Rep*:
assumes *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows *monofun Rep*
by (*rule monofunI*, *unfold below*)

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

theorem *typedef-chfin*:
fixes *Abs* :: '*a*::*chfin* ⇒ '*b*::*po*
assumes *type*: *type-definition Rep Abs A*
and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows *OFCLASS*('*b*, *chfin-class*)

apply *intro-classes*
apply (*drule ch2ch-Rep* [*OF below*])
apply (*drule chfin*)
apply (*unfold max-in-chain-def*)
apply (*simp add*: *type-definition.Rep-inject* [*OF type*])
done

6.4 Proving a subtype is complete

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

lemma *Abs-inverse-lub-Rep*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *chain S* ⟹ *Rep* (*Abs* (⨆ *i*. *Rep* (*S i*))) = (⨆ *i*. *Rep* (*S i*))
apply (*rule type-definition.Abs-inverse* [*OF type*])
apply (*erule admD* [*OF adm ch2ch-Rep* [*OF below*]])
apply (*rule type-definition.Rep* [*OF type*])
done

theorem *typedef-lub*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *chain S* ⟹ *range S* <<| *Abs* (⨆ *i*. *Rep* (*S i*))
apply (*frule ch2ch-Rep* [*OF below*])
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*simp only*: *below Abs-inverse-lub-Rep* [*OF type below adm*])
apply (*erule is-ub-thelub*)
apply (*simp only*: *below Abs-inverse-lub-Rep* [*OF type below adm*])
apply (*erule is-lub-thelub*)
apply (*erule ub2ub-Rep* [*OF below*])
done

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

theorem *typedef-cpo*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *OFCLASS*(*'b*, *cpo-class*)
proof
fix *S*::*nat* ⇒ *'b* **assume** *chain S*

hence *range S <<| Abs* ($\bigsqcup$ *i. Rep* (*S i*))
by (*rule typedef-lub* [*OF type below adm*])
thus $\exists$ *x. range S <<| x* **..**
qed

6.4.1 Continuity of *Rep* and *Abs*

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

theorem *typedef-cont-Rep*:
fixes *Abs* :: *'a*::*cpo* $\Rightarrow$ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* $\sqsubseteq$ $\equiv$ λ*x y. Rep x* $\sqsubseteq$ *Rep y*
and *adm*: *adm* (λ*x. x* $\in$ *A*)
shows *cont Rep*
apply (*rule contI*)
apply (*simp only*: *typedef-thelub* [*OF type below adm*])
apply (*simp only*: *Abs-inverse-lub-Rep* [*OF type below adm*])
apply (*rule cpo-lubI*)
apply (*erule ch2ch-Rep* [*OF below*])
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 *below*: *op* $\sqsubseteq$ $\equiv$ λ*x y. Rep x* $\sqsubseteq$ *Rep y*
shows *range* (λ*i. Rep* (*S i*)) *<<| Rep x* $\Longrightarrow$ *range S <<| x*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*subst below*)
apply (*erule is-ub-lub*)
apply (*subst below*)
apply (*erule is-lub-lub*)
apply (*erule ub2ub-Rep* [*OF below*])
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 *below*: *op* $\sqsubseteq$ $\equiv$ λ*x y. Rep x* $\sqsubseteq$ *Rep y*
and *adm*: *adm* (λ*x. x* $\in$ *A*)
and *f-in-A*: $\bigwedge$*x. f x* $\in$ *A*
and *cont-f*: *cont f*
shows *cont* (λ*x. Abs* (*f x*))
apply (*rule contI*)
apply (*rule typedef-is-lubI* [*OF below*])
apply (*simp only*: *type-definition.Abs-inverse* [*OF type f-in-A*])

apply (*erule cont-f* [*THEN contE*])
done

6.5 Proving subtype elements are compact

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

6.6 Proving a subtype is pointed

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

theorem *typedef-pcpo-generic*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *z-in-A*: *z* ∈ *A*
and *z-least*: ⋀*x*. *x* ∈ *A* ⟹ *z* ⊑ *x*
shows *OFCLASS*(*'b*, *pcpo-class*)
apply (*intro-classes*)
apply (*rule-tac x*=*Abs z* **in** *exI*, *rule allI*)
apply (*unfold below*)
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 ⊥.

theorem *typedef-pcpo*:
fixes *Abs* :: *'a*::*pcpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *OFCLASS*(*'b*, *pcpo-class*)
by (*rule typedef-pcpo-generic* [*OF type below UU-in-A*], *rule minimal*)

6.6.1 Strictness of *Rep* and *Abs*

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

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

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

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

theorem *typedef-Abs-defined*:
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *UU-in-A*: $\bot \in A$
  shows $[\![x \neq \bot;\ x \in A]\!] \Longrightarrow Abs\ x \neq \bot$
by (*simp add*: *typedef-Abs-strict-iff* [*OF type below UU-in-A*])

theorem *typedef-Rep-defined*:
  assumes *type*: *type-definition Rep Abs A*

and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $x \neq \bot \Longrightarrow Rep\ x \neq \bot$
by (*simp add*: *typedef-Rep-strict-iff* [*OF type below UU-in-A*])

6.7 Proving a subtype is flat

theorem *typedef-flat*:
fixes *Abs* :: $'a$::*flat* $\Rightarrow$ $'b$::*pcpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows *OFCLASS*($'b$, *flat-class*)
apply (*intro-classes*)
apply (*unfold below*)
apply (*simp add*: *type-definition.Rep-inject* [*OF type*, *symmetric*])
apply (*simp add*: *typedef-Rep-strict* [*OF type below UU-in-A*])
apply (*simp add*: *ax-flat*)
done

6.8 HOLCF type definition package

use *Tools/pcpodef.ML*

end

7 Ffun: Class instances for the full function space

theory *Ffun*
imports *Cont*
begin

7.1 Full function space is a partial order

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

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

instance ..
end

instance *fun* :: (*type*, *po*) *po*
proof
fix f :: $'a \Rightarrow 'b$
show $f \sqsubseteq f$
by (*simp add*: *below-fun-def*)

next
fix $f\ g :: 'a \Rightarrow 'b$
assume $f \sqsubseteq g$ **and** $g \sqsubseteq f$ **thus** $f = g$
by (*simp add*: *below-fun-def expand-fun-eq below-antisym*)
next
fix $f\ g\ h :: 'a \Rightarrow 'b$
assume $f \sqsubseteq g$ **and** $g \sqsubseteq h$ **thus** $f \sqsubseteq h$
unfolding *below-fun-def* **by** (*fast elim*: *below-trans*)
qed

make the symbol << accessible for type fun

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

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

7.2 Full function space is chain complete

function application is monotone

lemma *monofun-app*: *monofun* $(\lambda f.\ f\ x)$
by (*rule monofunI*, *simp add*: *below-fun-def*)

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 below-fun-def*)

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

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

lemma *ub2ub-fun*:
range $S <|\ u \Longrightarrow$ *range* $(\lambda i.\ S\ i\ x) <|\ u\ x$
by (*auto simp add*: *is-ub-def below-fun-def*)

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

lemma *is-lub-lambda*:
assumes f: $\bigwedge x.$ *range* $(\lambda i.\ Y\ i\ x) <<|\ f\ x$
shows *range* $Y <<|\ f$
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*rule below-fun-ext*)
apply (*rule is-ub-lub* [*OF f*])
apply (*rule below-fun-ext*)
apply (*rule is-lub-lub* [*OF f*])
apply (*erule ub2ub-fun*)
done

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

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

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

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

instance *fun* :: (*finite*, *finite-po*) *finite-po* **..**

instance *fun* :: (*type*, *discrete-cpo*) *discrete-cpo*
proof
  fix *f g* :: '*a* ⇒ '*b*
  show *f* ⊑ *g* ⟷ *f* = *g*
    unfolding *expand-fun-below expand-fun-eq*
    by *simp*
qed

chain-finite function spaces

lemma *maxinch2maxinch-lambda*:
  (⋀*x*. *max-in-chain n* (λ*i*. *S i x*)) ⟹ *max-in-chain n S*
unfolding *max-in-chain-def expand-fun-eq* **by** *simp*

lemma *maxinch-mono*:
  ⟦*max-in-chain i Y*; *i* ≤ *j*⟧ ⟹ *max-in-chain j Y*
unfolding *max-in-chain-def*
proof (*intro allI impI*)
  fix *k*
  assume *Y*: ∀ *n*≥*i*. *Y i* = *Y n*
  assume *ij*: *i* ≤ *j*
  assume *jk*: *j* ≤ *k*
  from *ij jk* **have** *ik*: *i* ≤ *k* **by** *simp*
  from *Y ij* **have** *Yij*: *Y i* = *Y j* **by** *simp*
  from *Y ik* **have** *Yik*: *Y i* = *Y k* **by** *simp*
  from *Yij Yik* **show** *Y j* = *Y k* **by** *auto*
qed

instance *fun* :: (*finite*, *chfin*) *chfin*
proof
fix *Y* :: *nat* ⇒ *'a* ⇒ *'b*
let *?n* = λ*x*. *LEAST n*. *max-in-chain n* (λ*i*. *Y i x*)
assume *chain Y*
hence ⋀*x*. *chain* (λ*i*. *Y i x*)
by (*rule ch2ch-fun*)
hence ⋀*x*. ∃ *n*. *max-in-chain n* (λ*i*. *Y i x*)
by (*rule chfin*)
hence ⋀*x*. *max-in-chain* (*?n x*) (λ*i*. *Y i x*)
by (*rule LeastI-ex*)
hence ⋀*x*. *max-in-chain* (*Max* (*range ?n*)) (λ*i*. *Y i x*)
by (*rule maxinch-mono* [*OF - Max-ge*], *simp-all*)
hence *max-in-chain* (*Max* (*range ?n*)) *Y*
by (*rule maxinch2maxinch-lambda*)
thus ∃ *n*. *max-in-chain n Y* **..**
qed

7.3 Full function space is pointed

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

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

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

for compatibility with old HOLCF-Version

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

function application is strict in the left argument

lemma *app-strict* [*simp*]: ⊥ *x* = ⊥
by (*simp add*: *inst-fun-pcpo*)

The following results are about application for functions in *'a* ⇒ *'b*

lemma *monofun-fun-fun*: *f* ⊑ *g* ⟹ *f x* ⊑ *g x*
by (*simp add*: *below-fun-def*)

lemma *monofun-fun-arg*: ⟦*monofun f*; *x* ⊑ *y*⟧ ⟹ *f x* ⊑ *f y*
by (*rule monofunE*)

lemma *monofun-fun*: ⟦*monofun f*; *monofun g*; *f* ⊑ *g*; *x* ⊑ *y*⟧ ⟹ *f x* ⊑ *g y*
by (*rule below-trans* [*OF monofun-fun-arg monofun-fun-fun*])

7.4 Propagation of monotonicity and continuity

the lub of a chain of monotone functions is monotone

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

the lub of a chain of continuous functions is continuous

lemma *cont-lub-fun*:
  ⟦*chain F*; ∀ *i*. *cont* (*F i*)⟧ ⟹ *cont* (⨆ *i*. *F i*)
apply (*rule contI2*)
apply (*erule monofun-lub-fun*)
apply (*simp add*: *cont2mono*)
apply (*simp add*: *thelub-fun cont2contlubE*)
apply (*simp add*: *diag-lub ch2ch-fun ch2ch-cont*)
done

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

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

lemma *cont2cont-fun*: *cont f* ⟹ *cont* (λ*x*. *f x y*)
apply (*rule contI2*)
apply (*erule cont2mono* [*THEN mono2mono-fun*])
apply (*simp add*: *cont2contlubE*)
apply (*simp add*: *thelub-fun ch2ch-cont*)
done

Note (λ*x*. λ*y*. *f x y*) = *f*

lemma *mono2mono-lambda*:
  assumes *f*: ⋀*y*. *monofun* (λ*x*. *f x y*) **shows** *monofun f*
apply (*rule monofunI*)
apply (*rule below-fun-ext*)
apply (*erule monofunE* [*OF f*])
done

lemma *cont2cont-lambda* [*simp*]:

assumes f: $\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)$ **shows** $cont\ f$
apply (*rule contI2*)
apply (*simp add*: *mono2mono-lambda cont2mono f*)
apply (*rule below-fun-ext*)
apply (*simp add*: *thelub-fun cont2contlubE* [*OF f*])
done

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

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

lemma *contlub-abstraction*:
$[\![chain\ Y;\ \forall y.\ cont\ (\lambda x.(c{::}'a{::}cpo \Rightarrow 'b{::}type \Rightarrow 'c{::}cpo)\ x\ y)]\!] \Longrightarrow (\lambda y.\ \bigsqcup i.\ c\ (Y\ i)\ y) = (\bigsqcup i.\ (\lambda y.\ c\ (Y\ i)\ y))$
apply (*rule thelub-fun* [*symmetric*])
apply (*simp add*: *ch2ch-cont*)
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 *cont2cont-app*:
$[\![cont\ f;\ \forall x.\ cont\ (f\ x);\ cont\ t]\!] \Longrightarrow cont\ (\lambda x.\ (f\ x)\ (t\ x))$
apply (*erule cont-apply* [**where** $t{=}t$])
apply (*erule spec*)
apply (*erule cont2cont-fun*)
done

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

lemma *cont2cont-app3*: $[\![cont\ f;\ cont\ t]\!] \Longrightarrow cont\ (\lambda x.\ f\ (t\ x))$
by (*rule cont2cont-app2* [*OF cont-const*])

end

8 Product-Cpo: The cpo of cartesian products

theory *Product-Cpo*
imports *Adm*
begin

default-sort *cpo*

8.1 Unit type is a pcpo

instantiation *unit* :: *below*
begin

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

instance ..
end

instance *unit* :: *discrete-cpo*
by *intro-classes simp*

instance *unit* :: *finite-po* **..**

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

8.2 Product type is a partial order

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

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

instance ..
end

instance $*$ :: (*po*, *po*) *po*
proof
fix $x :: 'a \times 'b$
show $x \sqsubseteq x$
unfolding *below-prod-def* **by** *simp*
next
fix $x\ y :: 'a \times 'b$
assume $x \sqsubseteq y\ y \sqsubseteq x$ **thus** $x = y$
unfolding *below-prod-def Pair-fst-snd-eq*
by (*fast intro*: *below-antisym*)
next
fix $x\ y\ z :: 'a \times 'b$
assume $x \sqsubseteq y\ y \sqsubseteq z$ **thus** $x \sqsubseteq z$
unfolding *below-prod-def*
by (*fast intro*: *below-trans*)
qed

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

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

unfolding *below-prod-def* **by** *simp*

lemma *Pair-below-iff* [*simp*]: $(a, b) \sqsubseteq (c, d) \longleftrightarrow a \sqsubseteq c \wedge b \sqsubseteq d$
unfolding *below-prod-def* **by** *simp*

Pair (-,-) is monotone in both arguments

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

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

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

lemma *ch2ch-Pair* [*simp*]:
chain $X \Longrightarrow$ *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ (X\ i,\ Y\ i))$
by (*rule chainI*, *simp add*: *chainE*)

fst and *snd* are monotone

lemma *fst-monofun*: $x \sqsubseteq y \Longrightarrow fst\ x \sqsubseteq fst\ y$
unfolding *below-prod-def* **by** *simp*

lemma *snd-monofun*: $x \sqsubseteq y \Longrightarrow snd\ x \sqsubseteq snd\ y$
unfolding *below-prod-def* **by** *simp*

lemma *monofun-fst*: *monofun fst*
by (*simp add*: *monofun-def below-prod-def*)

lemma *monofun-snd*: *monofun snd*
by (*simp add*: *monofun-def below-prod-def*)

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

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

lemma *prod-chain-cases*:
assumes *chain Y*
obtains *A B*
where *chain A* **and** *chain B* **and** $Y = (\lambda i.\ (A\ i,\ B\ i))$
proof
from ‹*chain Y*› **show** *chain* $(\lambda i.\ fst\ (Y\ i))$ **by** (*rule ch2ch-fst*)
from ‹*chain Y*› **show** *chain* $(\lambda i.\ snd\ (Y\ i))$ **by** (*rule ch2ch-snd*)
show $Y = (\lambda i.\ (fst\ (Y\ i),\ snd\ (Y\ i)))$ **by** *simp*
qed

8.4 Product type is a cpo

lemma *is-lub-Pair*:
⟦*range A <<| x*; *range B <<| y*⟧ ⟹ *range* (λ*i*. (*A i*, *B i*)) <<| (*x*, *y*)
apply (*rule is-lubI* [*OF ub-rangeI*])
apply (*simp add*: *is-ub-lub*)
apply (*frule ub2ub-monofun* [*OF monofun-fst*])
apply (*drule ub2ub-monofun* [*OF monofun-snd*])
apply (*simp add*: *below-prod-def is-lub-lub*)
done

lemma *thelub-Pair*:
⟦*chain* (*A*::*nat* ⇒ *'a*::*cpo*); *chain* (*B*::*nat* ⇒ *'b*::*cpo*)⟧
⟹ (⨆ *i*. (*A i*, *B i*)) = (⨆ *i*. *A i*, ⨆ *i*. *B i*)
by (*fast intro*: *thelubI is-lub-Pair elim*: *thelubE*)

lemma *lub-cprod*:
fixes *S* :: *nat* ⇒ (*'a*::*cpo* × *'b*::*cpo*)
assumes *S*: *chain S*
shows *range S <<|* (⨆ *i*. *fst* (*S i*), ⨆ *i*. *snd* (*S i*))
proof −
from ‹*chain S*› **have** *chain* (λ*i*. *fst* (*S i*))
by (*rule ch2ch-fst*)
hence *1*: *range* (λ*i*. *fst* (*S i*)) <<| (⨆ *i*. *fst* (*S i*))
by (*rule cpo-lubI*)
from ‹*chain S*› **have** *chain* (λ*i*. *snd* (*S i*))
by (*rule ch2ch-snd*)
hence *2*: *range* (λ*i*. *snd* (*S i*)) <<| (⨆ *i*. *snd* (*S i*))
by (*rule cpo-lubI*)
show *range S <<|* (⨆ *i*. *fst* (*S i*), ⨆ *i*. *snd* (*S i*))
using *is-lub-Pair* [*OF 1 2*] **by** *simp*
qed

lemma *thelub-cprod*:
chain (*S*::*nat* ⇒ *'a*::*cpo* × *'b*::*cpo*)
⟹ (⨆ *i*. *S i*) = (⨆ *i*. *fst* (*S i*), ⨆ *i*. *snd* (*S i*))
by (*rule lub-cprod* [*THEN thelubI*])

instance ∗ :: (*cpo*, *cpo*) *cpo*
proof
fix *S* :: *nat* ⇒ (*'a* × *'b*)
assume *chain S*
hence *range S <<|* (⨆ *i*. *fst* (*S i*), ⨆ *i*. *snd* (*S i*))
by (*rule lub-cprod*)
thus ∃ *x*. *range S <<| x* **..**
qed

instance ∗ :: (*finite-po*, *finite-po*) *finite-po* **..**

instance ∗ :: (*discrete-cpo*, *discrete-cpo*) *discrete-cpo*

proof
  fix $x\ y :: 'a \times 'b$
  show $x \sqsubseteq y \longleftrightarrow x = y$
    unfolding *below-prod-def Pair-fst-snd-eq*
    by *simp*
qed

8.5 Product type is pointed

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

instance $* :: (pcpo, pcpo)\ pcpo$
by *intro-classes* (*fast intro*: *minimal-cprod*)

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

lemma *Pair-defined-iff* [*simp*]: $(x, y) = \bot \longleftrightarrow x = \bot \wedge y = \bot$
unfolding *inst-cprod-pcpo* **by** *simp*

lemma *fst-strict* [*simp*]: $fst\ \bot = \bot$
unfolding *inst-cprod-pcpo* **by** (*rule fst-conv*)

lemma *snd-strict* [*simp*]: $snd\ \bot = \bot$
unfolding *inst-cprod-pcpo* **by** (*rule snd-conv*)

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

lemma *split-strict* [*simp*]: $split\ f\ \bot = f\ \bot\ \bot$
unfolding *split-def* **by** *simp*

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

lemma *cont-pair1*: $cont\ (\lambda x.\ (x, y))$
apply (*rule contI*)
apply (*rule is-lub-Pair*)
apply (*erule cpo-lubI*)
apply (*rule lub-const*)
done

lemma *cont-pair2*: $cont\ (\lambda y.\ (x, y))$
apply (*rule contI*)
apply (*rule is-lub-Pair*)
apply (*rule lub-const*)
apply (*erule cpo-lubI*)
done

lemma *cont-fst*: *cont fst*

apply (*rule contI*)
apply (*simp add*: *thelub-cprod*)
apply (*erule cpo-lubI* [*OF ch2ch-fst*])
done

lemma *cont-snd*: *cont snd*
apply (*rule contI*)
apply (*simp add*: *thelub-cprod*)
apply (*erule cpo-lubI* [*OF ch2ch-snd*])
done

lemma *cont2cont-Pair* [*simp*, *cont2cont*]:
  assumes *f*: *cont* (λ*x*. *f x*)
  assumes *g*: *cont* (λ*x*. *g x*)
  shows *cont* (λ*x*. (*f x*, *g x*))
apply (*rule cont-apply* [*OF f cont-pair1*])
apply (*rule cont-apply* [*OF g cont-pair2*])
apply (*rule cont-const*)
done

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

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

lemma *cont2cont-split*:
  assumes *f1*: ⋀*a b*. *cont* (λ*x*. *f x a b*)
  assumes *f2*: ⋀*x b*. *cont* (λ*a*. *f x a b*)
  assumes *f3*: ⋀*x a*. *cont* (λ*b*. *f x a b*)
  assumes *g*: *cont* (λ*x*. *g x*)
  shows *cont* (λ*x*. *split* (λ*a b*. *f x a b*) (*g x*))
unfolding *split-def*
apply (*rule cont-apply* [*OF g*])
apply (*rule cont-apply* [*OF cont-fst f2*])
apply (*rule cont-apply* [*OF cont-snd f3*])
apply (*rule cont-const*)
apply (*rule f1*)
done

lemma *cont-fst-snd-D1*:
  cont (λ*p*. *f* (*fst p*) (*snd p*)) ⟹ *cont* (λ*x*. *f x y*)
by (*drule cont-compose* [*OF - cont-pair1*], *simp*)

lemma *cont-fst-snd-D2*:
  cont (λ*p*. *f* (*fst p*) (*snd p*)) ⟹ *cont* (λ*y*. *f x y*)
by (*drule cont-compose* [*OF - cont-pair2*], *simp*)

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

shows *cont* $(\lambda x.\ split\ (f\ x)\ (g\ x))$
proof −
note *f1* = *f* [*THEN cont-fst-snd-D1*]
note *f2* = *f* [*THEN cont-fst-snd-D2*, *THEN cont-fst-snd-D1*]
note *f3* = *f* [*THEN cont-fst-snd-D2*, *THEN cont-fst-snd-D2*]
show *?thesis*
unfolding *split-def*
apply (*rule cont-apply* [*OF g*])
apply (*rule cont-apply* [*OF cont-fst f2*])
apply (*rule cont-apply* [*OF cont-snd f3*])
apply (*rule cont-const*)
apply (*rule f1*)
done
qed

8.7 Compactness and chain-finiteness

lemma *fst-below-iff*: $fst\ (x::'a \times 'b) \sqsubseteq y \longleftrightarrow x \sqsubseteq (y,\ snd\ x)$
unfolding *below-prod-def* **by** *simp*

lemma *snd-below-iff*: $snd\ (x::'a \times 'b) \sqsubseteq y \longleftrightarrow x \sqsubseteq (fst\ x,\ y)$
unfolding *below-prod-def* **by** *simp*

lemma *compact-fst*: $compact\ x \Longrightarrow compact\ (fst\ x)$
by (*rule compactI*, *simp add*: *fst-below-iff*)

lemma *compact-snd*: $compact\ x \Longrightarrow compact\ (snd\ x)$
by (*rule compactI*, *simp add*: *snd-below-iff*)

lemma *compact-Pair*: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact\ (x,\ y)$
by (*rule compactI*, *simp add*: *below-prod-def*)

lemma *compact-Pair-iff* [*simp*]: $compact\ (x,\ y) \longleftrightarrow compact\ x \wedge compact\ y$
apply (*safe intro*!: *compact-Pair*)
apply (*drule compact-fst*, *simp*)
apply (*drule compact-snd*, *simp*)
done

instance $*$ :: (*chfin*, *chfin*) *chfin*
apply *intro-classes*
apply (*erule compact-imp-max-in-chain*)
apply (*case-tac* $\bigsqcup i.\ Y\ i$, *simp*)
done

end

9 Cfun: The type of continuous functions

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

default-sort *cpo*

9.1 Definition of continuous function type

lemma *Ex-cont*: $\exists f.\ cont\ f$
by (*rule exI*, *rule cont-const*)

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

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

type-notation (*xsymbols*)
cfun ((- →/ -) [1, 0] 0)

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

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

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

9.2 Syntax for continuous lambda abstraction

syntax -*cabs* :: '*a*

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

To avoid eta-contraction of body:

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

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

```
        val (x,t') = atomic-abs-tr' abs';
      in Syntax.const @{syntax-const -cabs} $ x $ t' end;

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

Syntax for nested abstractions

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

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

parse-ast-translation ⟪
(* rewrite (LAM x y z. t) => (-cabs x (-cabs y (-cabs z t))) *)
(* cf. Syntax.lambda-ast-tr from src/Pure/Syntax/syn-trans.ML *)
  let
    fun Lambda-ast-tr [pats, body] =
          Syntax.fold-ast-p @{syntax-const -cabs}
            (Syntax.unfold-ast @{syntax-const -cargs} pats, body)
      | Lambda-ast-tr asts = raise Syntax.AST (Lambda-ast-tr, asts);
  in [(@{syntax-const -Lambda}, Lambda-ast-tr)] end;
⟫

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

Dummy patterns for continuous abstraction

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

9.3 Continuous function space is pointed

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

instance cfun :: (finite-po, finite-po) finite-po
by (rule typedef-finite-po [OF type-definition-CFun])
```

instance *cfun* :: (*finite-po*, *chfin*) *chfin*
by (*rule typedef-chfin* [*OF type-definition-CFun below-CFun-def*])

instance *cfun* :: (*cpo*, *discrete-cpo*) *discrete-cpo*
by *intro-classes* (*simp add*: *below-CFun-def Rep-CFun-inject*)

instance *cfun* :: (*cpo*, *pcpo*) *pcpo*
by (*rule typedef-pcpo* [*OF type-definition-CFun below-CFun-def UU-CFun*])

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

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

function application is strict in its first argument

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

lemma *LAM-strict* [*simp*]: $(\Lambda\ x.\ \bot) = \bot$
by (*simp add*: *inst-fun-pcpo* [*symmetric*] *Abs-CFun-strict*)

for compatibility with old HOLCF-Version

lemma *inst-cfun-pcpo*: $\bot = (\Lambda\ x.\ \bot)$
by *simp*

9.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

Beta-reduction simproc

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

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

simproc-setup *beta-cfun-proc* (*Abs-CFun f*·*x*) = ⟪
fn phi => fn ss => fn ct =>

```
  let
    val dest = Thm.dest-comb;
    val (f, x) = (apfst (snd o dest o snd o dest) o dest) ct;
    val [T, U] = Thm.dest-ctyp (ctyp-of-term f);
    val tr = instantiate' [SOME T, SOME U] [SOME f, SOME x]
        (mk-meta-eq @{thm beta-cfun});
    val rules = Cont2ContData.get (Simplifier.the-context ss);
    val tac = SOLVED' (REPEAT-ALL-NEW (match-tac rules));
  in SOME (perhaps (SINGLE (tac 1)) tr) end
⟩⟩
```

Eta-equality for continuous functions

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

Extensionality for continuous functions

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

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

Extensionality wrt. ordering for continuous functions

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

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

Congruence for continuous function application

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

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

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

9.5 Continuity of application

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

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

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

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

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

lemma *contlub-cfun-arg*: *chain Y* $\Longrightarrow f{\cdot}(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f{\cdot}(Y\ i))$
by (*rule cont-Rep-CFun2* [*THEN cont2contlubE*])

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

lemma *contlub-cfun-fun*: *chain F* $\Longrightarrow (\bigsqcup i.\ F\ i){\cdot}x = (\bigsqcup i.\ F\ i{\cdot}x)$
by (*rule cont-Rep-CFun1* [*THEN cont2contlubE*])

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

monotonicity of application

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

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 below-trans* [*OF monofun-cfun-fun monofun-cfun-arg*])

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

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

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

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

lemma *ch2ch-Rep-CFun* [*simp*]:
$[\![$*chain F*; *chain Y*$]\!] \Longrightarrow$ *chain* $(\lambda i.\ (F\ i){\cdot}(Y\ i))$
by (*simp add*: *chain-def monofun-cfun*)

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

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

lemma *contlub-cfun*:

$[\![chain\ F;\ chain\ Y]\!] \Longrightarrow (\bigsqcup i.\ F\ i)\cdot(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ F\ i\cdot(Y\ i))$
by (*simp add*: *contlub-cfun-fun contlub-cfun-arg diag-lub*)

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

lemma *contlub-LAM*:
$[\![\bigwedge x.\ chain\ (\lambda i.\ F\ i\ x);\ \bigwedge i.\ cont\ (\lambda x.\ F\ i\ x)]\!]$
$\Longrightarrow (\Lambda\ x.\ \bigsqcup i.\ F\ i\ x) = (\bigsqcup i.\ \Lambda\ x.\ F\ i\ x)$
apply (*simp add*: *thelub-CFun*)
apply (*simp add*: *Abs-CFun-inverse2*)
apply (*simp add*: *thelub-fun ch2ch-lambda*)
done

lemmas *lub-distribs* =
contlub-cfun [*symmetric*]
contlub-LAM [*symmetric*]

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 $'a \to 'b$

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

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 $'a \to 'b$ is chain complete

lemma *lub-cfun*: *chain F* $\Longrightarrow$ *range F* <<| $(\Lambda\ x.\ \bigsqcup i.\ F\ i{\cdot}x)$
by (*simp only*: *contlub-cfun-fun* [*symmetric*] *eta-cfun thelubE*)

lemma *thelub-cfun*: *chain F* $\Longrightarrow$ $(\bigsqcup i.\ F\ i) = (\Lambda\ x.\ \bigsqcup i.\ F\ i{\cdot}x)$
by (*rule lub-cfun* [*THEN thelubI*])

9.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

lemma *cont2cont-Rep-CFun* [*simp*, *cont2cont*]:
assumes *f*: *cont* $(\lambda x.\ f\ x)$
assumes *t*: *cont* $(\lambda x.\ t\ x)$
shows *cont* $(\lambda x.\ (f\ x){\cdot}(t\ x))$
proof −
have *cont* $(\lambda x.$ *Rep-CFun* $(f\ x))$
using *cont-Rep-CFun f* **by** (*rule cont2cont-app3*)
thus *cont* $(\lambda x.\ (f\ x){\cdot}(t\ x))$
using *cont-Rep-CFun2 t* **by** (*rule cont2cont-app2*)
qed

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

lemma *cont2mono-LAM*:
$[\![\bigwedge x.$ *cont* $(\lambda y.\ f\ x\ y);\ \bigwedge y.$ *monofun* $(\lambda x.\ f\ x\ y)]\!]$
$\Longrightarrow$ *monofun* $(\lambda x.\ \Lambda\ y.\ f\ x\ y)$
unfolding *monofun-def expand-cfun-below* **by** *simp*

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

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

lemma *cont2cont-LAM*:
assumes *f1*: $\bigwedge x.$ *cont* $(\lambda y.\ f\ x\ y)$
assumes *f2*: $\bigwedge y.$ *cont* $(\lambda x.\ f\ x\ y)$
shows *cont* $(\lambda x.\ \Lambda\ y.\ f\ x\ y)$
proof (*rule cont-Abs-CFun*)
fix *x*
from *f1* **show** *f x* $\in$ *CFun* **by** (*simp add*: *CFun-def*)
from *f2* **show** *cont f* **by** (*rule cont2cont-lambda*)
qed

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

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

using *f* **by** (*rule cont-fst-snd-D2*)
next
fix *y* :: *'b* **show** *cont* (λ*x*. *f x y*)
using *f* **by** (*rule cont-fst-snd-D1*)
qed

lemma *cont2cont-LAM-discrete* [*simp*, *cont2cont*]:
(⋀*y*::*'a*::*discrete-cpo*. *cont* (λ*x*. *f x y*)) ⟹ *cont* (λ*x*. Λ *y*. *f x y*)
by (*simp add*: *cont2cont-LAM*)

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

9.7 Miscellaneous

Monotonicity of *Abs-CFun*

lemma *semi-monofun-Abs-CFun*:
⟦*cont f*; *cont g*; *f* ⊑ *g*⟧ ⟹ *Abs-CFun f* ⊑ *Abs-CFun g*
by (*simp add*: *below-CFun-def Abs-CFun-inverse2*)

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

lemma *chfin-Rep-CFunR*: *chain* (*Y*::*nat* => *'a*::*cpo*−>*'b*::*chfin*)
==> !*s*. *?* *n*. (*LUB i*. *Y i*)$*s* = *Y n*$*s*
apply (*rule allI*)
apply (*subst contlub-cfun-fun*)
apply *assumption*
apply (*fast intro*!: *thelubI chfin lub-finch2 chfin2finch ch2ch-Rep-CFunL*)
done

lemma *adm-chfindom*: *adm* (λ(*u*::*'a*::*cpo* → *'b*::*chfin*). *P*(*u*·*s*))
by (*rule adm-subst*, *simp*, *rule adm-chfin*)

9.8 Continuous injection-retraction pairs

Continuous retractions are strict.

lemma *retraction-strict*:
∀ *x*. *f*·(*g*·*x*) = *x* ⟹ *f*·⊥ = ⊥
apply (*rule UU-I*)
apply (*drule-tac x*=⊥ **in** *spec*)
apply (*erule subst*)
apply (*rule monofun-cfun-arg*)
apply (*rule minimal*)
done

lemma *injection-eq*:
∀ *x*. *f*·(*g*·*x*) = *x* ⟹ (*g*·*x* = *g*·*y*) = (*x* = *y*)
apply (*rule iffI*)
apply (*drule-tac f*=*f* **in** *cfun-arg-cong*)

apply *simp*
apply *simp*
done

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

lemma *flat-codom*:

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

9.9 Identity and composition

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

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

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

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*)

lemma *cfcomp-LAM*: $cont\ g \Longrightarrow f\ oo\ (\Lambda\ x.\ g\ x) = (\Lambda\ x.\ f{\cdot}(g\ x))$
by (*simp add*: *cfcomp1*)

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

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

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

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

by (*rule ext-cfun*, *simp*)

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

9.10 Strictified functions

default-sort *pcpo*

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

results about strictify

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

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

lemma *cont-strictify2*: *cont* (λ*x*. *if x* = ⊥ *then* ⊥ *else f*·*x*)
apply (*rule contI2*)
apply (*rule monofun-strictify2*)
apply (*case-tac* (⨆ *i*. *Y i*) = ⊥, *simp*)
apply (*simp add*: *contlub-cfun-arg del*: *if-image-distrib*)
apply (*drule chain-UU-I-inverse2*, *clarify*, *rename-tac j*)
apply (*rule lub-mono2*, *rule-tac x*=*j* **in** *exI*, *simp-all*)
apply (*auto dest*!: *chain-mono-less*)
done

lemma *strictify-conv-if*: *strictify*·*f*·*x* = (*if x* = ⊥ *then* ⊥ *else f*·*x*)
unfolding *strictify-def*
by (*simp add*: *cont-strictify1 cont-strictify2 cont2cont-LAM*)

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

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

9.11 Continuity of let-bindings

lemma *cont2cont-Let*:
assumes *f*: *cont* (λ*x*. *f x*)
assumes *g1*: ⋀*y*. *cont* (λ*x*. *g x y*)
assumes *g2*: ⋀*x*. *cont* (λ*y*. *g x y*)
shows *cont* (λ*x*. *let y* = *f x in g x y*)

unfolding *Let-def* **using** *f g2 g1* **by** (*rule cont-apply*)

lemma *cont2cont-Let'* [*simp*, *cont2cont*]:
assumes *f*: *cont* (λ*x*. *f x*)
assumes *g*: *cont* (λ*p*. *g* (*fst p*) (*snd p*))
shows *cont* (λ*x*. *let y* = *f x in g x y*)
using *f*
proof (*rule cont2cont-Let*)
fix *x* **show** *cont* (λ*y*. *g x y*)
using *g* **by** (*rule cont-fst-snd-D2*)
next
fix *y* **show** *cont* (λ*x*. *g x y*)
using *g* **by** (*rule cont-fst-snd-D1*)
qed

end

10 Deflation: Continuous deflations and ep-pairs

theory *Deflation*
imports *Cfun*
begin

default-sort *cpo*

10.1 Continuous deflations

locale *deflation* =
fixes *d* :: 'a → 'a
assumes *idem*: ⋀*x*. *d*·(*d*·*x*) = *d*·*x*
assumes *below*: ⋀*x*. *d*·*x* ⊑ *x*
begin

lemma *below-ID*: *d* ⊑ *ID*
by (*rule below-cfun-ext*, *simp add*: *below*)

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

lemma *fixes-eq-range*: {*x*. *d*·*x* = *x*} = *range* (λ*x*. *d*·*x*)
by (*auto simp add*: *eq-sym-conv idem*)

lemma *range-eq-fixes*: *range* (λ*x*. *d*·*x*) = {*x*. *d*·*x* = *x*}
by (*auto simp add*: *eq-sym-conv idem*)

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

lemma *belowI*:
assumes *f*: ⋀*x*. *d*·*x* = *x* ⟹ *f*·*x* = *x* **shows** *d* ⊑ *f*

proof (*rule below-cfun-ext*)
fix x
from *below* **have** $f{\cdot}(d{\cdot}x) \sqsubseteq f{\cdot}x$ **by** (*rule monofun-cfun-arg*)
also from *idem* **have** $f{\cdot}(d{\cdot}x) = d{\cdot}x$ **by** (*rule f*)
finally show $d{\cdot}x \sqsubseteq f{\cdot}x$ **.**
qed

lemma *belowD*: $[\![f \sqsubseteq d;\ f{\cdot}x = x]\!] \Longrightarrow d{\cdot}x = x$
proof (*rule below-antisym*)
from *below* **show** $d{\cdot}x \sqsubseteq x$ **.**
next
assume $f \sqsubseteq d$
hence $f{\cdot}x \sqsubseteq d{\cdot}x$ **by** (*rule monofun-cfun-fun*)
also assume $f{\cdot}x = x$
finally show $x \sqsubseteq d{\cdot}x$ **.**
qed

end

lemma *deflation-strict*: *deflation* $d \Longrightarrow d{\cdot}\bot = \bot$
by (*rule deflation.below* [*THEN UU-I*])

lemma *adm-deflation*: *adm* $(\lambda d.\ \mathit{deflation}\ d)$
by (*simp add*: *deflation-def*)

lemma *deflation-ID*: *deflation ID*
by (*simp add*: *deflation.intro*)

lemma *deflation-UU*: *deflation* $\bot$
by (*simp add*: *deflation.intro*)

lemma *deflation-below-iff*:
$[\![\mathit{deflation}\ p;\ \mathit{deflation}\ q]\!] \Longrightarrow p \sqsubseteq q \longleftrightarrow (\forall x.\ p{\cdot}x = x \longrightarrow q{\cdot}x = x)$
apply *safe*
apply (*simp add*: *deflation.belowD*)
apply (*simp add*: *deflation.belowI*)
done

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

lemma *deflation-below-comp1*:
assumes *deflation f*
assumes *deflation g*
shows $f \sqsubseteq g \Longrightarrow f{\cdot}(g{\cdot}x) = f{\cdot}x$
proof (*rule below-antisym*)
interpret *g*: *deflation g* **by** *fact*
from *g.below* **show** $f{\cdot}(g{\cdot}x) \sqsubseteq f{\cdot}x$ **by** (*rule monofun-cfun-arg*)
next
interpret *f*: *deflation f* **by** *fact*

assume $f \sqsubseteq g$ **hence** $f{\cdot}x \sqsubseteq g{\cdot}x$ **by** (*rule monofun-cfun-fun*)
hence $f{\cdot}(f{\cdot}x) \sqsubseteq f{\cdot}(g{\cdot}x)$ **by** (*rule monofun-cfun-arg*)
also have $f{\cdot}(f{\cdot}x) = f{\cdot}x$ **by** (*rule f.idem*)
finally show $f{\cdot}x \sqsubseteq f{\cdot}(g{\cdot}x)$ **.**
qed

lemma *deflation-below-comp2*:
$[\![\mathit{deflation}\ f;\ \mathit{deflation}\ g;\ f \sqsubseteq g]\!] \Longrightarrow g{\cdot}(f{\cdot}x) = f{\cdot}x$
by (*simp only*: *deflation.belowD deflation.idem*)

10.2 Deflations with finite range

lemma *finite-range-imp-finite-fixes*:
$\mathit{finite}\ (\mathit{range}\ f) \Longrightarrow \mathit{finite}\ \{x.\ f\,x = x\}$
proof −
have $\{x.\ f\,x = x\} \subseteq \mathit{range}\ f$
by (*clarify*, *erule subst*, *rule rangeI*)
moreover assume *finite* (*range f*)
ultimately show *finite* $\{x.\ f\,x = x\}$
by (*rule finite-subset*)
qed

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

lemma *finite-range*: *finite* (*range* $(\lambda x.\ d{\cdot}x)$)
by (*simp add*: *range-eq-fixes finite-fixes*)

lemma *finite-image*: *finite* $((\lambda x.\ d{\cdot}x)\ `\ A)$
by (*rule finite-subset* [*OF image-mono* [*OF subset-UNIV*] *finite-range*])

lemma *compact*: *compact* $(d{\cdot}x)$
proof (*rule compactI2*)
fix $Y :: \mathit{nat} \Rightarrow {'a}$
assume Y: *chain* Y
have *finite-chain* $(\lambda i.\ d{\cdot}(Y\ i))$
proof (*rule finite-range-imp-finch*)
show *chain* $(\lambda i.\ d{\cdot}(Y\ i))$
using Y **by** *simp*
have *range* $(\lambda i.\ d{\cdot}(Y\ i)) \subseteq \mathit{range}\ (\lambda x.\ d{\cdot}x)$
by *clarsimp*
thus *finite* (*range* $(\lambda i.\ d{\cdot}(Y\ i))$)
using *finite-range* **by** (*rule finite-subset*)
qed
hence $\exists j.\ (\bigsqcup i.\ d{\cdot}(Y\ i)) = d{\cdot}(Y\ j)$
by (*simp add*: *finite-chain-def maxinch-is-thelub* Y)
then obtain j **where** j: $(\bigsqcup i.\ d{\cdot}(Y\ i)) = d{\cdot}(Y\ j)$ **..**

assume $d{\cdot}x \sqsubseteq (\bigsqcup i.\ Y\ i)$
hence $d{\cdot}(d{\cdot}x) \sqsubseteq d{\cdot}(\bigsqcup i.\ Y\ i)$
by (*rule monofun-cfun-arg*)
hence $d{\cdot}x \sqsubseteq (\bigsqcup i.\ d{\cdot}(Y\ i))$
by (*simp add*: *contlub-cfun-arg Y idem*)
hence $d{\cdot}x \sqsubseteq d{\cdot}(Y\ j)$
using j **by** *simp*
hence $d{\cdot}x \sqsubseteq Y\ j$
using *below* **by** (*rule below-trans*)
thus $\exists j.\ d{\cdot}x \sqsubseteq Y\ j$ **..**
qed

end

10.3 Continuous embedding-projection pairs

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

lemma *e-below-iff* [*simp*]: $e{\cdot}x \sqsubseteq e{\cdot}y \longleftrightarrow x \sqsubseteq y$
proof
assume $e{\cdot}x \sqsubseteq e{\cdot}y$
hence $p{\cdot}(e{\cdot}x) \sqsubseteq p{\cdot}(e{\cdot}y)$ **by** (*rule monofun-cfun-arg*)
thus $x \sqsubseteq y$ **by** *simp*
next
assume $x \sqsubseteq y$
thus $e{\cdot}x \sqsubseteq e{\cdot}y$ **by** (*rule monofun-cfun-arg*)
qed

lemma *e-eq-iff* [*simp*]: $e{\cdot}x = e{\cdot}y \longleftrightarrow x = y$
unfolding *po-eq-conv e-below-iff* **..**

lemma *p-eq-iff*:
$[\![e{\cdot}(p{\cdot}x) = x;\ e{\cdot}(p{\cdot}y) = y]\!] \Longrightarrow p{\cdot}x = p{\cdot}y \longleftrightarrow x = y$
by (*safe*, *erule subst*, *erule subst*, *simp*)

lemma *p-inverse*: $(\exists x.\ y = e{\cdot}x) = (e{\cdot}(p{\cdot}y) = y)$
by (*auto*, *rule exI*, *erule sym*)

lemma *e-below-iff-below-p*: $e{\cdot}x \sqsubseteq y \longleftrightarrow x \sqsubseteq p{\cdot}y$
proof
assume $e{\cdot}x \sqsubseteq y$
then have $p{\cdot}(e{\cdot}x) \sqsubseteq p{\cdot}y$ **by** (*rule monofun-cfun-arg*)
then show $x \sqsubseteq p{\cdot}y$ **by** *simp*
next
assume $x \sqsubseteq p{\cdot}y$

then have $e{\cdot}x \sqsubseteq e{\cdot}(p{\cdot}y)$ **by** (*rule monofun-cfun-arg*)
then show $e{\cdot}x \sqsubseteq y$ **using** *e-p-below* **by** (*rule below-trans*)
qed

lemma *compact-e-rev*: *compact* $(e{\cdot}x) \Longrightarrow$ *compact* x
proof −
assume *compact* $(e{\cdot}x)$
hence *adm* $(\lambda y.\ \neg\ e{\cdot}x \sqsubseteq y)$ **by** (*rule compactD*)
hence *adm* $(\lambda y.\ \neg\ e{\cdot}x \sqsubseteq e{\cdot}y)$ **by** (*rule adm-subst* [*OF cont-Rep-CFun2*])
hence *adm* $(\lambda y.\ \neg\ x \sqsubseteq y)$ **by** *simp*
thus *compact* x **by** (*rule compactI*)
qed

lemma *compact-e*: *compact* $x \Longrightarrow$ *compact* $(e{\cdot}x)$
proof −
assume *compact* x
hence *adm* $(\lambda y.\ \neg\ x \sqsubseteq y)$ **by** (*rule compactD*)
hence *adm* $(\lambda y.\ \neg\ x \sqsubseteq p{\cdot}y)$ **by** (*rule adm-subst* [*OF cont-Rep-CFun2*])
hence *adm* $(\lambda y.\ \neg\ e{\cdot}x \sqsubseteq y)$ **by** (*simp add*: *e-below-iff-below-p*)
thus *compact* $(e{\cdot}x)$ **by** (*rule compactI*)
qed

lemma *compact-e-iff*: *compact* $(e{\cdot}x) \longleftrightarrow$ *compact* x
by (*rule iffI* [*OF compact-e-rev compact-e*])

Deflations from ep-pairs

lemma *deflation-e-p*: *deflation* $(e\ oo\ p)$
by (*simp add*: *deflation.intro e-p-below*)

lemma *deflation-e-d-p*:
assumes *deflation d*
shows *deflation* $(e\ oo\ d\ oo\ p)$
proof
interpret *deflation d* **by** *fact*
fix $x :: 'b$
show $(e\ oo\ d\ oo\ p){\cdot}((e\ oo\ d\ oo\ p){\cdot}x) = (e\ oo\ d\ oo\ p){\cdot}x$
by (*simp add*: *idem*)
show $(e\ oo\ d\ oo\ p){\cdot}x \sqsubseteq x$
by (*simp add*: *e-below-iff-below-p below*)
qed

lemma *finite-deflation-e-d-p*:
assumes *finite-deflation d*
shows *finite-deflation* $(e\ oo\ d\ oo\ p)$
proof
interpret *finite-deflation d* **by** *fact*
fix $x :: 'b$
show $(e\ oo\ d\ oo\ p){\cdot}((e\ oo\ d\ oo\ p){\cdot}x) = (e\ oo\ d\ oo\ p){\cdot}x$
by (*simp add*: *idem*)

```
  show (e oo d oo p)·x ⊑ x
    by (simp add: e-below-iff-below-p below)
  have finite ((λx. e·x) ‘ (λx. d·x) ‘ range (λx. p·x))
    by (simp add: finite-image)
  hence finite (range (λx. (e oo d oo p)·x))
    by (simp add: image-image)
  thus finite {x. (e oo d oo p)·x = x}
    by (rule finite-range-imp-finite-fixes)
qed

lemma deflation-p-d-e:
  assumes deflation d
  assumes d: ⋀x. d·x ⊑ e·(p·x)
  shows deflation (p oo d oo e)
proof −
  interpret d: deflation d by fact
  {
    fix x
    have d·(e·x) ⊑ e·x
      by (rule d.below)
    hence p·(d·(e·x)) ⊑ p·(e·x)
      by (rule monofun-cfun-arg)
    hence (p oo d oo e)·x ⊑ x
      by simp
  }
  note p-d-e-below = this
  show ?thesis
  proof
    fix x
    show (p oo d oo e)·x ⊑ x
      by (rule p-d-e-below)
  next
    fix x
    show (p oo d oo e)·((p oo d oo e)·x) = (p oo d oo e)·x
    proof (rule below-antisym)
      show (p oo d oo e)·((p oo d oo e)·x) ⊑ (p oo d oo e)·x
        by (rule p-d-e-below)
      have p·(d·(d·(d·(e·x)))) ⊑ p·(d·(e·(p·(d·(e·x)))))
        by (intro monofun-cfun-arg d)
      hence p·(d·(e·x)) ⊑ p·(d·(e·(p·(d·(e·x)))))
        by (simp only: d.idem)
      thus (p oo d oo e)·x ⊑ (p oo d oo e)·((p oo d oo e)·x)
        by simp
    qed
  qed
qed

lemma finite-deflation-p-d-e:
  assumes finite-deflation d
```

```
  assumes d: ⋀x. d·x ⊑ e·(p·x)
  shows finite-deflation (p oo d oo e)
proof −
  interpret d: finite-deflation d by fact
  show ?thesis
  proof (intro-locales)
    have deflation d ..
    thus deflation (p oo d oo e)
      using d by (rule deflation-p-d-e)
  next
    show finite-deflation-axioms (p oo d oo e)
    proof
      have finite ((λx. d·x) ‘ range (λx. e·x))
        by (rule d.finite-image)
      hence finite ((λx. p·x) ‘ (λx. d·x) ‘ range (λx. e·x))
        by (rule finite-imageI)
      hence finite (range (λx. (p oo d oo e)·x))
        by (simp add: image-image)
      thus finite {x. (p oo d oo e)·x = x}
        by (rule finite-range-imp-finite-fixes)
    qed
  qed
qed

end
```

10.4 Uniqueness of ep-pairs

```
lemma ep-pair-unique-e-lemma:
  assumes 1: ep-pair e1 p and 2: ep-pair e2 p
  shows e1 ⊑ e2
proof (rule below-cfun-ext)
  fix x
  have e1·(p·(e2·x)) ⊑ e2·x
    by (rule ep-pair.e-p-below [OF 1])
  thus e1·x ⊑ e2·x
    by (simp only: ep-pair.e-inverse [OF 2])
qed

lemma ep-pair-unique-e:
  ⟦ep-pair e1 p; ep-pair e2 p⟧ ⟹ e1 = e2
by (fast intro: below-antisym elim: ep-pair-unique-e-lemma)

lemma ep-pair-unique-p-lemma:
  assumes 1: ep-pair e p1 and 2: ep-pair e p2
  shows p1 ⊑ p2
proof (rule below-cfun-ext)
  fix x
  have e·(p1·x) ⊑ x
```

by (*rule ep-pair.e-p-below* [*OF 1*])
hence $p2{\cdot}(e{\cdot}(p1{\cdot}x)) \sqsubseteq p2{\cdot}x$
by (*rule monofun-cfun-arg*)
thus $p1{\cdot}x \sqsubseteq p2{\cdot}x$
by (*simp only*: *ep-pair.e-inverse* [*OF 2*])
qed

lemma *ep-pair-unique-p*:
$[\![$*ep-pair e p1*; *ep-pair e p2*$]\!] \Longrightarrow$ *p1* = *p2*
by (*fast intro*: *below-antisym elim*: *ep-pair-unique-p-lemma*)

10.5 Composing ep-pairs

lemma *ep-pair-ID-ID*: *ep-pair ID ID*
by *default simp-all*

lemma *ep-pair-comp*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*e2 oo e1*) (*p1 oo p2*)
proof
interpret *ep1*: *ep-pair e1 p1* **by** *fact*
interpret *ep2*: *ep-pair e2 p2* **by** *fact*
fix *x y*
show $(p1\ oo\ p2){\cdot}((e2\ oo\ e1){\cdot}x) = x$
by *simp*
have $e1{\cdot}(p1{\cdot}(p2{\cdot}y)) \sqsubseteq p2{\cdot}y$
by (*rule ep1.e-p-below*)
hence $e2{\cdot}(e1{\cdot}(p1{\cdot}(p2{\cdot}y))) \sqsubseteq e2{\cdot}(p2{\cdot}y)$
by (*rule monofun-cfun-arg*)
also have $e2{\cdot}(p2{\cdot}y) \sqsubseteq y$
by (*rule ep2.e-p-below*)
finally show $(e2\ oo\ e1){\cdot}((p1\ oo\ p2){\cdot}y) \sqsubseteq y$
by *simp*
qed

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

lemma *e-strict* [*simp*]: $e{\cdot}\bot = \bot$
proof −
have $\bot \sqsubseteq p{\cdot}\bot$ **by** (*rule minimal*)
hence $e{\cdot}\bot \sqsubseteq e{\cdot}(p{\cdot}\bot)$ **by** (*rule monofun-cfun-arg*)
also have $e{\cdot}(p{\cdot}\bot) \sqsubseteq \bot$ **by** (*rule e-p-below*)
finally show $e{\cdot}\bot = \bot$ **by** *simp*
qed

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

by (*rule e-eq-iff* [**where** $y{=}\bot$, *unfolded e-strict*])

lemma *e-defined*: $x \neq \bot \Longrightarrow e{\cdot}x \neq \bot$
by *simp*

lemma *p-strict* [*simp*]: $p{\cdot}\bot = \bot$
by (*rule e-inverse* [**where** $x{=}\bot$, *unfolded e-strict*])

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

end

end

11 Bifinite: Bifinite domains and approximation

theory *Bifinite*
imports *Deflation*
begin

11.1 Omega-profinite and bifinite domains

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

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

lemma *approx-below*: $approx\ i{\cdot}x \sqsubseteq x$
proof −
  have *chain* $(\lambda i.\ approx\ i{\cdot}x)$ **by** *simp*
  hence $approx\ i{\cdot}x \sqsubseteq (\bigsqcup i.\ approx\ i{\cdot}x)$ **by** (*rule is-ub-thelub*)
  thus $approx\ i{\cdot}x \sqsubseteq x$ **by** *simp*
qed

lemma *finite-deflation-approx*: *finite-deflation* (*approx i*)
proof
  fix $x :: 'a$
  show $approx\ i{\cdot}(approx\ i{\cdot}x) = approx\ i{\cdot}x$
    by (*rule approx-idem*)
  show $approx\ i{\cdot}x \sqsubseteq x$
    by (*rule approx-below*)
  show *finite* $\{x.\ approx\ i{\cdot}x = x\}$
    by (*rule finite-fixes-approx*)
qed

interpretation *approx*: *finite-deflation approx i*
by (*rule finite-deflation-approx*)

lemma (**in** *deflation*) *deflation*: *deflation d* **..**

lemma *deflation-approx*: *deflation* (*approx i*)
by (*rule approx.deflation*)

lemma *lub-approx* [*simp*]: $(\bigsqcup i.\ approx\ i) = (\Lambda\ x.\ x)$
by (*rule ext-cfun*, *simp add*: *contlub-cfun-fun*)

lemma *approx-strict* [*simp*]: $approx\ i{\cdot}\bot = \bot$
by (*rule UU-I*, *rule approx-below*)

lemma *approx-approx1*:
$i \leq j \Longrightarrow approx\ i{\cdot}(approx\ j{\cdot}x) = approx\ i{\cdot}x$
apply (*rule deflation-below-comp1* [*OF deflation-approx deflation-approx*])
apply (*erule chain-mono* [*OF chain-approx*])
done

lemma *approx-approx2*:
$j \leq i \Longrightarrow approx\ i{\cdot}(approx\ j{\cdot}x) = approx\ j{\cdot}x$
apply (*rule deflation-below-comp2* [*OF deflation-approx deflation-approx*])
apply (*erule chain-mono* [*OF chain-approx*])
done

lemma *approx-approx* [*simp*]:
$approx\ i{\cdot}(approx\ j{\cdot}x) = approx\ (min\ i\ j){\cdot}x$
apply (*rule-tac x*=*i* **and** *y*=*j* **in** *linorder-le-cases*)
apply (*simp add*: *approx-approx1 min-def*)
apply (*simp add*: *approx-approx2 min-def*)
done

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

lemma *finite-range-approx*: *finite* (*range* $(\lambda x.\ approx\ i{\cdot}x)$)
by (*rule approx.finite-range*)

lemma *compact-approx* [*simp*]: *compact* $(approx\ n{\cdot}x)$
by (*rule approx.compact*)

lemma *profinite-compact-eq-approx*: $compact\ x \Longrightarrow \exists\, i.\ approx\ i{\cdot}x = x$
by (*rule admD2*, *simp-all*)

lemma *profinite-compact-iff*: $compact\ x \longleftrightarrow (\exists\, n.\ approx\ n{\cdot}x = x)$
apply (*rule iffI*)
apply (*erule profinite-compact-eq-approx*)

apply (*erule exE*)
apply (*erule subst*)
apply (*rule compact-approx*)
done

lemma *approx-induct*:
assumes *adm*: *adm P* **and** *P*: $\bigwedge n\ x.\ P\ (approx\ n{\cdot}x)$
shows *P x*
proof −
have *P* $(\bigsqcup n.\ approx\ n{\cdot}x)$
by (*rule admD* [*OF adm*], *simp*, *simp add*: *P*)
thus *P x* **by** *simp*
qed

lemma *profinite-below-ext*: $(\bigwedge i.\ approx\ i{\cdot}x \sqsubseteq approx\ i{\cdot}y) \Longrightarrow x \sqsubseteq y$
apply (*subgoal-tac* $(\bigsqcup i.\ approx\ i{\cdot}x) \sqsubseteq (\bigsqcup i.\ approx\ i{\cdot}y)$, *simp*)
apply (*rule lub-mono*, *simp*, *simp*, *simp*)
done

11.2 Instance for product type

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

lemma *cprod-map-Pair* [*simp*]: *cprod-map*·*f*·*g*·(*x*, *y*) = (*f*·*x*, *g*·*y*)
unfolding *cprod-map-def* **by** *simp*

lemma *cprod-map-ID*: *cprod-map*·*ID*·*ID* = *ID*
unfolding *expand-cfun-eq* **by** *auto*

lemma *cprod-map-map*:
cprod-map·*f1*·*g1*·(*cprod-map*·*f2*·*g2*·*p*) =
cprod-map·(Λ *x*. *f1*·(*f2*·*x*))·(Λ *x*. *g1*·(*g2*·*x*))·*p*
by (*induct p*) *simp*

lemma *ep-pair-cprod-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*cprod-map*·*e1*·*e2*) (*cprod-map*·*p1*·*p2*)
proof
interpret *e1p1*: *ep-pair e1 p1* **by** *fact*
interpret *e2p2*: *ep-pair e2 p2* **by** *fact*
fix *x* **show** *cprod-map*·*p1*·*p2*·(*cprod-map*·*e1*·*e2*·*x*) = *x*
by (*induct x*) *simp*
fix *y* **show** *cprod-map*·*e1*·*e2*·(*cprod-map*·*p1*·*p2*·*y*) ⊑ *y*
by (*induct y*) (*simp add*: *e1p1.e-p-below e2p2.e-p-below*)
qed

lemma *deflation-cprod-map*:
  assumes *deflation d1* **and** *deflation d2*
  shows *deflation* (*cprod-map*·*d1*·*d2*)
proof
  interpret *d1*: *deflation d1* **by** *fact*
  interpret *d2*: *deflation d2* **by** *fact*
  fix *x*
  show *cprod-map*·*d1*·*d2*·(*cprod-map*·*d1*·*d2*·*x*) = *cprod-map*·*d1*·*d2*·*x*
    by (*induct x*) (*simp add*: *d1.idem d2.idem*)
  show *cprod-map*·*d1*·*d2*·*x* ⊑ *x*
    by (*induct x*) (*simp add*: *d1.below d2.below*)
qed

lemma *finite-deflation-cprod-map*:
  assumes *finite-deflation d1* **and** *finite-deflation d2*
  shows *finite-deflation* (*cprod-map*·*d1*·*d2*)
proof (*intro finite-deflation.intro finite-deflation-axioms.intro*)
  interpret *d1*: *finite-deflation d1* **by** *fact*
  interpret *d2*: *finite-deflation d2* **by** *fact*
  have *deflation d1* **and** *deflation d2* **by** *fact*+
  thus *deflation* (*cprod-map*·*d1*·*d2*) **by** (*rule deflation-cprod-map*)
  have {*p*. *cprod-map*·*d1*·*d2*·*p* = *p*} ⊆ {*x*. *d1*·*x* = *x*} × {*y*. *d2*·*y* = *y*}
    by *clarsimp*
  thus *finite* {*p*. *cprod-map*·*d1*·*d2*·*p* = *p*}
    by (*rule finite-subset*, *simp add*: *d1.finite-fixes d2.finite-fixes*)
qed

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

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

instance proof
  fix *i* :: *nat* **and** *x* :: *'a* × *'b*
  show *chain* (*approx* :: *nat* ⇒ *'a* × *'b* → *'a* × *'b*)
    unfolding *approx-prod-def* **by** *simp*
  show (⨆ *i*. *approx i*·*x*) = *x*
    unfolding *approx-prod-def cprod-map-def*
    by (*simp add*: *lub-distribs thelub-Pair*)
  show *approx i*·(*approx i*·*x*) = *approx i*·*x*
    unfolding *approx-prod-def cprod-map-def* **by** *simp*
  have {*x*::*'a* × *'b*. *approx i*·*x* = *x*} ⊆
      {*x*::*'a*. *approx i*·*x* = *x*} × {*x*::*'b*. *approx i*·*x* = *x*}
    unfolding *approx-prod-def* **by** *clarsimp*
  thus *finite* {*x*::*'a* × *'b*. *approx i*·*x* = *x*}
    by (*rule finite-subset*,
      intro finite-cartesian-product finite-fixes-approx)

qed

end

instance $*$ :: (*bifinite*, *bifinite*) *bifinite* **..**

lemma *approx-Pair* [*simp*]:
approx i·(x, y) = (approx i·x, approx i·y)
unfolding *approx-prod-def* **by** *simp*

lemma *fst-approx*: *fst (approx i·p) = approx i·(fst p)*
by (*induct p*, *simp*)

lemma *snd-approx*: *snd (approx i·p) = approx i·(snd p)*
by (*induct p*, *simp*)

11.3 Instance for continuous function space

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

lemma *cfun-map-beta* [*simp*]: *cfun-map·a·b·f·x = b·(f·(a·x))*
unfolding *cfun-map-def* **by** *simp*

lemma *cfun-map-ID*: *cfun-map·ID·ID = ID*
unfolding *expand-cfun-eq* **by** *simp*

lemma *cfun-map-map*:
cfun-map·f1·g1·(cfun-map·f2·g2·p) =
cfun-map·(Λ x. f2·(f1·x))·(Λ x. g1·(g2·x))·p
by (*rule ext-cfun*) *simp*

lemma *ep-pair-cfun-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair (cfun-map·p1·e2) (cfun-map·e1·p2)*
proof
interpret *e1p1*: *ep-pair e1 p1* **by** *fact*
interpret *e2p2*: *ep-pair e2 p2* **by** *fact*
fix *f* **show** *cfun-map·e1·p2·(cfun-map·p1·e2·f) = f*
by (*simp add*: *expand-cfun-eq*)
fix *g* **show** *cfun-map·p1·e2·(cfun-map·e1·p2·g) ⊑ g*
apply (*rule below-cfun-ext*, *simp*)
apply (*rule below-trans* [*OF e2p2.e-p-below*])
apply (*rule monofun-cfun-arg*)
apply (*rule e1p1.e-p-below*)
done
qed

lemma *deflation-cfun-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*cfun-map*·*d1*·*d2*)
proof
interpret *d1*: *deflation d1* **by** *fact*
interpret *d2*: *deflation d2* **by** *fact*
fix *f*
show *cfun-map*·*d1*·*d2*·(*cfun-map*·*d1*·*d2*·*f*) = *cfun-map*·*d1*·*d2*·*f*
by (*simp add*: *expand-cfun-eq d1.idem d2.idem*)
show *cfun-map*·*d1*·*d2*·*f* ⊑ *f*
apply (*rule below-cfun-ext*, *simp*)
apply (*rule below-trans* [*OF d2.below*])
apply (*rule monofun-cfun-arg*)
apply (*rule d1.below*)
done
qed

lemma *finite-range-cfun-map*:
assumes *a*: *finite* (*range* (λ*x*. *a*·*x*))
assumes *b*: *finite* (*range* (λ*y*. *b*·*y*))
shows *finite* (*range* (λ*f*. *cfun-map*·*a*·*b*·*f*)) (**is** *finite* (*range* *?h*))
proof (*rule finite-imageD*)
let *?f* = λ*g*. *range* (λ*x*. (*a*·*x*, *g*·*x*))
show *finite* (*?f* ‘ *range* *?h*)
proof (*rule finite-subset*)
let *?B* = *Pow* (*range* (λ*x*. *a*·*x*) × *range* (λ*y*. *b*·*y*))
show *?f* ‘ *range* *?h* ⊆ *?B*
by *clarsimp*
show *finite ?B*
by (*simp add*: *a b*)
qed
show *inj-on ?f* (*range ?h*)
proof (*rule inj-onI*, *rule ext-cfun*, *clarsimp*)
fix *x f g*
assume *range* (λ*x*. (*a*·*x*, *b*·(*f*·(*a*·*x*)))) = *range* (λ*x*. (*a*·*x*, *b*·(*g*·(*a*·*x*))))
hence *range* (λ*x*. (*a*·*x*, *b*·(*f*·(*a*·*x*)))) ⊆ *range* (λ*x*. (*a*·*x*, *b*·(*g*·(*a*·*x*))))
by (*rule equalityD1*)
hence (*a*·*x*, *b*·(*f*·(*a*·*x*))) ∈ *range* (λ*x*. (*a*·*x*, *b*·(*g*·(*a*·*x*))))
by (*simp add*: *subset-eq*)
then obtain *y* **where** (*a*·*x*, *b*·(*f*·(*a*·*x*))) = (*a*·*y*, *b*·(*g*·(*a*·*y*)))
by (*rule rangeE*)
thus *b*·(*f*·(*a*·*x*)) = *b*·(*g*·(*a*·*x*))
by *clarsimp*
qed
qed

lemma *finite-deflation-cfun-map*:
assumes *finite-deflation d1* **and** *finite-deflation d2*

```
  shows finite-deflation (cfun-map·d1·d2)
proof (intro finite-deflation.intro finite-deflation-axioms.intro)
  interpret d1: finite-deflation d1 by fact
  interpret d2: finite-deflation d2 by fact
  have deflation d1 and deflation d2 by fact+
  thus deflation (cfun-map·d1·d2) by (rule deflation-cfun-map)
  have finite (range (λf. cfun-map·d1·d2·f))
    using d1.finite-range d2.finite-range
    by (rule finite-range-cfun-map)
  thus finite {f. cfun-map·d1·d2·f = f}
    by (rule finite-range-imp-finite-fixes)
qed

instantiation cfun :: (profinite, profinite) profinite
begin

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

instance proof
  show chain (approx :: nat ⇒ ('a → 'b) → ('a → 'b))
    unfolding approx-cfun-def by simp
next
  fix x :: 'a → 'b
  show (⨆ i. approx i·x) = x
    unfolding approx-cfun-def cfun-map-def
    by (simp add: lub-distribs eta-cfun)
next
  fix i :: nat and x :: 'a → 'b
  show approx i·(approx i·x) = approx i·x
    unfolding approx-cfun-def cfun-map-def by simp
next
  fix i :: nat
  show finite {x::'a → 'b. approx i·x = x}
    unfolding approx-cfun-def
    by (intro finite-deflation.finite-fixes
              finite-deflation-cfun-map
              finite-deflation-approx)
qed

end

instance cfun :: (profinite, bifinite) bifinite ..

lemma approx-cfun: approx n·f·x = approx n·(f·(approx n·x))
by (simp add: approx-cfun-def)

end
```

12 Up: The type of lifted values

theory *Up*
imports *Bifinite*
begin

default-sort *cpo*

12.1 Definition of new type for lifting

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

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

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

12.2 Ordering on lifted cpo

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

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

instance ..
end

lemma *minimal-up* [*iff*]: $Ibottom \sqsubseteq z$
by (*simp add*: *below-up-def*)

lemma *not-Iup-below* [*iff*]: $\neg\ Iup\ x \sqsubseteq Ibottom$
by (*simp add*: *below-up-def*)

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

12.3 Lifted cpo is a partial order

instance u :: (*cpo*) *po*
proof
  fix x :: $'a\ u$
  show $x \sqsubseteq x$

unfolding *below-up-def* **by** (*simp split*: *u.split*)
next
fix *x y* :: *'a u*
assume *x* ⊑ *y y* ⊑ *x* **thus** *x* = *y*
unfolding *below-up-def*
by (*auto split*: *u.split-asm intro*: *below-antisym*)
next
fix *x y z* :: *'a u*
assume *x* ⊑ *y y* ⊑ *z* **thus** *x* ⊑ *z*
unfolding *below-up-def*
by (*auto split*: *u.split-asm intro*: *below-trans*)
qed

lemma *u-UNIV*: *UNIV* = *insert Ibottom* (*range Iup*)
by (*auto*, *case-tac x*, *auto*)

instance *u* :: (*finite-po*) *finite-po*
by (*intro-classes*, *simp add*: *u-UNIV*)

12.4 Lifted cpo is a cpo

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

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

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

lemma *up-lemma2*:
⟦*chain Y*; *Y j* ≠ *Ibottom*⟧ ⟹ *Y* (*i* + *j*) ≠ *Ibottom*
apply (*erule contrapos-nn*)
apply (*drule-tac i*=*j* **and** *j*=*i* + *j* **in** *chain-mono*)
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-below* [*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 cpo-lubI*)
apply (*erule* (*1*) *up-lemma4*)
done

lemma *up-chain-lemma*:
$chain\ Y \Longrightarrow$
$(\exists A.\ chain\ A \wedge (\bigsqcup i.\ Y\ i) = Iup\ (\bigsqcup i.\ A\ i) \wedge$
$(\exists j.\ \forall i.\ Y\ (i + j) = Iup\ (A\ i))) \vee (Y = (\lambda i.\ Ibottom))$
apply (*rule disjCI*)
apply (*simp add*: *expand-fun-eq*)
apply (*erule exE*, *rename-tac j*)
apply (*rule-tac x*=$\lambda i.\ THE\ a.\ Iup\ a = Y\ (i + j)$ **in** *exI*)
apply (*simp add*: *up-lemma4*)
apply (*simp add*: *up-lemma6* [*THEN thelubI*])
apply (*rule-tac x*=j **in** *exI*)
apply (*simp add*: *up-lemma3*)
done

lemma *cpo-up*: $chain\ (Y{::}nat \Rightarrow 'a\ u) \Longrightarrow \exists x.\ range\ Y <<|\ x$
apply (*frule up-chain-lemma*, *safe*)
apply (*rule-tac x*=$Iup\ (\bigsqcup i.\ A\ i)$ **in** *exI*)
apply (*erule-tac j*=j **in** *is-lub-range-shift* [*THEN iffD1*, *standard*])
apply (*simp add*: *is-lub-Iup cpo-lubI*)

apply (*rule exI*, *rule lub-const*)
done

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

12.5 Lifted cpo 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*: $\bot$ = *Ibottom*
by (*rule minimal-up* [*THEN UU-I*, *symmetric*])

12.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

lemma *cont-Iup*: *cont Iup*
apply (*rule contI*)
apply (*rule is-lub-Iup*)
apply (*erule cpo-lubI*)
done

continuity for *Ifup*

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

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

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

12.7 Continuous versions of constants

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

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

translations
case l of XCONST up·*x* ⇒ *t* == *CONST fup*·$(\Lambda\ x.\ t)$·*l*
Λ(*XCONST up*·*x*). *t* == *CONST fup*·$(\Lambda\ x.\ t)$

continuous versions of lemmas for $'a_\perp$

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

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

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

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

lemma *not-up-less-UU*: $\neg\ up{\cdot}x \sqsubseteq \perp$
by *simp*

lemma *up-below* [*simp*]: $up{\cdot}x \sqsubseteq up{\cdot}y \longleftrightarrow x \sqsubseteq y$
by (*simp add*: *up-def cont-Iup*)

lemma *upE* [*case-names bottom up*, *cases type*: *u*]:
$[\![p = \perp \Longrightarrow Q;\ \bigwedge x.\ p = up{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*cases p*)
apply (*simp add*: *inst-up-pcpo*)
apply (*simp add*: *up-def cont-Iup*)
done

lemma *up-induct* [*case-names bottom up*, *induct type*: *u*]:
$[\![P\ \perp;\ \bigwedge x.\ P\ (up{\cdot}x)]\!] \Longrightarrow P\ x$
by (*cases x*, *simp-all*)

lifting preserves chain-finiteness

lemma *up-chain-cases*:
chain Y ⟹

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

lemma *compact-up*: *compact x* $\Longrightarrow$ *compact* (*up*·*x*)
apply (*rule compactI2*)
apply (*drule up-chain-cases*, *safe*)
apply (*drule* (*1*) *compactD2*, *simp*)
apply (*erule exE*, *rule-tac x*=*i* + *j* **in** *exI*)
apply *simp*
apply *simp*
done

lemma *compact-upD*: *compact* (*up*·*x*) $\Longrightarrow$ *compact x*
unfolding *compact-def*
by (*drule adm-subst* [*OF cont-Rep-CFun2* [**where** *f*=*up*]], *simp*)

lemma *compact-up-iff* [*simp*]: *compact* (*up*·*x*) = *compact x*
by (*safe elim*!: *compact-up compact-upD*)

instance *u* :: (*chfin*) *chfin*
apply *intro-classes*
apply (*erule compact-imp-max-in-chain*)
apply (*rule-tac p*=$\bigsqcup i.\ Y\ i$ **in** *upE*, *simp-all*)
done

properties of fup

lemma *fup1* [*simp*]: *fup*·*f*·$\bot$ = $\bot$
by (*simp add*: *fup-def cont-Ifup1 cont-Ifup2 inst-up-pcpo cont2cont-LAM*)

lemma *fup2* [*simp*]: *fup*·*f*·(*up*·*x*) = *f*·*x*
by (*simp add*: *up-def fup-def cont-Iup cont-Ifup1 cont-Ifup2 cont2cont-LAM*)

lemma *fup3* [*simp*]: *fup*·*up*·*x* = *x*
by (*cases x*, *simp-all*)

12.8 Map function for lifted cpo

definition
u-map :: ('*a* $\rightarrow$ '*b*) $\rightarrow$ '*a u* $\rightarrow$ '*b u*
where
u-map = (Λ *f*. *fup*·(*up oo f*))

lemma *u-map-strict* [*simp*]: *u-map*·*f*·$\bot$ = $\bot$
unfolding *u-map-def* **by** *simp*

lemma *u-map-up* [*simp*]: *u-map*·*f*·(*up*·*x*) = *up*·(*f*·*x*)
unfolding *u-map-def* **by** *simp*

lemma *u-map-ID*: *u-map·ID = ID*
unfolding *u-map-def* **by** (*simp add*: *expand-cfun-eq eta-cfun*)

lemma *u-map-map*: *u-map·f·(u-map·g·p) = u-map·(Λ x. f·(g·x))·p*
by (*induct p*) *simp-all*

lemma *ep-pair-u-map*: *ep-pair e p ⟹ ep-pair (u-map·e) (u-map·p)*
apply *default*
apply (*case-tac x*, *simp*, *simp add*: *ep-pair.e-inverse*)
apply (*case-tac y*, *simp*, *simp add*: *ep-pair.e-p-below*)
done

lemma *deflation-u-map*: *deflation d ⟹ deflation (u-map·d)*
apply *default*
apply (*case-tac x*, *simp*, *simp add*: *deflation.idem*)
apply (*case-tac x*, *simp*, *simp add*: *deflation.below*)
done

lemma *finite-deflation-u-map*:
  assumes *finite-deflation d* **shows** *finite-deflation (u-map·d)*
proof (*intro finite-deflation.intro finite-deflation-axioms.intro*)
  interpret *d*: *finite-deflation d* **by** *fact*
  have *deflation d* **by** *fact*
  thus *deflation (u-map·d)* **by** (*rule deflation-u-map*)
  have {*x*. *u-map·d·x = x*} ⊆ *insert* ⊥ ((λ*x*. *up·x*) ‘ {*x*. *d·x = x*})
    by (*rule subsetI*, *case-tac x*, *simp-all*)
  thus *finite* {*x*. *u-map·d·x = x*}
    by (*rule finite-subset*, *simp add*: *d.finite-fixes*)
qed

12.9 Lifted cpo is a bifinite domain

instantiation *u* :: (*profinite*) *bifinite*
begin

definition
  approx-up-def:
    approx = (λn. u-map·(approx n))

instance proof
  fix *i* :: *nat* **and** *x* :: *'a u*
  show *chain* (*approx* :: *nat ⇒ 'a u → 'a u*)
    unfolding *approx-up-def* **by** *simp*
  show (⨆ *i*. *approx i·x*) = *x*
    unfolding *approx-up-def*
    by (*induct x*, *simp*, *simp add*: *lub-distribs*)
  show *approx i·(approx i·x) = approx i·x*
    unfolding *approx-up-def*
    by (*induct x*) *simp-all*

show *finite* {*x*::'*a u*. *approx i*·*x* = *x*}
unfolding *approx-up-def*
by (*intro finite-deflation.finite-fixes*
finite-deflation-u-map
finite-deflation-approx)
qed

end

lemma *approx-up* [*simp*]: *approx i*·(*up*·*x*) = *up*·(*approx i*·*x*)
unfolding *approx-up-def* **by** *simp*

end

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

theory *Lift*
imports *Discrete Up Countable*
begin

default-sort *type*

pcpodef '*a lift* = *UNIV* :: '*a discr u set*
by *simp-all*

instance *lift* :: (*finite*) *finite-po*
by (*rule typedef-finite-po* [*OF type-definition-lift*])

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

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

13.1 Lift as a datatype

lemma *lift-induct*: ⟦*P* ⊥; ⋀*x*. *P* (*Def x*)⟧ ⟹ *P y*
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 ⊥::'*a lift Def*
by (*erule lift-induct*) (*simp-all add*: *Def-def Abs-lift-inject lift-def inst-lift-pcpo*)

lemmas *lift-distinct1* = *lift.distinct*(*1*)

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

$\bot$ and *Def*

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

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

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

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

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

method-setup *defined* = ⟪
Scan.succeed (*fn ctxt => SIMPLE-METHOD'*
(*etac* @{*thm lift-definedE*} *THEN' asm-simp-tac* (*simpset-of ctxt*)))
⟫

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

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

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

lemma *Def-below-iff* [*simp*]: $Def\ x \sqsubseteq y \longleftrightarrow Def\ x = y$
by (*induct y*, *simp*, *simp add*: *Def-below-Def*)

13.2 Lift is flat

instance *lift* :: (*type*) *flat*
proof
fix *x y* :: *'a lift*
assume $x \sqsubseteq y$ **thus** $x = \bot \vee x = y$
by (*induct x*) *auto*
qed

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

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

lemma *cont-Rep-CFun-app-app* [*simp*]: ⟦*cont g*; *cont f*⟧ ⟹ *cont*(λ*x*. ((*f x*)·(*g x*)) *s t*)
by (*rule cont-Rep-CFun-app* [*THEN cont2cont-fun*])

13.3 Further operations

definition
flift1 :: (′*a* ⇒ ′*b*::*pcpo*) ⇒ (′*a lift* → ′*b*) (**binder** *FLIFT 10*) **where**
flift1 = (λ*f*. (Λ *x*. *lift-case* ⊥ *f x*))

definition
flift2 :: (′*a* ⇒ ′*b*) ⇒ (′*a lift* → ′*b lift*) **where**
flift2 f = (*FLIFT x*. *Def* (*f x*))

13.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

lemma *cont-lift-case1*: *cont* (λ*f*. *lift-case a f x*)
apply (*induct x*)
apply *simp*
apply *simp*
apply (*rule cont-id* [*THEN cont2cont-fun*])
done

lemma *cont-lift-case2*: *cont* (λ*x*. *lift-case* ⊥ *f x*)
apply (*rule flatdom-strict2cont*)
apply *simp*
done

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

lemma *FLIFT-mono*:
(⋀*x*. *f x* ⊑ *g x*) ⟹ (*FLIFT x*. *f x*) ⊑ (*FLIFT x*. *g x*)
apply (*rule monofunE* [**where** *f*=*flift1*])
apply (*rule cont2mono* [*OF cont-flift1*])
apply (*simp add*: *below-fun-ext*)
done

lemma *cont2cont-flift1* [*simp*, *cont2cont*]:
⟦⋀*y*. *cont* (λ*x*. *f x y*)⟧ ⟹ *cont* (λ*x*. *FLIFT y*. *f x y*)
apply (*rule cont-flift1* [*THEN cont2cont-app3*])
apply *simp*
done

lemma *cont2cont-lift-case* [*simp*]:
⟦⋀*y*. *cont* (λ*x*. *f x y*); *cont g*⟧ ⟹ *cont* (λ*x*. *lift-case UU* (*f x*) (*g x*))
apply (*subgoal-tac cont* (λ*x*. (*FLIFT y*. *f x y*)·(*g x*)))
apply (*simp add*: *flift1-def cont-lift-case2*)
apply *simp*
done

rewrites for *flift1*, *flift2*

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

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

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

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

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

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

13.5 Lifted countable types are bifinite

instantiation *lift* :: (*countable*) *bifinite*
begin

definition
approx-lift-def:
approx = (λ*n*. *FLIFT x*. *if to-nat x* < *n then Def x else* ⊥)

instance proof
fix *x* :: *'a lift*
show *chain* (*approx* :: *nat* ⇒ *'a lift* → *'a lift*)
unfolding *approx-lift-def*
by (*rule chainI*, *simp add*: *FLIFT-mono*)
next
fix *x* :: *'a lift*
show (⨆ *i*. *approx i*·*x*) = *x*
unfolding *approx-lift-def*
apply (*cases x*, *simp*)
apply (*rule thelubI*)
apply (*rule is-lubI*)
apply (*rule ub-rangeI*, *simp*)
apply (*drule ub-rangeD*)

```
    apply (erule rev-below-trans)
    apply simp
    apply (rule lessI)
    done
next
  fix i :: nat and x :: 'a lift
  show approx i·(approx i·x) = approx i·x
    unfolding approx-lift-def
    by (cases x, simp, simp)
next
  fix i :: nat
  show finite {x::'a lift. approx i·x = x}
  proof (rule finite-subset)
    let ?S = insert (⊥::'a lift) (Def ` to-nat -` {..<i})
    show {x::'a lift. approx i·x = x} ⊆ ?S
      unfolding approx-lift-def
      by (rule subsetI, case-tac x, simp, simp split: split-if-asm)
    show finite ?S
      by (simp add: finite-vimageI)
  qed
qed

end

end
```

14 Tr: The type of lifted booleans

```
theory Tr
imports Lift
begin
```

14.1 Type definition and constructors

```
types
  tr = bool lift

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

definition
  TT :: tr where
  TT = Def True

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

Exhaustion and Elimination for type *tr*

lemma *Exh-tr*: $t = \bot \vee t = TT \vee t = FF$
unfolding *FF-def TT-def* **by** (*induct t*) *auto*

lemma *trE* [*case-names bottom TT FF*]:
$[\![p = \bot \Longrightarrow Q;\ p = TT \Longrightarrow Q;\ p = FF \Longrightarrow Q]\!] \Longrightarrow Q$
unfolding *FF-def TT-def* **by** (*induct p*) *auto*

lemma *tr-induct* [*case-names bottom TT FF*]:
$[\![P\ \bot;\ P\ TT;\ P\ FF]\!] \Longrightarrow P\ x$
by (*cases x rule*: *trE*) *simp-all*

distinctness for type *tr*

lemma *dist-below-tr* [*simp*]:
$\neg\ TT \sqsubseteq \bot\ \neg\ FF \sqsubseteq \bot\ \neg\ TT \sqsubseteq FF\ \neg\ FF \sqsubseteq TT$
unfolding *TT-def FF-def* **by** *simp-all*

lemma *dist-eq-tr* [*simp*]:
$TT \neq \bot\ FF \neq \bot\ TT \neq FF\ \bot \neq TT\ \bot \neq FF\ FF \neq TT$
unfolding *TT-def FF-def* **by** *simp-all*

lemma *TT-below-iff* [*simp*]: $TT \sqsubseteq x \longleftrightarrow x = TT$
by (*induct x rule*: *tr-induct*) *simp-all*

lemma *FF-below-iff* [*simp*]: $FF \sqsubseteq x \longleftrightarrow x = FF$
by (*induct x rule*: *tr-induct*) *simp-all*

lemma *not-below-TT-iff* [*simp*]: $\neg\ (x \sqsubseteq TT) \longleftrightarrow x = FF$
by (*induct x rule*: *tr-induct*) *simp-all*

lemma *not-below-FF-iff* [*simp*]: $\neg\ (x \sqsubseteq FF) \longleftrightarrow x = TT$
by (*induct x rule*: *tr-induct*) *simp-all*

14.2 Case analysis

default-sort *pcpo*

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

translations
Λ (*XCONST TT*). *t* == *CONST trifte*·*t*·$\bot$
Λ (*XCONST FF*). *t* == *CONST trifte*·$\bot$·*t*

lemma *ifte-thms* [*simp*]:

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

14.3 Boolean connectives

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

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

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

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

tactic for tr-thms with case split

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

lemmas about andalso, orelse, neg and if

lemma *andalso-thms* [*simp*]:
(*TT andalso y*) = *y*
(*FF andalso y*) = *FF*
(⊥ *andalso y*) = ⊥
(*y andalso TT*) = *y*
(*y andalso y*) = *y*
apply (*unfold andalso-def*, *simp-all*)
apply (*cases y rule*: *trE*, *simp-all*)
apply (*cases y rule*: *trE*, *simp-all*)
done

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

(*y orelse y*) = *y*
apply (*unfold orelse-def*, *simp-all*)
apply (*cases y rule*: *trE*, *simp-all*)
apply (*cases y rule*: *trE*, *simp-all*)
done

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

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

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

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

14.4 Rewriting of HOLCF operations to HOL functions

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

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

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

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

14.5 Compactness

lemma *compact-TT*: *compact TT*
by (*rule compact-chfin*)

lemma *compact-FF*: *compact FF*
by (*rule compact-chfin*)

end

15 Ssum: The type of strict sums

theory *Ssum*
imports *Tr*
begin

default-sort *pcpo*

15.1 Definition of strict sum type

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

instance *ssum* :: ({*finite-po*,*pcpo*}, {*finite-po*,*pcpo*}) *finite-po*
by (*rule typedef-finite-po* [*OF type-definition-Ssum*])

instance *ssum* :: ({*chfin*,*pcpo*}, {*chfin*,*pcpo*}) *chfin*
by (*rule typedef-chfin* [*OF type-definition-Ssum below-Ssum-def*])

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

15.2 Definitions of constructors

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

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

lemma *sinl-Ssum*: $(strictify{\cdot}(\Lambda\ \text{-}.\ TT){\cdot}a,\ a,\ \bot) \in Ssum$
by (*simp add*: *Ssum-def strictify-conv-if*)

lemma *sinr-Ssum*: $(strictify{\cdot}(\Lambda\ \text{-}.\ FF){\cdot}b,\ \bot,\ b) \in Ssum$
by (*simp add*: *Ssum-def strictify-conv-if*)

lemma *sinl-Abs-Ssum*: $sinl{\cdot}a = Abs\text{-}Ssum\ (strictify{\cdot}(\Lambda\ \text{-}.\ TT){\cdot}a,\ a,\ \bot)$
by (*unfold sinl-def*, *simp add*: *cont-Abs-Ssum sinl-Ssum*)

lemma *sinr-Abs-Ssum*: $sinr{\cdot}b = Abs\text{-}Ssum\ (strictify{\cdot}(\Lambda\ \text{-}.\ FF){\cdot}b,\ \bot,\ b)$
by (*unfold sinr-def*, *simp add*: *cont-Abs-Ssum sinr-Ssum*)

lemma *Rep-Ssum-sinl*: $Rep\text{-}Ssum\ (sinl{\cdot}a) = (strictify{\cdot}(\Lambda\ \text{-}.\ TT){\cdot}a,\ a,\ \bot)$
by (*simp add*: *sinl-Abs-Ssum Abs-Ssum-inverse sinl-Ssum*)

lemma *Rep-Ssum-sinr*: $Rep\text{-}Ssum\ (sinr{\cdot}b) = (strictify{\cdot}(\Lambda\ \text{-}.\ FF){\cdot}b,\ \bot,\ b)$
by (*simp add*: *sinr-Abs-Ssum Abs-Ssum-inverse sinr-Ssum*)

15.3 Properties of *sinl* and *sinr*

Ordering

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

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

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

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

Equality

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

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

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

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

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

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

Strictness

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

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

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

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

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*

Compactness

lemma *compact-sinl*: *compact* $x \Longrightarrow$ *compact* $(sinl{\cdot}x)$
by (*rule compact-Ssum*, *simp add*: *Rep-Ssum-sinl strictify-conv-if*)

lemma *compact-sinr*: *compact* $x \Longrightarrow$ *compact* $(sinr{\cdot}x)$
by (*rule compact-Ssum*, *simp add*: *Rep-Ssum-sinr strictify-conv-if*)

lemma *compact-sinlD*: *compact* $(sinl{\cdot}x) \Longrightarrow$ *compact* x
unfolding *compact-def*
by (*drule adm-subst* [*OF cont-Rep-CFun2* [**where** *f*=*sinl*]], *simp*)

lemma *compact-sinrD*: *compact* $(sinr{\cdot}x) \Longrightarrow$ *compact* x
unfolding *compact-def*
by (*drule adm-subst* [*OF cont-Rep-CFun2* [**where** *f*=*sinr*]], *simp*)

lemma *compact-sinl-iff* [*simp*]: *compact* $(sinl{\cdot}x) =$ *compact* x
by (*safe elim*!: *compact-sinl compact-sinlD*)

lemma *compact-sinr-iff* [*simp*]: *compact* (*sinr*·*x*) = *compact x*
by (*safe elim*!: *compact-sinr compact-sinrD*)

15.4 Case analysis

lemma *Exh-Ssum*:
  $z = \bot \vee (\exists\, a.\ z = sinl{\cdot}a \wedge a \neq \bot) \vee (\exists\, b.\ z = sinr{\cdot}b \wedge b \neq \bot)$
apply (*induct z rule*: *Abs-Ssum-induct*)
apply (*case-tac y*, *rename-tac t a b*)
apply (*case-tac t rule*: *trE*)
apply (*rule disjI1*)
apply (*simp add*: *Ssum-def Abs-Ssum-strict*)
apply (*rule disjI2*, *rule disjI1*, *rule-tac x*=*a* **in** *exI*)
apply (*simp add*: *sinl-Abs-Ssum Ssum-def*)
apply (*rule disjI2*, *rule disjI2*, *rule-tac x*=*b* **in** *exI*)
apply (*simp add*: *sinr-Abs-Ssum Ssum-def*)
done

lemma *ssumE* [*case-names bottom sinl sinr*, *cases type*: *ssum*]:
  $[\![p = \bot \Longrightarrow Q;$
  $\bigwedge x.\ [\![p = sinl{\cdot}x;\ x \neq \bot]\!] \Longrightarrow Q;$
  $\bigwedge y.\ [\![p = sinr{\cdot}y;\ y \neq \bot]\!] \Longrightarrow Q]\!] \Longrightarrow Q$
using *Exh-Ssum* [*of p*] **by** *auto*

lemma *ssum-induct* [*case-names bottom sinl sinr*, *induct type*: *ssum*]:
  $[\![P\ \bot;$
  $\bigwedge x.\ x \neq \bot \Longrightarrow P\ (sinl{\cdot}x);$
  $\bigwedge y.\ y \neq \bot \Longrightarrow P\ (sinr{\cdot}y)]\!] \Longrightarrow P\ x$
by (*cases x*, *simp-all*)

lemma *ssumE2* [*case-names sinl sinr*]:
  $[\![\bigwedge x.\ p = sinl{\cdot}x \Longrightarrow Q;\ \bigwedge y.\ p = sinr{\cdot}y \Longrightarrow Q]\!] \Longrightarrow Q$
by (*cases p*, *simp only*: *sinl-strict* [*symmetric*], *simp*, *simp*)

lemma *below-sinlD*: $p \sqsubseteq sinl{\cdot}x \Longrightarrow \exists\, y.\ p = sinl{\cdot}y \wedge y \sqsubseteq x$
by (*cases p*, *rule-tac x*=$\bot$ **in** *exI*, *simp-all*)

lemma *below-sinrD*: $p \sqsubseteq sinr{\cdot}x \Longrightarrow \exists\, y.\ p = sinr{\cdot}y \wedge y \sqsubseteq x$
by (*cases p*, *rule-tac x*=$\bot$ **in** *exI*, *simp-all*)

15.5 Case analysis combinator

definition
  sscase :: $('a \to 'c) \to ('b \to 'c) \to ('a \mathbin{++} 'b) \to 'c$ **where**
  sscase = (Λ *f g s*. (λ(*t*, *x*, *y*). *If t then f*·*x else g*·*y fi*) (*Rep-Ssum s*))

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

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

lemma *beta-sscase*:
$sscase{\cdot}f{\cdot}g{\cdot}s = (\lambda(t,\ x,\ y).\ If\ t\ then\ f{\cdot}x\ else\ g{\cdot}y\ fi)\ (Rep\text{-}Ssum\ s)$
unfolding *sscase-def* **by** (*simp add*: *cont-Rep-Ssum* [*THEN cont-compose*])

lemma *sscase1* [*simp*]: $sscase{\cdot}f{\cdot}g{\cdot}\bot = \bot$
unfolding *beta-sscase* **by** (*simp add*: *Rep-Ssum-strict*)

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

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

lemma *sscase4* [*simp*]: $sscase{\cdot}sinl{\cdot}sinr{\cdot}z = z$
by (*cases z*, *simp-all*)

15.6 Strict sum preserves flatness

instance *ssum* :: (*flat*, *flat*) *flat*
apply (*intro-classes*, *clarify*)
apply (*case-tac x*, *simp*)
apply (*case-tac y*, *simp-all add*: *flat-below-iff*)
apply (*case-tac y*, *simp-all add*: *flat-below-iff*)
done

15.7 Map function for strict sums

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

lemma *ssum-map-strict* [*simp*]: $ssum\text{-}map{\cdot}f{\cdot}g{\cdot}\bot = \bot$
unfolding *ssum-map-def* **by** *simp*

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

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

lemma *ssum-map-sinl'*: $f{\cdot}\bot = \bot \Longrightarrow ssum\text{-}map{\cdot}f{\cdot}g{\cdot}(sinl{\cdot}x) = sinl{\cdot}(f{\cdot}x)$
by (*cases* $x = \bot$) *simp-all*

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

by (*cases* $x = \bot$) *simp-all*

lemma *ssum-map-ID*: *ssum-map·ID·ID = ID*
unfolding *ssum-map-def* **by** (*simp add*: *expand-cfun-eq eta-cfun*)

lemma *ssum-map-map*:
$[\![f1\cdot\bot = \bot;\ g1\cdot\bot = \bot]\!] \Longrightarrow$
ssum-map·f1·g1·(*ssum-map·f2·g2·p*) =
ssum-map·(Λ *x*. *f1·*(*f2·x*))*·*(Λ *x*. *g1·*(*g2·x*))*·p*
apply (*induct p*, *simp*)
apply (*case-tac f2·x = ⊥*, *simp*, *simp*)
apply (*case-tac g2·y = ⊥*, *simp*, *simp*)
done

lemma *ep-pair-ssum-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*ssum-map·e1·e2*) (*ssum-map·p1·p2*)
proof
interpret *e1p1*: *pcpo-ep-pair e1 p1* **unfolding** *pcpo-ep-pair-def* **by** *fact*
interpret *e2p2*: *pcpo-ep-pair e2 p2* **unfolding** *pcpo-ep-pair-def* **by** *fact*
fix *x* **show** *ssum-map·p1·p2·*(*ssum-map·e1·e2·x*) = *x*
by (*induct x*) *simp-all*
fix *y* **show** *ssum-map·e1·e2·*(*ssum-map·p1·p2·y*) ⊑ *y*
apply (*induct y*, *simp*)
apply (*case-tac p1·x = ⊥*, *simp*, *simp add*: *e1p1.e-p-below*)
apply (*case-tac p2·y = ⊥*, *simp*, *simp add*: *e2p2.e-p-below*)
done
qed

lemma *deflation-ssum-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*ssum-map·d1·d2*)
proof
interpret *d1*: *deflation d1* **by** *fact*
interpret *d2*: *deflation d2* **by** *fact*
fix *x*
show *ssum-map·d1·d2·*(*ssum-map·d1·d2·x*) = *ssum-map·d1·d2·x*
apply (*induct x*, *simp*)
apply (*case-tac d1·x = ⊥*, *simp*, *simp add*: *d1.idem*)
apply (*case-tac d2·y = ⊥*, *simp*, *simp add*: *d2.idem*)
done
show *ssum-map·d1·d2·x* ⊑ *x*
apply (*induct x*, *simp*)
apply (*case-tac d1·x = ⊥*, *simp*, *simp add*: *d1.below*)
apply (*case-tac d2·y = ⊥*, *simp*, *simp add*: *d2.below*)
done
qed

lemma *finite-deflation-ssum-map*:

assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*ssum-map*·*d1*·*d2*)
proof (*intro finite-deflation.intro finite-deflation-axioms.intro*)
interpret *d1*: *finite-deflation d1* **by** *fact*
interpret *d2*: *finite-deflation d2* **by** *fact*
have *deflation d1* **and** *deflation d2* **by** *fact+*
thus *deflation* (*ssum-map*·*d1*·*d2*) **by** (*rule deflation-ssum-map*)
have $\{x.\ \textit{ssum-map}{\cdot}d1{\cdot}d2{\cdot}x = x\} \subseteq$
$(\lambda x.\ \textit{sinl}{\cdot}x)\ `\ \{x.\ d1{\cdot}x = x\} \cup$
$(\lambda x.\ \textit{sinr}{\cdot}x)\ `\ \{x.\ d2{\cdot}x = x\} \cup \{\bot\}$
by (*rule subsetI*, *case-tac x*, *simp-all*)
thus *finite* $\{x.\ \textit{ssum-map}{\cdot}d1{\cdot}d2{\cdot}x = x\}$
by (*rule finite-subset*, *simp add*: *d1.finite-fixes d2.finite-fixes*)
qed

15.8 Strict sum is a bifinite domain

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

definition
approx-ssum-def:
approx = $(\lambda n.\ \textit{ssum-map}{\cdot}(\textit{approx}\ n){\cdot}(\textit{approx}\ n))$

lemma *approx-sinl* [*simp*]: *approx* $i{\cdot}(\textit{sinl}{\cdot}x) = \textit{sinl}{\cdot}(\textit{approx}\ i{\cdot}x)$
unfolding *approx-ssum-def* **by** (*cases* $x = \bot$) *simp-all*

lemma *approx-sinr* [*simp*]: *approx* $i{\cdot}(\textit{sinr}{\cdot}x) = \textit{sinr}{\cdot}(\textit{approx}\ i{\cdot}x)$
unfolding *approx-ssum-def* **by** (*cases* $x = \bot$) *simp-all*

instance proof
fix i :: *nat* **and** x :: $'a \oplus 'b$
show *chain* (*approx* :: *nat* ⇒ $'a \oplus 'b \to 'a \oplus 'b$)
unfolding *approx-ssum-def* **by** *simp*
show $(\bigsqcup i.\ \textit{approx}\ i{\cdot}x) = x$
unfolding *approx-ssum-def*
by (*cases x*, *simp-all add*: *lub-distribs*)
show *approx* $i{\cdot}(\textit{approx}\ i{\cdot}x) = \textit{approx}\ i{\cdot}x$
by (*cases x*, *simp add*: *approx-ssum-def*, *simp*, *simp*)
show *finite* $\{x::'a \oplus 'b.\ \textit{approx}\ i{\cdot}x = x\}$
unfolding *approx-ssum-def*
by (*intro finite-deflation.finite-fixes*
finite-deflation-ssum-map
finite-deflation-approx)
qed

end

end

16 Sprod: The type of strict products

theory *Sprod*
imports *Bifinite*
begin

default-sort *pcpo*

16.1 Definition of strict product type

pcpodef (*Sprod*) (*'a*, *'b*) *sprod* (**infixr** $**$ *20*) =
$\{p{::}'a \times 'b.\ p = \bot \vee (fst\ p \neq \bot \wedge snd\ p \neq \bot)\}$
by *simp-all*

instance *sprod* :: ({*finite-po*,*pcpo*}, {*finite-po*,*pcpo*}) *finite-po*
by (*rule typedef-finite-po* [*OF type-definition-Sprod*])

instance *sprod* :: ({*chfin*,*pcpo*}, {*chfin*,*pcpo*}) *chfin*
by (*rule typedef-chfin* [*OF type-definition-Sprod below-Sprod-def*])

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

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

16.2 Definitions of constants

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

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

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

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

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

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

16.3 Case analysis

lemma *Rep-Sprod-spair*:
Rep-Sprod (:*a*, *b*:) = (*strictify*·(Λ *b*. *a*)·*b*, *strictify*·(Λ *a*. *b*)·*a*)
unfolding *spair-def*
by (*simp add*: *cont-Abs-Sprod Abs-Sprod-inverse spair-lemma*)

lemmas *Rep-Sprod-simps* =
Rep-Sprod-inject [*symmetric*] *below-Sprod-def*
Rep-Sprod-strict Rep-Sprod-spair

lemma *Exh-Sprod*:
z = ⊥ ∨ (∃ *a b*. *z* = (:*a*, *b*:) ∧ *a* ≠ ⊥ ∧ *b* ≠ ⊥)
apply (*insert Rep-Sprod* [*of z*])
apply (*simp add*: *Rep-Sprod-simps Pair-fst-snd-eq*)
apply (*simp add*: *Sprod-def*)
apply (*erule disjE*, *simp*)
apply (*simp add*: *strictify-conv-if*)
apply *fast*
done

lemma *sprodE* [*case-names bottom spair*, *cases type*: *sprod*]:
⟦*p* = ⊥ ⟹ *Q*; ⋀*x y*. ⟦*p* = (:*x*, *y*:); *x* ≠ ⊥; *y* ≠ ⊥⟧ ⟹ *Q*⟧ ⟹ *Q*
using *Exh-Sprod* [*of p*] **by** *auto*

lemma *sprod-induct* [*case-names bottom spair*, *induct type*: *sprod*]:
⟦*P* ⊥; ⋀*x y*. ⟦*x* ≠ ⊥; *y* ≠ ⊥⟧ ⟹ *P* (:*x*, *y*:)⟧ ⟹ *P x*
by (*cases x*, *simp-all*)

16.4 Properties of *spair*

lemma *spair-strict1* [*simp*]: (:⊥, *y*:) = ⊥
by (*simp add*: *Rep-Sprod-simps strictify-conv-if*)

lemma *spair-strict2* [*simp*]: (:*x*, ⊥:) = ⊥
by (*simp add*: *Rep-Sprod-simps strictify-conv-if*)

lemma *spair-strict-iff* [*simp*]: ((:*x*, *y*:) = ⊥) = (*x* = ⊥ ∨ *y* = ⊥)
by (*simp add*: *Rep-Sprod-simps strictify-conv-if*)

lemma *spair-below-iff*:
$((:a,\ b:) \sqsubseteq (:c,\ d:)) = (a = \bot \vee b = \bot \vee (a \sqsubseteq c \wedge b \sqsubseteq d))$
by (*simp add*: *Rep-Sprod-simps strictify-conv-if*)

lemma *spair-eq-iff*:
$((:a,\ b:) = (:c,\ d:)) =$
$(a = c \wedge b = d \vee (a = \bot \vee b = \bot) \wedge (c = \bot \vee d = \bot))$
by (*simp add*: *Rep-Sprod-simps strictify-conv-if*)

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

lemma *spair-strict-rev*: $(:x,\ y:) \neq \bot \Longrightarrow x \neq \bot \wedge y \neq \bot$
by *simp*

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

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

lemma *spair-eq*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow ((:x,\ y:) = (:a,\ b:)) = (x = a \wedge y = b)$
by (*simp add*: *spair-eq-iff*)

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

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

lemma *sprodE2*: $(\bigwedge x\ y.\ p = (:x,\ y:) \Longrightarrow Q) \Longrightarrow Q$
by (*cases p*, *simp only*: *inst-sprod-pcpo2*, *simp*)

16.5 Properties of *sfst* and *ssnd*

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

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

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

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

lemma *sfst-defined-iff* [*simp*]: $(sfst{\cdot}p = \bot) = (p = \bot)$
by (*cases p*, *simp-all*)

lemma *ssnd-defined-iff* [*simp*]: $(ssnd{\cdot}p = \bot) = (p = \bot)$
by (*cases p*, *simp-all*)

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

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

lemma *surjective-pairing-Sprod2*: $(:sfst{\cdot}p,\ ssnd{\cdot}p:) = p$
by (*cases p*, *simp-all*)

lemma *below-sprod*: $x \sqsubseteq y = (sfst{\cdot}x \sqsubseteq sfst{\cdot}y \wedge ssnd{\cdot}x \sqsubseteq ssnd{\cdot}y)$
apply (*simp add*: *below-Sprod-def sfst-def ssnd-def cont-Rep-Sprod*)
apply (*simp only*: *below-prod-def*)
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 below-sprod*)

lemma *spair-below*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) = (x \sqsubseteq a \wedge y \sqsubseteq b)$
apply (*cases a* $= \bot$, *simp*)
apply (*cases b* $= \bot$, *simp*)
apply (*simp add*: *below-sprod*)
done

lemma *sfst-below-iff*: $sfst{\cdot}x \sqsubseteq y = x \sqsubseteq (:y,\ ssnd{\cdot}x:)$
apply (*cases x* $= \bot$, *simp*, *cases y* $= \bot$, *simp*)
apply (*simp add*: *below-sprod*)
done

lemma *ssnd-below-iff*: $ssnd{\cdot}x \sqsubseteq y = x \sqsubseteq (:sfst{\cdot}x,\ y:)$
apply (*cases x* $= \bot$, *simp*, *cases y* $= \bot$, *simp*)
apply (*simp add*: *below-sprod*)
done

16.6 Compactness

lemma *compact-sfst*: *compact x* $\Longrightarrow$ *compact* $(sfst{\cdot}x)$
by (*rule compactI*, *simp add*: *sfst-below-iff*)

lemma *compact-ssnd*: *compact x* $\Longrightarrow$ *compact* $(ssnd{\cdot}x)$
by (*rule compactI*, *simp add*: *ssnd-below-iff*)

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

by (*rule compact-Sprod*, *simp add*: *Rep-Sprod-spair strictify-conv-if*)

lemma *compact-spair-iff*:
  compact (:*x*, *y*:) = (*x* = ⊥ ∨ *y* = ⊥ ∨ (*compact x* ∧ *compact y*))
apply (*safe elim*!: *compact-spair*)
apply (*drule compact-sfst*, *simp*)
apply (*drule compact-ssnd*, *simp*)
apply *simp*
apply *simp*
done

16.7 Properties of *ssplit*

lemma *ssplit1* [*simp*]: *ssplit*·*f*·⊥ = ⊥
by (*simp add*: *ssplit-def*)

lemma *ssplit2* [*simp*]: ⟦*x* ≠ ⊥; *y* ≠ ⊥⟧ ⟹ *ssplit*·*f*·(:*x*, *y*:) = *f*·*x*·*y*
by (*simp add*: *ssplit-def*)

lemma *ssplit3* [*simp*]: *ssplit*·*spair*·*z* = *z*
by (*cases z*, *simp-all*)

16.8 Strict product preserves flatness

instance *sprod* :: (*flat*, *flat*) *flat*
proof
  fix *x y* :: *'a* ⊗ *'b*
  assume *x* ⊑ *y* **thus** *x* = ⊥ ∨ *x* = *y*
    apply (*induct x*, *simp*)
    apply (*induct y*, *simp*)
    apply (*simp add*: *spair-below-iff flat-below-iff*)
    done
qed

16.9 Map function for strict products

definition
  sprod-map :: (*'a* → *'b*) → (*'c* → *'d*) → *'a* ⊗ *'c* → *'b* ⊗ *'d*
where
  sprod-map = (Λ *f g*. *ssplit*·(Λ *x y*. (:*f*·*x*, *g*·*y*:)))

lemma *sprod-map-strict* [*simp*]: *sprod-map*·*a*·*b*·⊥ = ⊥
unfolding *sprod-map-def* **by** *simp*

lemma *sprod-map-spair* [*simp*]:
  x ≠ ⊥ ⟹ *y* ≠ ⊥ ⟹ *sprod-map*·*f*·*g*·(:*x*, *y*:) = (:*f*·*x*, *g*·*y*:)
by (*simp add*: *sprod-map-def*)

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

by (*cases* $x = \bot \vee y = \bot$) *auto*

lemma *sprod-map-ID*: *sprod-map·ID·ID = ID*
unfolding *sprod-map-def* **by** (*simp add*: *expand-cfun-eq eta-cfun*)

lemma *sprod-map-map*:
$[\![$*f1·⊥ = ⊥*; *g1·⊥ = ⊥*$]\!] \Longrightarrow$
sprod-map·f1·g1·(*sprod-map·f2·g2·p*) =
sprod-map·(Λ *x*. *f1·*(*f2·x*))·(Λ *x*. *g1·*(*g2·x*))·*p*
apply (*induct p*, *simp*)
apply (*case-tac f2·x = ⊥*, *simp*)
apply (*case-tac g2·y = ⊥*, *simp*)
apply *simp*
done

lemma *ep-pair-sprod-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*sprod-map·e1·e2*) (*sprod-map·p1·p2*)
proof
interpret *e1p1*: *pcpo-ep-pair e1 p1* **unfolding** *pcpo-ep-pair-def* **by** *fact*
interpret *e2p2*: *pcpo-ep-pair e2 p2* **unfolding** *pcpo-ep-pair-def* **by** *fact*
fix *x* **show** *sprod-map·p1·p2·*(*sprod-map·e1·e2·x*) = *x*
by (*induct x*) *simp-all*
fix *y* **show** *sprod-map·e1·e2·*(*sprod-map·p1·p2·y*) ⊑ *y*
apply (*induct y*, *simp*)
apply (*case-tac p1·x = ⊥*, *simp*, *case-tac p2·y = ⊥*, *simp*)
apply (*simp add*: *monofun-cfun e1p1.e-p-below e2p2.e-p-below*)
done
qed

lemma *deflation-sprod-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*sprod-map·d1·d2*)
proof
interpret *d1*: *deflation d1* **by** *fact*
interpret *d2*: *deflation d2* **by** *fact*
fix *x*
show *sprod-map·d1·d2·*(*sprod-map·d1·d2·x*) = *sprod-map·d1·d2·x*
apply (*induct x*, *simp*)
apply (*case-tac d1·x = ⊥*, *simp*, *case-tac d2·y = ⊥*, *simp*)
apply (*simp add*: *d1.idem d2.idem*)
done
show *sprod-map·d1·d2·x* ⊑ *x*
apply (*induct x*, *simp*)
apply (*simp add*: *monofun-cfun d1.below d2.below*)
done
qed

lemma *finite-deflation-sprod-map*:

assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*sprod-map*·*d1*·*d2*)
proof (*intro finite-deflation.intro finite-deflation-axioms.intro*)
interpret *d1*: *finite-deflation d1* **by** *fact*
interpret *d2*: *finite-deflation d2* **by** *fact*
have *deflation d1* **and** *deflation d2* **by** *fact+*
thus *deflation* (*sprod-map*·*d1*·*d2*) **by** (*rule deflation-sprod-map*)
have {*x*. *sprod-map*·*d1*·*d2*·*x* = *x*} ⊆ *insert* ⊥
((λ(*x*, *y*). (:*x*, *y*:)) ‘ ({*x*. *d1*·*x* = *x*} × {*y*. *d2*·*y* = *y*}))
by (*rule subsetI*, *case-tac x*, *auto simp add*: *spair-eq-iff*)
thus *finite* {*x*. *sprod-map*·*d1*·*d2*·*x* = *x*}
by (*rule finite-subset*, *simp add*: *d1.finite-fixes d2.finite-fixes*)
qed

16.10 Strict product is a bifinite domain

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

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

instance proof
fix *i* :: *nat* **and** *x* :: *'a* ⊗ *'b*
show *chain* (*approx* :: *nat* ⇒ *'a* ⊗ *'b* → *'a* ⊗ *'b*)
unfolding *approx-sprod-def* **by** *simp*
show (⨆ *i*. *approx i*·*x*) = *x*
unfolding *approx-sprod-def sprod-map-def*
by (*simp add*: *lub-distribs eta-cfun*)
show *approx i*·(*approx i*·*x*) = *approx i*·*x*
unfolding *approx-sprod-def sprod-map-def*
by (*simp add*: *ssplit-def strictify-conv-if*)
show *finite* {*x*::*'a* ⊗ *'b*. *approx i*·*x* = *x*}
unfolding *approx-sprod-def*
by (*intro finite-deflation.finite-fixes*
finite-deflation-sprod-map
finite-deflation-approx)
qed

end

lemma *approx-spair* [*simp*]:
approx i·(:*x*, *y*:) = (:*approx i*·*x*, *approx i*·*y*:)
unfolding *approx-sprod-def sprod-map-def*
by (*simp add*: *ssplit-def strictify-conv-if*)

end

17 One: The unit domain

theory *One*
imports *Lift*
begin

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

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

Exhaustion and Elimination for type *one*

lemma *Exh-one*: $t = \bot \vee t = ONE$
unfolding *ONE-def* **by** (*induct t*) *simp-all*

lemma *oneE* [*case-names bottom ONE*]: $[\![p = \bot \Longrightarrow Q; p = ONE \Longrightarrow Q]\!] \Longrightarrow Q$
unfolding *ONE-def* **by** (*induct p*) *simp-all*

lemma *one-induct* [*case-names bottom ONE*]: $[\![P\ \bot; P\ ONE]\!] \Longrightarrow P\ x$
by (*cases x rule*: *oneE*) *simp-all*

lemma *dist-below-one* [*simp*]: $\neg\ ONE \sqsubseteq \bot$
unfolding *ONE-def* **by** *simp*

lemma *below-ONE* [*simp*]: $x \sqsubseteq ONE$
by (*induct x rule*: *one-induct*) *simp-all*

lemma *ONE-below-iff* [*simp*]: $ONE \sqsubseteq x \longleftrightarrow x = ONE$
by (*induct x rule*: *one-induct*) *simp-all*

lemma *ONE-defined* [*simp*]: $ONE \neq \bot$
unfolding *ONE-def* **by** *simp*

lemma *one-neq-iffs* [*simp*]:
$x \neq ONE \longleftrightarrow x = \bot$
$ONE \neq x \longleftrightarrow x = \bot$
$x \neq \bot \longleftrightarrow x = ONE$
$\bot \neq x \longleftrightarrow x = ONE$
by (*induct x rule*: *one-induct*) *simp-all*

lemma *compact-ONE*: *compact ONE*
by (*rule compact-chfin*)

Case analysis function for type *one*

definition

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

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

lemma *one-when1* [*simp*]: (*case* ⊥ *of ONE* ⇒ *t*) = ⊥
by (*simp add*: *one-when-def*)

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

lemma *one-when3* [*simp*]: (*case x of ONE* ⇒ *ONE*) = *x*
by (*induct x rule*: *one-induct*) *simp-all*

end

18 Cprod: The cpo of cartesian products

theory *Cprod*
imports *Bifinite*
begin

default-sort *cpo*

18.1 Continuous case function for unit type

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

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

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

18.2 Continuous version of split function

definition
csplit :: (*'a* → *'b* → *'c*) → (*'a* ∗ *'b*) → *'c* **where**
csplit = (Λ *f p*. *f*·(*fst p*)·(*snd p*))

translations
Λ(*CONST Pair x y*). *t* == *CONST csplit*·(Λ *x y*. *t*)

18.3 Convert all lemmas to the continuous versions

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

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

end

19 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun*
begin

default-sort *pcpo*

19.1 Iteration

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

Derive inductive properties of iterate from primitive recursion

lemma *iterate-0* [*simp*]: $iterate\ 0{\cdot}F{\cdot}x = x$
by *simp*

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

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

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

lemma *iterate-iterate*:
  $iterate\ m{\cdot}F{\cdot}(iterate\ n{\cdot}F{\cdot}x) = iterate\ (m + n){\cdot}F{\cdot}x$
by (*induct m*) *simp-all*

The sequence of function iterations is a chain.

lemma *chain-iterate* [*simp*]: $chain\ (\lambda i.\ iterate\ i{\cdot}F{\cdot}\bot)$
by (*rule chainI*, *unfold iterate-Suc2*, *rule monofun-cfun-arg*, *rule minimal*)

19.2 Least fixed point operator

definition
  fix :: $('a \rightarrow 'a) \rightarrow 'a$ **where**

$fix = (\Lambda\ F.\ \bigsqcup i.\ iterate\ i{\cdot}F{\cdot}\bot)$

Binder syntax for *fix*

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

notation (*xsymbols*)
$fix\text{-}syn$ (**binder** μ *10*)

Properties of *fix*

direct connection between *fix* and iteration

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

lemma *iterate-below-fix*: $iterate\ n{\cdot}f{\cdot}\bot \sqsubseteq fix{\cdot}f$
unfolding *fix-def2*
using *chain-iterate* **by** (*rule is-ub-thelub*)

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

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

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

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

lemma *fix-eqI*:
  assumes *fixed*: $F{\cdot}x = x$ **and** *least*: $\bigwedge z.\ F{\cdot}z = z \Longrightarrow x \sqsubseteq z$
  shows $fix{\cdot}F = x$
apply (*rule below-antisym*)
apply (*rule fix-least* [*OF fixed*])
apply (*rule least* [*OF fix-eq* [*symmetric*]])
done

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

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

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

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

strictness of *fix*

lemma *fix-defined-iff*: $(fix{\cdot}F = \bot) = (F{\cdot}\bot = \bot)$
apply (*rule iffI*)
apply (*erule subst*)
apply (*rule fix-eq* [*symmetric*])
apply (*erule fix-least* [*THEN UU-I*])
done

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

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

fix applied to identity and constant functions

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

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

19.3 Fixed point induction

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

unfolding *fix-def2*
apply (*erule admD*)
apply (*rule chain-iterate*)
apply (*rule nat-induct*, *simp-all*)
done

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

lemma *fix-ind2*:
  assumes *adm*: *adm P*
  assumes *0*: $P\ \bot$ **and** *1*: $P\ (F{\cdot}\bot)$
  assumes *step*: $\bigwedge x.$ ⟦$P\ x$; $P\ (F{\cdot}x)$⟧ $\Longrightarrow P\ (F{\cdot}(F{\cdot}x))$
  shows $P\ (fix{\cdot}F)$
unfolding *fix-def2*
apply (*rule admD* [*OF adm chain-iterate*])
apply (*rule nat-less-induct*)
apply (*case-tac n*)
apply (*simp add*: *0*)
apply (*case-tac nat*)
apply (*simp add*: *1*)
apply (*frule-tac x*=*nat* **in** *spec*)
apply (*simp add*: *step*)
done

lemma *parallel-fix-ind*:
  assumes *adm*: *adm* $(\lambda x.\ P\ (fst\ x)\ (snd\ x))$
  assumes *base*: $P\ \bot\ \bot$
  assumes *step*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (F{\cdot}x)\ (G{\cdot}y)$
  shows $P\ (fix{\cdot}F)\ (fix{\cdot}G)$
proof −
  from *adm* **have** *adm′*: *adm* (*split P*)
    unfolding *split-def* **.**
  have $\bigwedge i.\ P\ (iterate\ i{\cdot}F{\cdot}\bot)\ (iterate\ i{\cdot}G{\cdot}\bot)$
    by (*induct-tac i*, *simp add*: *base*, *simp add*: *step*)
  hence $\bigwedge i.$ *split P* (*iterate* $i{\cdot}F{\cdot}\bot$, *iterate* $i{\cdot}G{\cdot}\bot$)
    by *simp*
  hence *split P* ($\bigsqcup i.$ (*iterate* $i{\cdot}F{\cdot}\bot$, *iterate* $i{\cdot}G{\cdot}\bot$))
    by − (*rule admD* [*OF adm′*], *simp*, *assumption*)
  hence *split P* ($\bigsqcup i.$ *iterate* $i{\cdot}F{\cdot}\bot$, $\bigsqcup i.$ *iterate* $i{\cdot}G{\cdot}\bot$)
    by (*simp add*: *thelub-Pair*)
  hence P ($\bigsqcup i.$ *iterate* $i{\cdot}F{\cdot}\bot$) ($\bigsqcup i.$ *iterate* $i{\cdot}G{\cdot}\bot$)
    by *simp*
  thus $P\ (fix{\cdot}F)\ (fix{\cdot}G)$
    by (*simp add*: *fix-def2*)
qed

19.4 Fixed-points on product types

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

lemma *fix-cprod*:
  $fix{\cdot}(F::'a \times 'b \to 'a \times 'b) =$
   $(\mu\ x.\ fst\ (F{\cdot}(x,\ \mu\ y.\ snd\ (F{\cdot}(x,\ y)))),$
    $\mu\ y.\ snd\ (F{\cdot}(\mu\ x.\ fst\ (F{\cdot}(x,\ \mu\ y.\ snd\ (F{\cdot}(x,\ y)))),\ y)))$
  (**is** $fix{\cdot}F = (?x,\ ?y)$)
proof (*rule fix-eqI*)
  have *1*: $fst\ (F{\cdot}(?x,\ ?y)) = ?x$
    by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
  have *2*: $snd\ (F{\cdot}(?x,\ ?y)) = ?y$
    by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
  from *1 2* **show** $F{\cdot}(?x,\ ?y) = (?x,\ ?y)$ **by** (*simp add*: *Pair-fst-snd-eq*)
next
  fix *z* **assume** *F-z*: $F{\cdot}z = z$
  obtain *x y* **where** *z*: $z = (x,y)$ **by** (*rule prod.exhaust*)
  from *F-z z* **have** *F-x*: $fst\ (F{\cdot}(x,\ y)) = x$ **by** *simp*
  from *F-z z* **have** *F-y*: $snd\ (F{\cdot}(x,\ y)) = y$ **by** *simp*
  let $?y1 = \mu\ y.\ snd\ (F{\cdot}(x,\ y))$
  have $?y1 \sqsubseteq y$ **by** (*rule fix-least*, *simp add*: *F-y*)
  hence $fst\ (F{\cdot}(x,\ ?y1)) \sqsubseteq fst\ (F{\cdot}(x,\ y))$
    by (*simp add*: *fst-monofun monofun-cfun*)
  hence $fst\ (F{\cdot}(x,\ ?y1)) \sqsubseteq x$ **using** *F-x* **by** *simp*
  hence *1*: $?x \sqsubseteq x$ **by** (*simp add*: *fix-least-below*)
  hence $snd\ (F{\cdot}(?x,\ y)) \sqsubseteq snd\ (F{\cdot}(x,\ y))$
    by (*simp add*: *snd-monofun monofun-cfun*)
  hence $snd\ (F{\cdot}(?x,\ y)) \sqsubseteq y$ **using** *F-y* **by** *simp*
  hence *2*: $?y \sqsubseteq y$ **by** (*simp add*: *fix-least-below*)
  show $(?x,\ ?y) \sqsubseteq z$ **using** *z 1 2* **by** *simp*
qed

end

20 Fixrec: Package for defining recursive functions in HOLCF

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

20.1 Pattern-match monad

default-sort *cpo*

pcpodef (**open**) *'a match* = *UNIV*::(*one* ++ *'a u*) *set*
by *simp-all*

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

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

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

lemma *matchE* [*case-names bottom fail succeed*, *cases type*: *match*]:
⟦*p* = ⊥ ⟹ *Q*; *p* = *fail* ⟹ *Q*; ⋀*x*. *p* = *succeed*·*x* ⟹ *Q*⟧ ⟹ *Q*
unfolding *fail-def succeed-def*
apply (*cases p*, *rename-tac r*)
apply (*rule-tac p*=*r* **in** *ssumE*, *simp add*: *Abs-match-strict*)
apply (*rule-tac p*=*x* **in** *oneE*, *simp*, *simp*)
apply (*rule-tac p*=*y* **in** *upE*, *simp*, *simp add*: *cont-Abs-match*)
done

lemma *succeed-defined* [*simp*]: *succeed*·*x* ≠ ⊥
by (*simp add*: *succeed-def cont-Abs-match Abs-match-defined*)

lemma *fail-defined* [*simp*]: *fail* ≠ ⊥
by (*simp add*: *fail-def Abs-match-defined*)

lemma *succeed-eq* [*simp*]: (*succeed*·*x* = *succeed*·*y*) = (*x* = *y*)
by (*simp add*: *succeed-def cont-Abs-match Abs-match-inject*)

lemma *succeed-neq-fail* [*simp*]:
succeed·*x* ≠ *fail fail* ≠ *succeed*·*x*
by (*simp-all add*: *succeed-def fail-def cont-Abs-match Abs-match-inject*)

lemma *match-case-simps* [*simp*]:
match-case·*f*·*r*·⊥ = ⊥
match-case·*f*·*r*·*fail* = *f*
match-case·*f*·*r*·(*succeed*·*x*) = *r*·*x*
by (*simp-all add*: *succeed-def fail-def match-case-def cont-Rep-match*
cont2cont-LAM
cont-Abs-match Abs-match-inverse Rep-match-strict)

translations

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

20.1.1 Run operator

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

rewrite rules for run

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

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

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

20.1.2 Monad plus operator

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

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

rewrite rules for mplus

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

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

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

lemma *mplus-fail2* [*simp*]: *m +++ fail = m*
by (*cases m*, *simp-all*)

lemma *mplus-assoc*: *(x +++ y) +++ z = x +++ (y +++ z)*
by (*cases x*, *simp-all*)

20.2 Match functions for built-in types

default-sort *pcpo*

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

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

definition
match-spair :: *'a ⊗ 'b → ('a → 'b → 'c match) → 'c match*
where
match-spair = (Λ x k. ssplit·k·x)

definition
match-sinl :: *'a ⊕ 'b → ('a → 'c match) → 'c match*
where
match-sinl = (Λ x k. sscase·k·(Λ b. fail)·x)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

20.3 Mutual recursion

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

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

by *simp*

lemma *Pair-eqD1*: $(x, y) = (x', y') \Longrightarrow x = x'$
by *simp*

lemma *Pair-eqD2*: $(x, y) = (x', y') \Longrightarrow y = y'$
by *simp*

lemma *def-cont-fix-eq*:
$[\![f \equiv fix{\cdot}(Abs\text{-}CFun\ F);\ cont\ F]\!] \Longrightarrow f = F\ f$
by (*simp*, *subst fix-eq*, *simp*)

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

lemma for proving rewrite rules

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

20.4 Initializing the fixrec package

use *Tools/holcf-library.ML*
use *Tools/fixrec.ML*

setup ⟪ *Fixrec.setup* ⟫

setup ⟪
Fixrec.add-matchers
[(@{*const-name up*}, @{*const-name match-up*}),
(@{*const-name sinl*}, @{*const-name match-sinl*}),
(@{*const-name sinr*}, @{*const-name match-sinr*}),
(@{*const-name spair*}, @{*const-name match-spair*}),
(@{*const-name Pair*}, @{*const-name match-cpair*}),
(@{*const-name ONE*}, @{*const-name match-ONE*}),
(@{*const-name TT*}, @{*const-name match-TT*}),
(@{*const-name FF*}, @{*const-name match-FF*}),
(@{*const-name UU*}, @{*const-name match-UU*})]
⟫

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

end

21 Completion: Defining bifinite domains by ideal completion

theory *Completion*
imports *Bifinite*
begin

21.1 Ideals over a preorder

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

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

lemma *idealI*:
  assumes $\exists x.\ x \in A$
  assumes $\bigwedge x\ y.\ \llbracket x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
  assumes $\bigwedge x\ y.\ \llbracket x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
  shows *ideal A*
unfolding *ideal-def* **using** *prems* **by** *fast*

lemma *idealD1*:
  $ideal\ A \Longrightarrow \exists x.\ x \in A$
unfolding *ideal-def* **by** *fast*

lemma *idealD2*:
  $\llbracket ideal\ A; x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
unfolding *ideal-def* **by** *fast*

lemma *idealD3*:
  $\llbracket ideal\ A; x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
unfolding *ideal-def* **by** *fast*

lemma *ideal-directed-finite*:
  assumes *A*: *ideal A*
  shows $\llbracket finite\ U; U \subseteq A \rrbracket \Longrightarrow \exists z{\in}A.\ \forall x{\in}U.\ x \preceq z$
apply (*induct U set*: *finite*)
apply (*simp add*: *idealD1* [*OF A*])
apply (*simp*, *clarify*, *rename-tac y*)
apply (*drule* (*1*) *idealD2* [*OF A*])
apply (*clarify*, *erule-tac x*=*z* **in** *rev-bexI*)
apply (*fast intro*: *r-trans*)
done

lemma *ideal-principal*: *ideal* $\{x.\ x \preceq z\}$
apply (*rule idealI*)
apply (*rule-tac x=z* **in** *exI*)
apply (*fast intro*: *r-refl*)
apply (*rule-tac x=z* **in** *bexI*, *fast*)
apply (*fast intro*: *r-refl*)
apply (*fast intro*: *r-trans*)
done

lemma *ex-ideal*: $\exists A.$ *ideal A*
by (*rule exI*, *rule ideal-principal*)

lemma *directed-image-ideal*:
assumes *A*: *ideal A*
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows *directed* (*f* ' *A*)
apply (*rule directedI*)
apply (*cut-tac idealD1* [*OF A*], *fast*)
apply (*clarify*, *rename-tac a b*)
apply (*drule* (*1*) *idealD2* [*OF A*])
apply (*clarify*, *rename-tac c*)
apply (*rule-tac x=f c* **in** *rev-bexI*)
apply (*erule imageI*)
apply (*simp add*: *f*)
done

lemma *lub-image-principal*:
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows $(\bigsqcup x{\in}\{x.\ x \preceq y\}.\ f\ x) = f\ y$
apply (*rule thelubI*)
apply (*rule is-lub-maximal*)
apply (*rule ub-imageI*)
apply (*simp add*: *f*)
apply (*rule imageI*)
apply (*simp add*: *r-refl*)
done

The set of ideals is a cpo

lemma *ideal-UN*:
fixes *A* :: *nat* $\Rightarrow$ *'a set*
assumes *ideal-A*: $\bigwedge i.$ *ideal* (*A i*)
assumes *chain-A*: $\bigwedge i\ j.\ i \leq j \Longrightarrow A\ i \subseteq A\ j$
shows *ideal* $(\bigcup i.\ A\ i)$
apply (*rule idealI*)
apply (*cut-tac idealD1* [*OF ideal-A*], *fast*)
apply (*clarify*, *rename-tac i j*)
apply (*drule subsetD* [*OF chain-A* [*OF le-maxI1*]])
apply (*drule subsetD* [*OF chain-A* [*OF le-maxI2*]])

apply (*drule* (*1*) *idealD2* [*OF ideal-A*])
apply *blast*
apply *clarify*
apply (*drule* (*1*) *idealD3* [*OF ideal-A*])
apply *fast*
done

lemma *typedef-ideal-po*:
fixes *Abs* :: *'a set* ⇒ *'b*::*below*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *below*: ⋀*x y*. *x* ⊑ *y* ⟷ *Rep x* ⊆ *Rep y*
shows *OFCLASS*(*'b*, *po-class*)
apply (*intro-classes*, *unfold below*)
apply (*rule subset-refl*)
apply (*erule* (*1*) *subset-trans*)
apply (*rule type-definition.Rep-inject* [*OF type*, *THEN iffD1*])
apply (*erule* (*1*) *subset-antisym*)
done

lemma
fixes *Abs* :: *'a set* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *below*: ⋀*x y*. *x* ⊑ *y* ⟷ *Rep x* ⊆ *Rep y*
assumes *S*: *chain S*
shows *typedef-ideal-lub*: *range S* <<| *Abs* (⋃ *i*. *Rep* (*S i*))
and *typedef-ideal-rep-contlub*: *Rep* (⨆ *i*. *S i*) = (⋃ *i*. *Rep* (*S i*))
proof −
have *1*: *ideal* (⋃ *i*. *Rep* (*S i*))
apply (*rule ideal-UN*)
apply (*rule type-definition.Rep* [*OF type*, *unfolded mem-Collect-eq*])
apply (*subst below* [*symmetric*])
apply (*erule chain-mono* [*OF S*])
done
hence *2*: *Rep* (*Abs* (⋃ *i*. *Rep* (*S i*))) = (⋃ *i*. *Rep* (*S i*))
by (*simp add*: *type-definition.Abs-inverse* [*OF type*])
show *3*: *range S* <<| *Abs* (⋃ *i*. *Rep* (*S i*))
apply (*rule is-lubI*)
apply (*rule is-ubI*)
apply (*simp add*: *below 2*, *fast*)
apply (*simp add*: *below 2 is-ub-def*, *fast*)
done
hence *4*: (⨆ *i*. *S i*) = *Abs* (⋃ *i*. *Rep* (*S i*))
by (*rule thelubI*)
show *5*: *Rep* (⨆ *i*. *S i*) = (⋃ *i*. *Rep* (*S i*))
by (*simp add*: *4 2*)
qed

lemma *typedef-ideal-cpo*:
fixes *Abs* :: *'a set* ⇒ *'b*::*po*

assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *below*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow Rep\ x \subseteq Rep\ y$
shows *OFCLASS*('*b*, *cpo-class*)
by (*default*, *rule exI*, *erule typedef-ideal-lub* [*OF type below*])

end

interpretation *below*: *preorder below* :: '*a*::*po* ⇒ '*a* ⇒ *bool*
apply *unfold-locales*
apply (*rule below-refl*)
apply (*erule* (*1*) *below-trans*)
done

21.2 Lemmas about least upper bounds

lemma *finite-directed-contains-lub*:
$[\![finite\ S;\ directed\ S]\!] \Longrightarrow \exists\, u{\in}S.\ S <<|\ u$
apply (*drule* (*1*) *directed-finiteD*, *rule subset-refl*)
apply (*erule bexE*)
apply (*rule rev-bexI*, *assumption*)
apply (*erule* (*1*) *is-lub-maximal*)
done

lemma *lub-finite-directed-in-self*:
$[\![finite\ S;\ directed\ S]\!] \Longrightarrow lub\ S \in S$
apply (*drule* (*1*) *finite-directed-contains-lub*, *clarify*)
apply (*drule thelubI*, *simp*)
done

lemma *finite-directed-has-lub*: $[\![finite\ S;\ directed\ S]\!] \Longrightarrow \exists\, u.\ S <<|\ u$
by (*drule* (*1*) *finite-directed-contains-lub*, *fast*)

lemma *is-ub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ x \in S]\!] \Longrightarrow x \sqsubseteq lub\ S$
apply (*erule exE*, *drule lubI*)
apply (*drule is-lubD1*)
apply (*erule* (*1*) *is-ubD*)
done

lemma *is-lub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ S <|\ x]\!] \Longrightarrow lub\ S \sqsubseteq x$
by (*erule exE*, *drule lubI*, *erule is-lub-lub*)

21.3 Locale for ideal completion

locale *basis-take* = *preorder* +
fixes *take* :: *nat* ⇒ '*a*::*type* ⇒ '*a*
assumes *take-less*: *take n a* ≼ *a*
assumes *take-take*: *take n* (*take n a*) = *take n a*
assumes *take-mono*: *a* ≼ *b* ⟹ *take n a* ≼ *take n b*
assumes *take-chain*: *take n a* ≼ *take* (*Suc n*) *a*
assumes *finite-range-take*: *finite* (*range* (*take n*))

assumes *take-covers*: $\exists\, n.\ take\ n\ a = a$
begin

lemma *take-chain-less*: $m < n \Longrightarrow take\ m\ a \preceq take\ n\ a$
by (*erule less-Suc-induct*, *rule take-chain*, *erule* (*1*) *r-trans*)

lemma *take-chain-le*: $m \leq n \Longrightarrow take\ m\ a \preceq take\ n\ a$
by (*cases* $m = n$, *simp add*: *r-refl*, *simp add*: *take-chain-less*)

end

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

lemma *finite-take-rep*: *finite* (*take n ' rep x*)
by (*rule finite-subset* [*OF image-mono* [*OF subset-UNIV*] *finite-range-take*])

lemma *rep-mono*: $x \sqsubseteq y \Longrightarrow rep\ x \subseteq rep\ y$
apply (*frule bin-chain*)
apply (*drule rep-contlub*)
apply (*simp only*: *thelubI* [*OF lub-bin-chain*])
apply (*rule subsetI*, *rule UN-I* [**where** *a=0*], *simp-all*)
done

lemma *below-def*: $x \sqsubseteq y \longleftrightarrow rep\ x \subseteq rep\ y$
by (*rule iffI* [*OF rep-mono subset-repD*])

lemma *rep-eq*: $rep\ x = \{a.\ principal\ a \sqsubseteq x\}$
unfolding *below-def rep-principal*
apply *safe*
apply (*erule* (*1*) *idealD3* [*OF ideal-rep*])
apply (*erule subsetD*, *simp add*: *r-refl*)
done

lemma *mem-rep-iff-principal-below*: $a \in rep\ x \longleftrightarrow principal\ a \sqsubseteq x$
by (*simp add*: *rep-eq*)

lemma *principal-below-iff-mem-rep*: $principal\ a \sqsubseteq x \longleftrightarrow a \in rep\ x$
by (*simp add*: *rep-eq*)

lemma *principal-below-iff* [*simp*]: $principal\ a \sqsubseteq principal\ b \longleftrightarrow a \preceq b$
by (*simp add*: *principal-below-iff-mem-rep rep-principal*)

lemma *principal-eq-iff*: *principal a = principal b* $\longleftrightarrow$ $a \preceq b \wedge b \preceq a$
unfolding *po-eq-conv* [**where** *'a='b*] *principal-below-iff* **..**

lemma *repD*: $a \in$ *rep x* $\Longrightarrow$ *principal a* $\sqsubseteq x$
by (*simp add*: *rep-eq*)

lemma *principal-mono*: $a \preceq b \Longrightarrow$ *principal a* $\sqsubseteq$ *principal b*
by (*simp only*: *principal-below-iff*)

lemma *belowI*: $(\bigwedge a.$ *principal a* $\sqsubseteq x \Longrightarrow$ *principal a* $\sqsubseteq u) \Longrightarrow x \sqsubseteq u$
unfolding *principal-below-iff-mem-rep*
by (*simp add*: *below-def subset-eq*)

lemma *lub-principal-rep*: *principal ' rep x <<| x*
apply (*rule is-lubI*)
apply (*rule ub-imageI*)
apply (*erule repD*)
apply (*subst below-def*)
apply (*rule subsetI*)
apply (*drule (1) ub-imageD*)
apply (*simp add*: *rep-eq*)
done

21.4 Defining functions in terms of basis elements

definition
basis-fun :: (*'a*::*type* $\Rightarrow$ *'c*::*cpo*) $\Rightarrow$ *'b* $\rightarrow$ *'c* **where**
basis-fun = ($\lambda f.$ (Λ *x. lub* (*f ' rep x*)))

lemma *basis-fun-lemma0*:
fixes *f* :: *'a*::*type* $\Rightarrow$ *'c*::*cpo*
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $\exists u.$ *f ' take i ' rep x <<| u*
apply (*rule finite-directed-has-lub*)
apply (*rule finite-imageI*)
apply (*rule finite-take-rep*)
apply (*subst image-image*)
apply (*rule directed-image-ideal*)
apply (*rule ideal-rep*)
apply (*rule f-mono*)
apply (*erule take-mono*)
done

lemma *basis-fun-lemma1*:
fixes *f* :: *'a*::*type* $\Rightarrow$ *'c*::*cpo*
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows *chain* ($\lambda i.$ *lub* (*f ' take i ' rep x*))
apply (*rule chainI*)
apply (*rule is-lub-thelub0*)

```
  apply (rule basis-fun-lemma0, erule f-mono)
 apply (rule is-ubI, clarsimp, rename-tac a)
 apply (rule below-trans [OF f-mono [OF take-chain]])
 apply (rule is-ub-thelub0)
  apply (rule basis-fun-lemma0, erule f-mono)
 apply simp
done

lemma basis-fun-lemma2:
  fixes f :: 'a::type ⇒ 'c::cpo
  assumes f-mono: ⋀a b. a ⪯ b ⟹ f a ⊑ f b
  shows f ‘ rep x <<| (⨆ i. lub (f ‘ take i ‘ rep x))
 apply (rule is-lubI)
 apply (rule ub-imageI, rename-tac a)
  apply (cut-tac a=a in take-covers, erule exE, rename-tac i)
  apply (erule subst)
  apply (rule rev-below-trans)
   apply (rule-tac x=i in is-ub-thelub)
   apply (rule basis-fun-lemma1, erule f-mono)
  apply (rule is-ub-thelub0)
   apply (rule basis-fun-lemma0, erule f-mono)
  apply simp
 apply (rule is-lub-thelub [OF - ub-rangeI])
  apply (rule basis-fun-lemma1, erule f-mono)
 apply (rule is-lub-thelub0)
  apply (rule basis-fun-lemma0, erule f-mono)
 apply (rule is-ubI, clarsimp, rename-tac a)
 apply (rule below-trans [OF f-mono [OF take-less]])
 apply (erule (1) ub-imageD)
done

lemma basis-fun-lemma:
  fixes f :: 'a::type ⇒ 'c::cpo
  assumes f-mono: ⋀a b. a ⪯ b ⟹ f a ⊑ f b
  shows ∃ u. f ‘ rep x <<| u
by (rule exI, rule basis-fun-lemma2, erule f-mono)

lemma basis-fun-beta:
  fixes f :: 'a::type ⇒ 'c::cpo
  assumes f-mono: ⋀a b. a ⪯ b ⟹ f a ⊑ f b
  shows basis-fun f·x = lub (f ‘ rep x)
unfolding basis-fun-def
proof (rule beta-cfun)
  have lub: ⋀x. ∃ u. f ‘ rep x <<| u
    using f-mono by (rule basis-fun-lemma)
  show cont: cont (λx. lub (f ‘ rep x))
    apply (rule contI2)
     apply (rule monofunI)
     apply (rule is-lub-thelub0 [OF lub ub-imageI])
```

```
    apply (rule is-ub-thelub0 [OF lub imageI])
    apply (erule (1) subsetD [OF rep-mono])
   apply (rule is-lub-thelub0 [OF lub ub-imageI])
   apply (simp add: rep-contlub, clarify)
   apply (erule rev-below-trans [OF is-ub-thelub])
   apply (erule is-ub-thelub0 [OF lub imageI])
   done
qed

lemma basis-fun-principal:
  fixes f :: 'a::type ⇒ 'c::cpo
  assumes f-mono: ⋀a b. a ≺ b ⟹ f a ⊑ f b
  shows basis-fun f·(principal a) = f a
apply (subst basis-fun-beta, erule f-mono)
apply (subst rep-principal)
apply (rule lub-image-principal, erule f-mono)
done

lemma basis-fun-mono:
  assumes f-mono: ⋀a b. a ≺ b ⟹ f a ⊑ f b
  assumes g-mono: ⋀a b. a ≺ b ⟹ g a ⊑ g b
  assumes below: ⋀a. f a ⊑ g a
  shows basis-fun f ⊑ basis-fun g
 apply (rule below-cfun-ext)
 apply (simp only: basis-fun-beta f-mono g-mono)
 apply (rule is-lub-thelub0)
  apply (rule basis-fun-lemma, erule f-mono)
 apply (rule ub-imageI, rename-tac a)
 apply (rule below-trans [OF below])
 apply (rule is-ub-thelub0)
  apply (rule basis-fun-lemma, erule g-mono)
 apply (erule imageI)
done

lemma compact-principal [simp]: compact (principal a)
by (rule compactI2, simp add: principal-below-iff-mem-rep rep-contlub)
```

21.5 Bifiniteness of ideal completions

```
definition
  completion-approx :: nat ⇒ 'b → 'b where
  completion-approx = (λi. basis-fun (λa. principal (take i a)))

lemma completion-approx-beta:
  completion-approx i·x = (⨆ a∈rep x. principal (take i a))
unfolding completion-approx-def
by (simp add: basis-fun-beta principal-mono take-mono)

lemma completion-approx-principal:
```

completion-approx i·(principal a) = principal (take i a)
unfolding *completion-approx-def*
by (*simp add*: *basis-fun-principal principal-mono take-mono*)

lemma *chain-completion-approx*: *chain completion-approx*
unfolding *completion-approx-def*
apply (*rule chainI*)
apply (*rule basis-fun-mono*)
apply (*erule principal-mono* [*OF take-mono*])
apply (*erule principal-mono* [*OF take-mono*])
apply (*rule principal-mono* [*OF take-chain*])
done

lemma *lub-completion-approx*: $(\bigsqcup i.\ \textit{completion-approx}\ i{\cdot}x) = x$
unfolding *completion-approx-beta*
apply (*subst image-image* [**where** *f*=*principal*, *symmetric*])
apply (*rule unique-lub* [*OF - lub-principal-rep*])
apply (*rule basis-fun-lemma2*, *erule principal-mono*)
done

lemma *completion-approx-eq-principal*:
$\exists\, a{\in}\textit{rep}\ x.\ \textit{completion-approx}\ i{\cdot}x = \textit{principal}\ (\textit{take}\ i\ a)$
unfolding *completion-approx-beta*
apply (*subst image-image* [**where** *f*=*principal*, *symmetric*])
apply (*subgoal-tac finite* (*principal ‘ take i ‘ rep x*))
apply (*subgoal-tac directed* (*principal ‘ take i ‘ rep x*))
apply (*drule* (*1*) *lub-finite-directed-in-self*, *fast*)
apply (*subst image-image*)
apply (*rule directed-image-ideal*)
apply (*rule ideal-rep*)
apply (*erule principal-mono* [*OF take-mono*])
apply (*rule finite-imageI*)
apply (*rule finite-take-rep*)
done

lemma *completion-approx-idem*:
completion-approx i·(completion-approx i·x) = completion-approx i·x
using *completion-approx-eq-principal* [**where** *i*=*i* **and** *x*=*x*]
by (*auto simp add*: *completion-approx-principal take-take*)

lemma *finite-fixes-completion-approx*:
finite $\{x.\ \textit{completion-approx}\ i{\cdot}x = x\}$ (**is** *finite ?S*)
apply (*subgoal-tac ?S* $\subseteq$ *principal ‘ range* (*take i*))
apply (*erule finite-subset*)
apply (*rule finite-imageI*)
apply (*rule finite-range-take*)
apply (*clarify*, *erule subst*)
apply (*cut-tac x*=*x* **and** *i*=*i* **in** *completion-approx-eq-principal*)
apply *fast*

done

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes *P*: $\bigwedge a.\ P\ (\mathit{principal}\ a)$
shows *P x*
apply (*subgoal-tac P* ($\bigsqcup i.\ \mathit{completion\text{-}approx}\ i{\cdot}x$))
apply (*simp add*: *lub-completion-approx*)
apply (*rule admD* [*OF adm*])
apply (*simp add*: *chain-completion-approx*)
apply (*cut-tac x=x* **and** *i=i* **in** *completion-approx-eq-principal*)
apply (*clarify*, *simp add*: *P*)
done

lemma *principal-induct2*:
$[\![\bigwedge y.\ \mathit{adm}\ (\lambda x.\ P\ x\ y);\ \bigwedge x.\ \mathit{adm}\ (\lambda y.\ P\ x\ y);$
$\bigwedge a\ b.\ P\ (\mathit{principal}\ a)\ (\mathit{principal}\ b)]\!] \Longrightarrow P\ x\ y$
apply (*rule-tac x=y* **in** *spec*)
apply (*rule-tac x=x* **in** *principal-induct*, *simp*)
apply (*rule allI*, *rename-tac y*)
apply (*rule-tac x=y* **in** *principal-induct*, *simp*)
apply *simp*
done

lemma *compact-imp-principal*: *compact x* $\Longrightarrow \exists\, a.\ x = \mathit{principal}\ a$
apply (*drule adm-compact-neq* [*OF - cont-id*])
apply (*drule admD2* [**where** $Y{=}\lambda n.\ \mathit{completion\text{-}approx}\ n{\cdot}x$])
apply (*simp add*: *chain-completion-approx*)
apply (*simp add*: *lub-completion-approx*)
apply (*erule exE*, *erule ssubst*)
apply (*cut-tac i=i* **and** *x=x* **in** *completion-approx-eq-principal*)
apply (*clarify*, *erule exI*)
done

end

end

22 Eventual: Eventually-constant sequences

theory *Eventual*
imports *Infinite-Set*
begin

22.1 Lemmas about MOST

lemma *MOST-INFM*:
assumes *inf*: *infinite* (*UNIV*::'*a set*)

shows *MOST x::'a. P x ⟹ INFM x::'a. P x*
unfolding *Alm-all-def Inf-many-def*
apply (*auto simp add*: *Collect-neg-eq*)
apply (*drule* (*1*) *finite-UnI*)
apply (*simp add*: *Compl-partition2 inf*)
done

lemma *MOST-SucI*: *MOST n. P n ⟹ MOST n. P* (*Suc n*)
by (*rule MOST-inj* [*OF - inj-Suc*])

lemma *MOST-SucD*: *MOST n. P* (*Suc n*) *⟹ MOST n. P n*
unfolding *MOST-nat*
apply (*clarify*, *rule-tac x=Suc m* **in** *exI*, *clarify*)
apply (*erule Suc-lessE*, *simp*)
done

lemma *MOST-Suc-iff*: (*MOST n. P* (*Suc n*)) *⟷* (*MOST n. P n*)
by (*rule iffI* [*OF MOST-SucD MOST-SucI*])

lemma *INFM-finite-Bex-distrib*:
finite A ⟹ (*INFM y. ∃ x∈A. P x y*) *⟷* (*∃ x∈A. INFM y. P x y*)
by (*induct set*: *finite*, *simp*, *simp add*: *INFM-disj-distrib*)

lemma *MOST-finite-Ball-distrib*:
finite A ⟹ (*MOST y. ∀ x∈A. P x y*) *⟷* (*∀ x∈A. MOST y. P x y*)
by (*induct set*: *finite*, *simp*, *simp add*: *MOST-conj-distrib*)

lemma *MOST-ge-nat*: *MOST n::nat. m ≤ n*
unfolding *MOST-nat-le* **by** *fast*

22.2 Eventually constant sequences

definition
eventually-constant :: (*nat ⇒ 'a*) *⇒ bool*
where
eventually-constant S = (*∃ x. MOST i. S i* = *x*)

lemma *eventually-constant-MOST-MOST*:
eventually-constant S ⟷ (*MOST m. MOST n. S n* = *S m*)
unfolding *eventually-constant-def MOST-nat*
apply *safe*
apply (*rule-tac x=m* **in** *exI*, *clarify*)
apply (*rule-tac x=m* **in** *exI*, *clarify*)
apply *simp*
apply *fast*
done

lemma *eventually-constantI*: *MOST i. S i* = *x ⟹ eventually-constant S*
unfolding *eventually-constant-def* **by** *fast*

lemma *eventually-constant-comp*:
eventually-constant (λ*i*. *S i*) ⟹ *eventually-constant* (λ*i*. *f* (*S i*))
unfolding *eventually-constant-def*
apply (*erule exE*, *rule-tac x*=*f x* **in** *exI*)
apply (*erule MOST-mono*, *simp*)
done

lemma *eventually-constant-Suc-iff*:
eventually-constant (λ*i*. *S* (*Suc i*)) ⟷ *eventually-constant* (λ*i*. *S i*)
unfolding *eventually-constant-def*
by (*subst MOST-Suc-iff*, *rule refl*)

lemma *eventually-constant-SucD*:
eventually-constant (λ*i*. *S* (*Suc i*)) ⟹ *eventually-constant* (λ*i*. *S i*)
by (*rule eventually-constant-Suc-iff* [*THEN iffD1*])

22.3 Limits of eventually constant sequences

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

lemma *eventual-eqI*: *MOST i*. *S i* = *x* ⟹ *eventual S* = *x*
unfolding *eventual-def*
apply (*rule the-equality*, *assumption*)
apply (*rename-tac y*)
apply (*subgoal-tac MOST i*::*nat*. *y* = *x*, *simp*)
apply (*erule MOST-rev-mp*)
apply (*erule MOST-rev-mp*)
apply *simp*
done

lemma *MOST-eq-eventual*:
eventually-constant S ⟹ *MOST i*. *S i* = *eventual S*
unfolding *eventually-constant-def*
by (*erule exE*, *simp add*: *eventual-eqI*)

lemma *eventual-mem-range*:
eventually-constant S ⟹ *eventual S* ∈ *range S*
apply (*drule MOST-eq-eventual*)
apply (*simp only*: *MOST-nat-le*, *clarify*)
apply (*drule spec*, *drule mp*, *rule order-refl*)
apply (*erule range-eqI* [*OF sym*])
done

lemma *eventually-constant-MOST-iff*:
assumes *S*: *eventually-constant S*
shows (*MOST n*. *P* (*S n*)) ⟷ *P* (*eventual S*)

apply (*subgoal-tac* (*MOST n. P* (*S n*)) ⟷ (*MOST n*::*nat. P* (*eventual S*)))
apply *simp*
apply (*rule iffI*)
apply (*rule MOST-rev-mp* [*OF MOST-eq-eventual* [*OF S*]])
apply (*erule MOST-mono*, *force*)
apply (*rule MOST-rev-mp* [*OF MOST-eq-eventual* [*OF S*]])
apply (*erule MOST-mono*, *simp*)
done

lemma *MOST-eventual*:
⟦*eventually-constant S*; *MOST n. P* (*S n*)⟧ ⟹ *P* (*eventual S*)
proof −
assume *eventually-constant S*
hence *MOST n. S n* = *eventual S*
by (*rule MOST-eq-eventual*)
moreover assume *MOST n. P* (*S n*)
ultimately have *MOST n. S n* = *eventual S* ∧ *P* (*S n*)
by (*rule MOST-conj-distrib* [*THEN iffD2*, *OF conjI*])
hence *MOST n*::*nat. P* (*eventual S*)
by (*rule MOST-mono*) *auto*
thus *?thesis* **by** *simp*
qed

lemma *eventually-constant-MOST-Suc-eq*:
eventually-constant S ⟹ *MOST n. S* (*Suc n*) = *S n*
apply (*drule MOST-eq-eventual*)
apply (*frule MOST-Suc-iff* [*THEN iffD2*])
apply (*erule MOST-rev-mp*)
apply (*erule MOST-rev-mp*)
apply *simp*
done

lemma *eventual-comp*:
eventually-constant S ⟹ *eventual* (λ*i. f* (*S i*)) = *f* (*eventual* (λ*i. S i*))
apply (*rule eventual-eqI*)
apply (*rule MOST-mono*)
apply (*erule MOST-eq-eventual*)
apply *simp*
done

end

23 Algebraic: Algebraic deflations

theory *Algebraic*
imports *Completion Fix Eventual*
begin

23.1 Constructing finite deflations by iteration

lemma *finite-deflation-imp-deflation*:
finite-deflation d $\Longrightarrow$ *deflation d*
unfolding *finite-deflation-def* **by** *simp*

lemma *le-Suc-induct*:
assumes *le*: $i \leq j$
assumes *step*: $\bigwedge i.\ P\ i\ (Suc\ i)$
assumes *refl*: $\bigwedge i.\ P\ i\ i$
assumes *trans*: $\bigwedge i\ j\ k.\ [\![P\ i\ j;\ P\ j\ k]\!] \Longrightarrow P\ i\ k$
shows $P\ i\ j$
proof (*cases* $i = j$)
assume $i = j$
thus $P\ i\ j$ **by** (*simp add*: *refl*)
next
assume $i \neq j$
with *le* **have** $i < j$ **by** *simp*
thus $P\ i\ j$ **using** *step trans* **by** (*rule less-Suc-induct*)
qed

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

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

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

lemma *iterate-below*: *iterate* $i{\cdot}f{\cdot}x \sqsubseteq x$
by (*induct i*, *simp-all add*: *below-trans* [*OF below*])

lemma *iterate-fixed*: $f{\cdot}x = x \Longrightarrow$ *iterate* $i{\cdot}f{\cdot}x = x$
by (*induct i*, *simp-all*)

lemma *antichain-iterate-app*: $i \leq j \Longrightarrow$ *iterate* $j{\cdot}f{\cdot}x \sqsubseteq$ *iterate* $i{\cdot}f{\cdot}x$
apply (*erule le-Suc-induct*)
apply (*simp add*: *below*)
apply (*rule below-refl*)
apply (*erule* (*1*) *below-trans*)
done

lemma *finite-range-iterate-app*: *finite* (*range* $(\lambda i.$ *iterate* $i{\cdot}f{\cdot}x)$)
proof (*rule finite-subset*)
show *range* $(\lambda i.$ *iterate* $i{\cdot}f{\cdot}x) \subseteq$ *insert x* (*range* $(\lambda x.\ f{\cdot}x)$)
by (*clarify*, *case-tac i*, *simp-all*)

show *finite* (*insert x* (*range* (λx. *f*·*x*)))
by (*simp add*: *finite-range*)
qed

lemma *eventually-constant-iterate-app*:
eventually-constant (λi. *iterate i*·*f*·*x*)
unfolding *eventually-constant-def MOST-nat-le*
proof −
let *?Y* = λi. *iterate i*·*f*·*x*
have ∃*j*. ∀*k*. *?Y j* ⊑ *?Y k*
apply (*rule finite-range-has-max*)
apply (*erule antichain-iterate-app*)
apply (*rule finite-range-iterate-app*)
done
then obtain *j* **where** *j*: ⋀*k*. *?Y j* ⊑ *?Y k* **by** *fast*
show ∃*z m*. ∀*n*≥*m*. *?Y n* = *z*
proof (*intro exI allI impI*)
fix *k*
assume *j* ≤ *k*
hence *?Y k* ⊑ *?Y j* **by** (*rule antichain-iterate-app*)
also have *?Y j* ⊑ *?Y k* **by** (*rule j*)
finally show *?Y k* = *?Y j* **.**
qed
qed

lemma *eventually-constant-iterate*:
eventually-constant (λn. *iterate n*·*f*)
proof −
have ∀*y*∈*range* (λx. *f*·*x*). *eventually-constant* (λi. *iterate i*·*f*·*y*)
by (*simp add*: *eventually-constant-iterate-app*)
hence ∀*y*∈*range* (λx. *f*·*x*). *MOST i. MOST j. iterate j*·*f*·*y* = *iterate i*·*f*·*y*
unfolding *eventually-constant-MOST-MOST* **.**
hence *MOST i. MOST j.* ∀*y*∈*range* (λx. *f*·*x*). *iterate j*·*f*·*y* = *iterate i*·*f*·*y*
by (*simp only*: *MOST-finite-Ball-distrib* [*OF finite-range*])
hence *MOST i. MOST j.* ∀*x*. *iterate j*·*f*·(*f*·*x*) = *iterate i*·*f*·(*f*·*x*)
by *simp*
hence *MOST i. MOST j.* ∀*x*. *iterate* (*Suc j*)·*f*·*x* = *iterate* (*Suc i*)·*f*·*x*
by (*simp only*: *iterate-Suc2*)
hence *MOST i. MOST j. iterate* (*Suc j*)·*f* = *iterate* (*Suc i*)·*f*
by (*simp only*: *expand-cfun-eq*)
hence *eventually-constant* (λi. *iterate* (*Suc i*)·*f*)
unfolding *eventually-constant-MOST-MOST* **.**
thus *eventually-constant* (λi. *iterate i*·*f*)
by (*rule eventually-constant-SucD*)
qed

abbreviation
d :: *'a* → *'a*
where

$d \equiv$ *eventual-iterate* f

lemma *MOST-d*: *MOST* n. P (*iterate* $n{\cdot}f$) $\Longrightarrow$ P d
unfolding *eventual-iterate-def*
using *eventually-constant-iterate* **by** (*rule MOST-eventual*)

lemma *f-d*: $f{\cdot}(d{\cdot}x) = d{\cdot}x$
apply (*rule MOST-d*)
apply (*subst iterate-Suc* [*symmetric*])
apply (*rule eventually-constant-MOST-Suc-eq*)
apply (*rule eventually-constant-iterate-app*)
done

lemma *d-fixed-iff*: $d{\cdot}x = x \longleftrightarrow f{\cdot}x = x$
proof
  assume $d{\cdot}x = x$
  with *f-d* [**where** $x{=}x$]
  show $f{\cdot}x = x$ **by** *simp*
next
  assume f: $f{\cdot}x = x$
  have $\forall\, n$. *iterate* $n{\cdot}f{\cdot}x = x$
    by (*rule allI*, *rule nat.induct*, *simp*, *simp add*: f)
  hence *MOST* n. *iterate* $n{\cdot}f{\cdot}x = x$
    by (*rule ALL-MOST*)
  thus $d{\cdot}x = x$
    by (*rule MOST-d*)
qed

lemma *finite-deflation-d*: *finite-deflation* d
proof
  fix $x :: 'a$
  have $d \in$ *range* (λn. *iterate* $n{\cdot}f$)
    unfolding *eventual-iterate-def*
    using *eventually-constant-iterate*
    by (*rule eventual-mem-range*)
  then obtain n **where** n: $d =$ *iterate* $n{\cdot}f$ **..**
  have *iterate* $n{\cdot}f{\cdot}(d{\cdot}x) = d{\cdot}x$
    using *f-d* **by** (*rule iterate-fixed*)
  thus $d{\cdot}(d{\cdot}x) = d{\cdot}x$
    by (*simp add*: n)
next
  fix $x :: 'a$
  show $d{\cdot}x \sqsubseteq x$
    by (*rule MOST-d*, *simp add*: *iterate-below*)
next
  from *finite-range*
  have *finite* $\{x.\ f{\cdot}x = x\}$
    by (*rule finite-range-imp-finite-fixes*)
  thus *finite* $\{x.\ d{\cdot}x = x\}$

```
    by (simp add: d-fixed-iff)
qed

lemma deflation-d: deflation d
using finite-deflation-d
by (rule finite-deflation-imp-deflation)

end

lemma finite-deflation-eventual-iterate:
  pre-deflation d ⟹ finite-deflation (eventual-iterate d)
by (rule pre-deflation.finite-deflation-d)

lemma pre-deflation-oo:
  assumes finite-deflation d
  assumes f: ⋀x. f·x ⊑ x
  shows pre-deflation (d oo f)
proof
  interpret d: finite-deflation d by fact
  fix x
  show ⋀x. (d oo f)·x ⊑ x
    by (simp, rule below-trans [OF d.below f])
  show finite (range (λx. (d oo f)·x))
    by (rule finite-subset [OF - d.finite-range], auto)
qed

lemma eventual-iterate-oo-fixed-iff:
  assumes finite-deflation d
  assumes f: ⋀x. f·x ⊑ x
  shows eventual-iterate (d oo f)·x = x ⟷ d·x = x ∧ f·x = x
proof −
  interpret d: finite-deflation d by fact
  let ?e = d oo f
  interpret e: pre-deflation d oo f
    using ‹finite-deflation d› f
    by (rule pre-deflation-oo)
  let ?g = eventual (λn. iterate n·?e)
  show ?thesis
    apply (subst e.d-fixed-iff)
    apply simp
    apply safe
    apply (erule subst)
    apply (rule d.idem)
    apply (rule below-antisym)
    apply (rule f)
    apply (erule subst, rule d.below)
    apply simp
    done
qed
```

lemma *eventual-mono*:
assumes *A*: *eventually-constant A*
assumes *B*: *eventually-constant B*
assumes *below*: $\bigwedge n.\ A\ n \sqsubseteq B\ n$
shows *eventual A* $\sqsubseteq$ *eventual B*
proof −
from *A* **have** *MOST n. A n = eventual A*
by (*rule MOST-eq-eventual*)
then have *MOST n. eventual A* $\sqsubseteq$ *B n*
by (*rule MOST-mono*) (*erule subst*, *rule below*)
with *B* **show** *eventual A* $\sqsubseteq$ *eventual B*
by (*rule MOST-eventual*)
qed

lemma *eventual-iterate-mono*:
assumes *f*: *pre-deflation f* **and** *g*: *pre-deflation g* **and** $f \sqsubseteq g$
shows *eventual-iterate f* $\sqsubseteq$ *eventual-iterate g*
unfolding *eventual-iterate-def*
apply (*rule eventual-mono*)
apply (*rule pre-deflation.eventually-constant-iterate* [*OF f*])
apply (*rule pre-deflation.eventually-constant-iterate* [*OF g*])
apply (*rule monofun-cfun-arg* [*OF* ‹$f \sqsubseteq g$›])
done

lemma *cont2cont-eventual-iterate-oo*:
assumes *d*: *finite-deflation d*
assumes *cont*: *cont f* **and** *below*: $\bigwedge x\ y.\ f\ x{\cdot}y \sqsubseteq y$
shows *cont* (λ*x*. *eventual-iterate* (*d oo f x*))
(**is** *cont ?e*)
proof (*rule contI2*)
show *monofun ?e*
apply (*rule monofunI*)
apply (*rule eventual-iterate-mono*)
apply (*rule pre-deflation-oo* [*OF d below*])
apply (*rule pre-deflation-oo* [*OF d below*])
apply (*rule monofun-cfun-arg*)
apply (*erule cont2monofunE* [*OF cont*])
done
next
fix *Y* :: *nat* ⇒ *'b*
assume *Y*: *chain Y*
with *cont* **have** *fY*: *chain* (λ*i*. *f* (*Y i*))
by (*rule ch2ch-cont*)
assume *eY*: *chain* (λ*i*. *?e* (*Y i*))
have *lub-below*: $\bigwedge x.\ f\ (\bigsqcup i.\ Y\ i){\cdot}x \sqsubseteq x$
by (*rule admD* [*OF - Y*], *simp add*: *cont*, *rule below*)
have *deflation* (*?e* (⨆ *i*. *Y i*))
apply (*rule pre-deflation.deflation-d*)

```
    apply (rule pre-deflation-oo [OF d lub-below])
    done
  then show ?e (⨆ i. Y i) ⊑ (⨆ i. ?e (Y i))
  proof (rule deflation.belowI)
    fix x :: 'a
    assume ?e (⨆ i. Y i)·x = x
    hence d·x = x and f (⨆ i. Y i)·x = x
      by (simp-all add: eventual-iterate-oo-fixed-iff [OF d lub-below])
    hence (⨆ i. f (Y i)·x) = x
      apply (simp only: cont2contlubE [OF cont Y])
      apply (simp only: contlub-cfun-fun [OF fY])
      done
    have compact (d·x)
      using d by (rule finite-deflation.compact)
    then have compact x
      using ‹d·x = x› by simp
    then have compact (⨆ i. f (Y i)·x)
      using ‹(⨆ i. f (Y i)·x) = x› by simp
    then have ∃ n. max-in-chain n (λi. f (Y i)·x)
      by − (rule compact-imp-max-in-chain, simp add: fY, assumption)
    then obtain n where n: max-in-chain n (λi. f (Y i)·x) ..
    then have f (Y n)·x = x
      using ‹(⨆ i. f (Y i)·x) = x› fY by (simp add: maxinch-is-thelub)
    with ‹d·x = x› have ?e (Y n)·x = x
      by (simp add: eventual-iterate-oo-fixed-iff [OF d below])
    moreover have ?e (Y n)·x ⊑ (⨆ i. ?e (Y i)·x)
      by (rule is-ub-thelub, simp add: eY)
    ultimately have x ⊑ (⨆ i. ?e (Y i))·x
      by (simp add: contlub-cfun-fun eY)
    also have (⨆ i. ?e (Y i))·x ⊑ x
      apply (rule deflation.below)
      apply (rule admD [OF adm-deflation eY])
      apply (rule pre-deflation.deflation-d)
      apply (rule pre-deflation-oo [OF d below])
      done
    finally show (⨆ i. ?e (Y i))·x = x ..
  qed
qed
```

23.2 Type constructor for finite deflations

```
default-sort profinite

typedef (open) 'a fin-defl = {d::'a → 'a. finite-deflation d}
by (fast intro: finite-deflation-approx)

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

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

instance ..
end

instance *fin-defl* :: (*profinite*) *po*
by (*rule typedef-po* [*OF type-definition-fin-defl below-fin-defl-def*])

lemma *finite-deflation-Rep-fin-defl*: *finite-deflation* (*Rep-fin-defl d*)
using *Rep-fin-defl* **by** *simp*

lemma *deflation-Rep-fin-defl*: *deflation* (*Rep-fin-defl d*)
using *finite-deflation-Rep-fin-defl*
by (*rule finite-deflation-imp-deflation*)

interpretation *Rep-fin-defl*: *finite-deflation Rep-fin-defl d*
by (*rule finite-deflation-Rep-fin-defl*)

lemma *fin-defl-belowI*:
  $(\bigwedge x.\ Rep\text{-}fin\text{-}defl\ a{\cdot}x = x \Longrightarrow Rep\text{-}fin\text{-}defl\ b{\cdot}x = x) \Longrightarrow a \sqsubseteq b$
unfolding *below-fin-defl-def*
by (*rule Rep-fin-defl.belowI*)

lemma *fin-defl-belowD*:
  $[\![a \sqsubseteq b;\ Rep\text{-}fin\text{-}defl\ a{\cdot}x = x]\!] \Longrightarrow Rep\text{-}fin\text{-}defl\ b{\cdot}x = x$
unfolding *below-fin-defl-def*
by (*rule Rep-fin-defl.belowD*)

lemma *fin-defl-eqI*:
  $(\bigwedge x.\ Rep\text{-}fin\text{-}defl\ a{\cdot}x = x \longleftrightarrow Rep\text{-}fin\text{-}defl\ b{\cdot}x = x) \Longrightarrow a = b$
apply (*rule below-antisym*)
apply (*rule fin-defl-belowI*, *simp*)
apply (*rule fin-defl-belowI*, *simp*)
done

lemma *Abs-fin-defl-mono*:
  $[\![finite\text{-}deflation\ a;\ finite\text{-}deflation\ b;\ a \sqsubseteq b]\!]$
    $\Longrightarrow Abs\text{-}fin\text{-}defl\ a \sqsubseteq Abs\text{-}fin\text{-}defl\ b$
unfolding *below-fin-defl-def*
by (*simp add*: *Abs-fin-defl-inverse*)

23.3 Take function for finite deflations

definition
  defl-approx :: *nat* ⇒ ('*a* → '*a*) ⇒ ('*a* → '*a*)
where
  defl-approx i d = *eventual-iterate* (*approx i oo d*)

lemma *finite-deflation-defl-approx*:
deflation d $\Longrightarrow$ *finite-deflation* (*defl-approx i d*)
unfolding *defl-approx-def*
apply (*rule pre-deflation.finite-deflation-d*)
apply (*rule pre-deflation-oo*)
apply (*rule finite-deflation-approx*)
apply (*erule deflation.below*)
done

lemma *deflation-defl-approx*:
deflation d $\Longrightarrow$ *deflation* (*defl-approx i d*)
apply (*rule finite-deflation-imp-deflation*)
apply (*erule finite-deflation-defl-approx*)
done

lemma *defl-approx-fixed-iff*:
deflation d $\Longrightarrow$ *defl-approx i d*·*x* = *x* $\longleftrightarrow$ *approx i*·*x* = *x* $\wedge$ *d*·*x* = *x*
unfolding *defl-approx-def*
apply (*rule eventual-iterate-oo-fixed-iff*)
apply (*rule finite-deflation-approx*)
apply (*erule deflation.below*)
done

lemma *defl-approx-below*:
⟦*a* $\sqsubseteq$ *b*; *deflation a*; *deflation b*⟧ $\Longrightarrow$ *defl-approx i a* $\sqsubseteq$ *defl-approx i b*
apply (*rule deflation.belowI*)
apply (*erule deflation-defl-approx*)
apply (*simp add*: *defl-approx-fixed-iff*)
apply (*erule* (*1*) *deflation.belowD*)
apply (*erule conjunct2*)
done

lemma *cont2cont-defl-approx*:
assumes *cont*: *cont f* **and** *below*: $\bigwedge$*x y*. *f x*·*y* $\sqsubseteq$ *y*
shows *cont* (λ*x*. *defl-approx i* (*f x*))
unfolding *defl-approx-def*
using *finite-deflation-approx assms*
by (*rule cont2cont-eventual-iterate-oo*)

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

lemma *Rep-fin-defl-fd-take*:
Rep-fin-defl (*fd-take i d*) = *defl-approx i* (*Rep-fin-defl d*)
unfolding *fd-take-def*
apply (*rule Abs-fin-defl-inverse* [*unfolded mem-Collect-eq*])
apply (*rule finite-deflation-defl-approx*)

apply (*rule deflation-Rep-fin-defl*)
done

lemma *fd-take-fixed-iff*:
  Rep-fin-defl (*fd-take i d*)·*x* = *x* ⟷
    approx i·*x* = *x* ∧ *Rep-fin-defl d*·*x* = *x*
unfolding *Rep-fin-defl-fd-take*
apply (*rule defl-approx-fixed-iff*)
apply (*rule deflation-Rep-fin-defl*)
done

lemma *fd-take-below*: *fd-take n d* ⊑ *d*
apply (*rule fin-defl-belowI*)
apply (*simp add*: *fd-take-fixed-iff*)
done

lemma *fd-take-idem*: *fd-take n* (*fd-take n d*) = *fd-take n d*
apply (*rule fin-defl-eqI*)
apply (*simp add*: *fd-take-fixed-iff*)
done

lemma *fd-take-mono*: *a* ⊑ *b* ⟹ *fd-take n a* ⊑ *fd-take n b*
apply (*rule fin-defl-belowI*)
apply (*simp add*: *fd-take-fixed-iff*)
apply (*simp add*: *fin-defl-belowD*)
done

lemma *approx-fixed-le-lemma*: ⟦*i* ≤ *j*; *approx i*·*x* = *x*⟧ ⟹ *approx j*·*x* = *x*
by (*erule subst*, *simp add*: *min-def*)

lemma *fd-take-chain*: *m* ≤ *n* ⟹ *fd-take m a* ⊑ *fd-take n a*
apply (*rule fin-defl-belowI*)
apply (*simp add*: *fd-take-fixed-iff*)
apply (*simp add*: *approx-fixed-le-lemma*)
done

lemma *finite-range-fd-take*: *finite* (*range* (*fd-take n*))
apply (*rule finite-imageD* [**where** *f*=λ*a*. {*x*. *Rep-fin-defl a*·*x* = *x*}])
apply (*rule finite-subset* [**where** *B*=*Pow* {*x*. *approx n*·*x* = *x*}])
apply (*clarify*, *simp add*: *fd-take-fixed-iff*)
apply (*simp add*: *finite-fixes-approx*)
apply (*rule inj-onI*, *clarify*)
apply (*simp add*: *expand-set-eq fin-defl-eqI*)
done

lemma *fd-take-covers*: ∃ *n*. *fd-take n a* = *a*
apply (*rule-tac x*=
  Max ((λ*x*. *LEAST n*. *approx n*·*x* = *x*) ‘ {*x*. *Rep-fin-defl a*·*x* = *x*}) **in** *exI*)
apply (*rule below-antisym*)

apply (*rule fd-take-below*)
apply (*rule fin-defl-belowI*)
apply (*simp add*: *fd-take-fixed-iff*)
apply (*rule approx-fixed-le-lemma*)
apply (*rule Max-ge*)
apply (*rule finite-imageI*)
apply (*rule Rep-fin-defl.finite-fixes*)
apply (*rule imageI*)
apply (*erule CollectI*)
apply (*rule LeastI-ex*)
apply (*rule profinite-compact-eq-approx*)
apply (*erule subst*)
apply (*rule Rep-fin-defl.compact*)
done

interpretation *fin-defl*: *basis-take below fd-take*
apply *default*
apply (*rule fd-take-below*)
apply (*rule fd-take-idem*)
apply (*erule fd-take-mono*)
apply (*rule fd-take-chain*, *simp*)
apply (*rule finite-range-fd-take*)
apply (*rule fd-take-covers*)
done

23.4 Defining algebraic deflations by ideal completion

typedef (**open**) *'a alg-defl* =
  {*S*::*'a fin-defl set*. *below.ideal S*}
by (*fast intro*: *below.ideal-principal*)

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

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

instance ..
end

instance *alg-defl* :: (*profinite*) *po*
by (*rule below.typedef-ideal-po*
    [*OF type-definition-alg-defl below-alg-defl-def*])

instance *alg-defl* :: (*profinite*) *cpo*
by (*rule below.typedef-ideal-cpo*
    [*OF type-definition-alg-defl below-alg-defl-def*])

lemma *Rep-alg-defl-lub*:

chain Y $\Longrightarrow$ *Rep-alg-defl* ($\bigsqcup i.\ Y\ i$) = ($\bigcup i.$ *Rep-alg-defl* (*Y i*))
by (*rule below.typedef-ideal-rep-contlub*
[*OF type-definition-alg-defl below-alg-defl-def*])

lemma *ideal-Rep-alg-defl*: *below.ideal* (*Rep-alg-defl xs*)
by (*rule Rep-alg-defl* [*unfolded mem-Collect-eq*])

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

lemma *Rep-alg-defl-principal*:
Rep-alg-defl (*alg-defl-principal t*) = {*u*. *u* $\sqsubseteq$ *t*}
unfolding *alg-defl-principal-def*
by (*simp add*: *Abs-alg-defl-inverse below.ideal-principal*)

interpretation *alg-defl*:
ideal-completion below fd-take alg-defl-principal Rep-alg-defl
apply *default*
apply (*rule ideal-Rep-alg-defl*)
apply (*erule Rep-alg-defl-lub*)
apply (*rule Rep-alg-defl-principal*)
apply (*simp only*: *below-alg-defl-def*)
done

Algebraic deflations are pointed

lemma *finite-deflation-UU*: *finite-deflation* $\bot$
by *default simp-all*

lemma *alg-defl-minimal*:
alg-defl-principal (*Abs-fin-defl* $\bot$) $\sqsubseteq$ *x*
apply (*induct x rule*: *alg-defl.principal-induct*, *simp*)
apply (*rule alg-defl.principal-mono*)
apply (*induct-tac a*)
apply (*rule Abs-fin-defl-mono*)
apply (*rule finite-deflation-UU*)
apply *simp*
apply (*rule minimal*)
done

instance *alg-defl* :: (*bifinite*) *pcpo*
by *intro-classes* (*fast intro*: *alg-defl-minimal*)

lemma *inst-alg-defl-pcpo*: $\bot$ = *alg-defl-principal* (*Abs-fin-defl* $\bot$)
by (*rule alg-defl-minimal* [*THEN UU-I*, *symmetric*])

Algebraic deflations are profinite

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

definition
approx-alg-defl-def: *approx* = *alg-defl.completion-approx*

instance
apply (*intro-classes*, *unfold approx-alg-defl-def*)
apply (*rule alg-defl.chain-completion-approx*)
apply (*rule alg-defl.lub-completion-approx*)
apply (*rule alg-defl.completion-approx-idem*)
apply (*rule alg-defl.finite-fixes-completion-approx*)
done

end

instance *alg-defl* :: (*bifinite*) *bifinite* **..**

lemma *approx-alg-defl-principal* [*simp*]:
approx n·(*alg-defl-principal t*) = *alg-defl-principal* (*fd-take n t*)
unfolding *approx-alg-defl-def*
by (*rule alg-defl.completion-approx-principal*)

lemma *approx-eq-alg-defl-principal*:
∃ *t*∈*Rep-alg-defl xs*. *approx n*·*xs* = *alg-defl-principal* (*fd-take n t*)
unfolding *approx-alg-defl-def*
by (*rule alg-defl.completion-approx-eq-principal*)

23.5 Applying algebraic deflations

definition
cast :: *'a alg-defl* → *'a* → *'a*
where
cast = *alg-defl.basis-fun Rep-fin-defl*

lemma *cast-alg-defl-principal*:
cast·(*alg-defl-principal a*) = *Rep-fin-defl a*
unfolding *cast-def*
apply (*rule alg-defl.basis-fun-principal*)
apply (*simp only*: *below-fin-defl-def*)
done

lemma *deflation-cast*: *deflation* (*cast*·*d*)
apply (*induct d rule*: *alg-defl.principal-induct*)
apply (*rule adm-subst* [*OF - adm-deflation*], *simp*)
apply (*simp add*: *cast-alg-defl-principal*)
apply (*rule finite-deflation-imp-deflation*)
apply (*rule finite-deflation-Rep-fin-defl*)
done

lemma *finite-deflation-cast*:

compact d ⟹ *finite-deflation* (*cast*·*d*)
apply (*drule alg-defl.compact-imp-principal*, *clarify*)
apply (*simp add*: *cast-alg-defl-principal*)
apply (*rule finite-deflation-Rep-fin-defl*)
done

interpretation *cast*: *deflation cast*·*d*
by (*rule deflation-cast*)

declare *cast.idem* [*simp*]

lemma *cast-approx*: *cast*·(*approx n*·*A*) = *defl-approx n* (*cast*·*A*)
apply (*rule alg-defl.principal-induct*)
apply (*rule adm-eq*)
apply *simp*
apply (*simp add*: *cont2cont-defl-approx cast.below*)
apply (*simp only*: *approx-alg-defl-principal*)
apply (*simp only*: *cast-alg-defl-principal*)
apply (*simp only*: *Rep-fin-defl-fd-take*)
done

lemma *cast-approx-fixed-iff*:
cast·(*approx i*·*A*)·*x* = *x* ⟷ *approx i*·*x* = *x* ∧ *cast*·*A*·*x* = *x*
apply (*simp only*: *cast-approx*)
apply (*rule defl-approx-fixed-iff*)
apply (*rule deflation-cast*)
done

lemma *defl-approx-cast*: *defl-approx i* (*cast*·*A*) = *cast*·(*approx i*·*A*)
by (*rule cast-approx* [*symmetric*])

lemma *cast-below-imp-below*: *cast*·*A* ⊑ *cast*·*B* ⟹ *A* ⊑ *B*
apply (*rule profinite-below-ext*)
apply (*drule-tac i*=*i* **in** *defl-approx-below*)
apply (*rule deflation-cast*)
apply (*rule deflation-cast*)
apply (*simp only*: *defl-approx-cast*)
apply (*cut-tac x*=*approx i*·*A* **in** *alg-defl.compact-imp-principal*)
apply (*rule compact-approx*)
apply (*cut-tac x*=*approx i*·*B* **in** *alg-defl.compact-imp-principal*)
apply (*rule compact-approx*)
apply *clarsimp*
apply (*simp add*: *cast-alg-defl-principal*)
apply (*simp add*: *below-fin-defl-def*)
done

lemma *cast-eq-imp-eq*: *cast*·*A* = *cast*·*B* ⟹ *A* = *B*
by (*simp add*: *below-antisym cast-below-imp-below*)

lemma *cast-strict1* [*simp*]: $cast{\cdot}\bot = \bot$
apply (*subst inst-alg-defl-pcpo*)
apply (*subst cast-alg-defl-principal*)
apply (*rule Abs-fin-defl-inverse*)
apply (*simp add*: *finite-deflation-UU*)
done

lemma *cast-strict2* [*simp*]: $cast{\cdot}A{\cdot}\bot = \bot$
by (*rule cast.below* [*THEN UU-I*])

23.6 Deflation membership relation

definition
in-deflation :: *'a*::*profinite* ⇒ *'a alg-defl* ⇒ *bool* (**infixl** ::: *50*)
where
$x ::: A \longleftrightarrow cast{\cdot}A{\cdot}x = x$

lemma *adm-in-deflation*: *adm* $(\lambda x.\ x ::: A)$
unfolding *in-deflation-def* **by** *simp*

lemma *in-deflationI*: $cast{\cdot}A{\cdot}x = x \Longrightarrow x ::: A$
unfolding *in-deflation-def* **.**

lemma *cast-fixed*: $x ::: A \Longrightarrow cast{\cdot}A{\cdot}x = x$
unfolding *in-deflation-def* **.**

lemma *cast-in-deflation* [*simp*]: $cast{\cdot}A{\cdot}x ::: A$
unfolding *in-deflation-def* **by** (*rule cast.idem*)

lemma *bottom-in-deflation* [*simp*]: $\bot ::: A$
unfolding *in-deflation-def* **by** (*rule cast-strict2*)

lemma *subdeflationD*: $A \sqsubseteq B \Longrightarrow x ::: A \Longrightarrow x ::: B$
unfolding *in-deflation-def*
apply (*rule deflation.belowD*)
apply (*rule deflation-cast*)
apply (*erule monofun-cfun-arg*)
apply *assumption*
done

lemma *rev-subdeflationD*: $x ::: A \Longrightarrow A \sqsubseteq B \Longrightarrow x ::: B$
by (*rule subdeflationD*)

lemma *subdeflationI*: $(\bigwedge x.\ x ::: A \Longrightarrow x ::: B) \Longrightarrow A \sqsubseteq B$
apply (*rule cast-below-imp-below*)
apply (*rule cast.belowI*)
apply (*simp add*: *in-deflation-def*)
done

Identity deflation:

lemma *cast*·(⨆ *i*. *alg-defl-principal* (*Abs-fin-defl* (*approx i*)))·*x* = *x*
apply (*subst contlub-cfun-arg*)
apply (*rule chainI*)
apply (*rule alg-defl.principal-mono*)
apply (*rule Abs-fin-defl-mono*)
apply (*rule finite-deflation-approx*)
apply (*rule finite-deflation-approx*)
apply (*rule chainE*)
apply (*rule chain-approx*)
apply (*simp add*: *cast-alg-defl-principal*
Abs-fin-defl-inverse finite-deflation-approx)
done

23.7 Bifinite domains and algebraic deflations

This lemma says that if we have an ep-pair from a bifinite domain into a universal domain, then e oo p is an algebraic deflation.

lemma
assumes *ep-pair e p*
constrains *e* :: *'a*::*profinite* → *'b*::*profinite*
shows ∃ *d*. *cast*·*d* = *e oo p*
proof
interpret *ep-pair e p* **by** *fact*
let *?a* = λ*i*. *e oo approx i oo p*
have *a*: ⋀*i*. *finite-deflation* (*?a i*)
apply (*rule finite-deflation-e-d-p*)
apply (*rule finite-deflation-approx*)
done
let *?d* = ⨆ *i*. *alg-defl-principal* (*Abs-fin-defl* (*?a i*))
show *cast*·*?d* = *e oo p*
apply (*subst contlub-cfun-arg*)
apply (*rule chainI*)
apply (*rule alg-defl.principal-mono*)
apply (*rule Abs-fin-defl-mono* [*OF a a*])
apply (*rule chainE*, *simp*)
apply (*subst cast-alg-defl-principal*)
apply (*simp add*: *Abs-fin-defl-inverse a*)
apply (*simp add*: *expand-cfun-eq lub-distribs*)
done
qed

This lemma says that if we have an ep-pair from a cpo into a bifinite domain, and e oo p is an algebraic deflation, then the cpo is bifinite.

lemma
assumes *ep-pair e p*
constrains *e* :: *'a*::*cpo* → *'b*::*profinite*
assumes *d*: ⋀*x*. *cast*·*d*·*x* = *e*·(*p*·*x*)
obtains *a* :: *nat* ⇒ *'a* → *'a* **where**

$\bigwedge i.$ *finite-deflation* $(a\ i)$
$(\bigsqcup i.\ a\ i) = ID$
proof
interpret *ep-pair e p* **by** *fact*
let *?a* $= \lambda i.$ *p oo cast*·(*approx i*·*d*) *oo e*
show $\bigwedge i.$ *finite-deflation* (*?a i*)
apply (*rule finite-deflation-p-d-e*)
apply (*rule finite-deflation-cast*)
apply (*rule compact-approx*)
apply (*rule below-eq-trans* [*OF - d*])
apply (*rule monofun-cfun-fun*)
apply (*rule monofun-cfun-arg*)
apply (*rule approx-below*)
done
show $(\bigsqcup i.$ *?a i*$) = ID$
apply (*rule ext-cfun*, *simp*)
apply (*simp add*: *lub-distribs*)
apply (*simp add*: *d*)
done
qed

end

24 CompactBasis: Compact bases of domains

theory *CompactBasis*
imports *Completion*
begin

24.1 Compact bases of bifinite domains

default-sort *profinite*

typedef (**open**) *'a compact-basis* $= \{x::'a::profinite.\ compact\ x\}$
by (*fast intro*: *compact-approx*)

lemma *compact-Rep-compact-basis*: *compact* (*Rep-compact-basis a*)
by (*rule Rep-compact-basis* [*unfolded mem-Collect-eq*])

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

definition
compact-le-def:
$(op \sqsubseteq) \equiv (\lambda x\ y.$ *Rep-compact-basis x* $\sqsubseteq$ *Rep-compact-basis y*$)$

instance ..

end

instance *compact-basis* :: (*profinite*) *po*
by (*rule typedef-po*
  [*OF type-definition-compact-basis compact-le-def*])

Take function for compact basis

definition
  compact-take :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis* **where**
  compact-take = (*λn a. Abs-compact-basis* (*approx n·*(*Rep-compact-basis a*)))

lemma *Rep-compact-take*:
  Rep-compact-basis (*compact-take n a*) = *approx n·*(*Rep-compact-basis a*)
unfolding *compact-take-def*
by (*simp add*: *Abs-compact-basis-inverse*)

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

interpretation *compact-basis*:
  basis-take below compact-take
proof
  fix *n* :: *nat* **and** *a* :: *'a compact-basis*
  show *compact-take n a* ⊑ *a*
    unfolding *compact-le-def*
    by (*simp add*: *Rep-compact-take approx-below*)
next
  fix *n* :: *nat* **and** *a* :: *'a compact-basis*
  show *compact-take n* (*compact-take n a*) = *compact-take n a*
    by (*simp add*: *Rep-compact-basis-inject* [*symmetric*] *Rep-compact-take*)
next
  fix *n* :: *nat* **and** *a b* :: *'a compact-basis*
  assume *a* ⊑ *b* **thus** *compact-take n a* ⊑ *compact-take n b*
    unfolding *compact-le-def Rep-compact-take*
    by (*rule monofun-cfun-arg*)
next
  fix *n* :: *nat* **and** *a* :: *'a compact-basis*
  show ⋀*n a. compact-take n a* ⊑ *compact-take* (*Suc n*) *a*
    unfolding *compact-le-def Rep-compact-take*
    by (*rule chainE*, *simp*)
next
  fix *n* :: *nat*
  show *finite* (*range* (*compact-take n*))
    apply (*rule finite-imageD* [**where** *f*=*Rep-compact-basis*])
    apply (*rule finite-subset* [**where** *B*=*range* (*λx. approx n·x*)])
    apply (*clarsimp simp add*: *Rep-compact-take*)
    apply (*rule finite-range-approx*)
    apply (*rule inj-onI*, *simp add*: *Rep-compact-basis-inject*)
    done
next

```
  fix a :: 'a compact-basis
  show ∃ n. compact-take n a = a
    apply (simp add: Rep-compact-basis-inject [symmetric])
    apply (simp add: Rep-compact-take)
    apply (rule profinite-compact-eq-approx)
    apply (rule compact-Rep-compact-basis)
    done
qed
```

Ideal completion

```
definition
  approximants :: 'a ⇒ 'a compact-basis set where
  approximants = (λx. {a. Rep-compact-basis a ⊑ x})

interpretation compact-basis:
  ideal-completion below compact-take Rep-compact-basis approximants
proof
  fix w :: 'a
  show preorder.ideal below (approximants w)
  proof (rule below.idealI)
    show ∃ x. x ∈ approximants w
      unfolding approximants-def
      apply (rule-tac x=Abs-compact-basis (approx 0·w) in exI)
      apply (simp add: Abs-compact-basis-inverse approx-below)
      done
  next
    fix x y :: 'a compact-basis
    assume x ∈ approximants w y ∈ approximants w
    thus ∃ z ∈ approximants w. x ⊑ z ∧ y ⊑ z
      unfolding approximants-def
      apply simp
      apply (cut-tac a=x in compact-Rep-compact-basis)
      apply (cut-tac a=y in compact-Rep-compact-basis)
      apply (drule profinite-compact-eq-approx)
      apply (drule profinite-compact-eq-approx)
      apply (clarify, rename-tac i j)
      apply (rule-tac x=Abs-compact-basis (approx (max i j)·w) in exI)
      apply (simp add: compact-le-def)
      apply (simp add: Abs-compact-basis-inverse approx-below)
      apply (erule subst, erule subst)
      apply (simp add: monofun-cfun chain-mono [OF chain-approx])
      done
  next
    fix x y :: 'a compact-basis
    assume x ⊑ y y ∈ approximants w thus x ∈ approximants w
      unfolding approximants-def
      apply simp
      apply (simp add: compact-le-def)
      apply (erule (1) below-trans)
```

```
    done
  qed
next
  fix Y :: nat ⇒ 'a
  assume Y: chain Y
  show approximants (⨆ i. Y i) = (⋃ i. approximants (Y i))
    unfolding approximants-def
    apply safe
    apply (simp add: compactD2 [OF compact-Rep-compact-basis Y])
    apply (erule below-trans, rule is-ub-thelub [OF Y])
    done
next
  fix a :: 'a compact-basis
  show approximants (Rep-compact-basis a) = {b. b ⊑ a}
    unfolding approximants-def compact-le-def ..
next
  fix x y :: 'a
  assume approximants x ⊆ approximants y thus x ⊑ y
    apply (subgoal-tac (⨆ i. approx i·x) ⊑ y, simp)
    apply (rule admD, simp, simp)
    apply (drule-tac c=Abs-compact-basis (approx i·x) in subsetD)
    apply (simp add: approximants-def Abs-compact-basis-inverse approx-below)
    apply (simp add: approximants-def Abs-compact-basis-inverse)
    done
qed
```

minimal compact element

```
definition
  compact-bot :: 'a::bifinite compact-basis where
  compact-bot = Abs-compact-basis ⊥

lemma Rep-compact-bot: Rep-compact-basis compact-bot = ⊥
unfolding compact-bot-def by (simp add: Abs-compact-basis-inverse)

lemma compact-bot-minimal [simp]: compact-bot ⊑ a
unfolding compact-le-def Rep-compact-bot by simp
```

24.2 A compact basis for powerdomains

```
typedef 'a pd-basis =
  {S::'a::profinite compact-basis set. finite S ∧ S ≠ {}}
by (rule-tac x={arbitrary} in exI, simp)

lemma finite-Rep-pd-basis [simp]: finite (Rep-pd-basis u)
by (insert Rep-pd-basis [of u, unfolded pd-basis-def]) simp

lemma Rep-pd-basis-nonempty [simp]: Rep-pd-basis u ≠ {}
by (insert Rep-pd-basis [of u, unfolded pd-basis-def]) simp
```

unit and plus

definition
PDUnit :: *'a compact-basis* ⇒ *'a pd-basis* **where**
PDUnit = (λ*x*. *Abs-pd-basis* {*x*})

definition
PDPlus :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* ∪ *Rep-pd-basis u*)

lemma *Rep-PDUnit*:
Rep-pd-basis (*PDUnit x*) = {*x*}
unfolding *PDUnit-def* **by** (*rule Abs-pd-basis-inverse*) (*simp add*: *pd-basis-def*)

lemma *Rep-PDPlus*:
Rep-pd-basis (*PDPlus u v*) = *Rep-pd-basis u* ∪ *Rep-pd-basis v*
unfolding *PDPlus-def* **by** (*rule Abs-pd-basis-inverse*) (*simp add*: *pd-basis-def*)

lemma *PDUnit-inject* [*simp*]: (*PDUnit a* = *PDUnit b*) = (*a* = *b*)
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDUnit* **by** *simp*

lemma *PDPlus-assoc*: *PDPlus* (*PDPlus t u*) *v* = *PDPlus t* (*PDPlus u v*)
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-assoc*)

lemma *PDPlus-commute*: *PDPlus t u* = *PDPlus u t*
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-commute*)

lemma *PDPlus-absorb*: *PDPlus t t* = *t*
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-absorb*)

lemma *pd-basis-induct1*:
assumes *PDUnit*: ⋀*a*. *P* (*PDUnit a*)
assumes *PDPlus*: ⋀*a t*. *P t* ⟹ *P* (*PDPlus* (*PDUnit a*) *t*)
shows *P x*
apply (*induct x*, *unfold pd-basis-def*, *clarify*)
apply (*erule* (*1*) *finite-ne-induct*)
apply (*cut-tac a*=*x* **in** *PDUnit*)
apply (*simp add*: *PDUnit-def*)
apply (*drule-tac a*=*x* **in** *PDPlus*)
apply (*simp add*: *PDUnit-def PDPlus-def*
Abs-pd-basis-inverse [*unfolded pd-basis-def*])
done

lemma *pd-basis-induct*:
assumes *PDUnit*: ⋀*a*. *P* (*PDUnit a*)
assumes *PDPlus*: ⋀*t u*. ⟦*P t*; *P u*⟧ ⟹ *P* (*PDPlus t u*)
shows *P x*
apply (*induct x rule*: *pd-basis-induct1*)
apply (*rule PDUnit*, *erule PDPlus* [*OF PDUnit*])
done

fold-pd

definition
fold-pd ::
(*'a compact-basis* ⇒ *'b*::*type*) ⇒ (*'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a pd-basis* ⇒ *'b*
where *fold-pd g f t* = *fold1 f* (*g* ‘ *Rep-pd-basis t*)

lemma *fold-pd-PDUnit*:
assumes *class.ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDUnit x*) = *g x*
unfolding *fold-pd-def Rep-PDUnit* **by** *simp*

lemma *fold-pd-PDPlus*:
assumes *class.ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDPlus t u*) = *f* (*fold-pd g f t*) (*fold-pd g f u*)
proof −
interpret *ab-semigroup-idem-mult f* **by** *fact*
show *?thesis* **unfolding** *fold-pd-def Rep-PDPlus*
by (*simp add*: *image-Un fold1-Un2*)
qed

Take function for powerdomain basis

definition
pd-take :: *nat* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
pd-take n = (*λt*. *Abs-pd-basis* (*compact-take n* ‘ *Rep-pd-basis t*))

lemma *Rep-pd-take*:
Rep-pd-basis (*pd-take n t*) = *compact-take n* ‘ *Rep-pd-basis t*
unfolding *pd-take-def*
apply (*rule Abs-pd-basis-inverse*)
apply (*simp add*: *pd-basis-def*)
done

lemma *pd-take-simps* [*simp*]:
pd-take n (*PDUnit a*) = *PDUnit* (*compact-take n a*)
pd-take n (*PDPlus t u*) = *PDPlus* (*pd-take n t*) (*pd-take n u*)
apply (*simp-all add*: *Rep-pd-basis-inject* [*symmetric*])
apply (*simp-all add*: *Rep-pd-take Rep-PDUnit Rep-PDPlus image-Un*)
done

lemma *pd-take-idem*: *pd-take n* (*pd-take n t*) = *pd-take n t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-take*)
apply *simp*
done

lemma *finite-range-pd-take*: *finite* (*range* (*pd-take n*))
apply (*rule finite-imageD* [**where** *f*=*Rep-pd-basis*])
apply (*rule finite-subset* [**where** *B*=*Pow* (*range* (*compact-take n*))])
apply (*clarsimp simp add*: *Rep-pd-take*)
apply (*simp add*: *compact-basis.finite-range-take*)

apply (*rule inj-onI*, *simp add*: *Rep-pd-basis-inject*)
done

lemma *pd-take-covers*: $\exists\, n.$ *pd-take* $n\ t = t$
apply (*subgoal-tac* $\exists\, n.\ \forall\, m{\geq}n.$ *pd-take* $m\ t = t$, *fast*)
apply (*induct* t *rule*: *pd-basis-induct*)
apply (*cut-tac* $a{=}a$ **in** *compact-basis.take-covers*)
apply (*clarify*, *rule-tac* $x{=}n$ **in** *exI*)
apply (*clarify*, *simp*)
apply (*rule below-antisym*)
apply (*rule compact-basis.take-less*)
apply (*drule-tac* $a{=}a$ **in** *compact-basis.take-chain-le*)
apply *simp*
apply (*clarify*, *rename-tac* $i\ j$)
apply (*rule-tac* $x{=}max\ i\ j$ **in** *exI*, *simp*)
done

end

25 Universal: A universal bifinite domain

theory *Universal*
imports *CompactBasis Nat-Bijection*
begin

25.1 Basis datatype

types *ubasis* = *nat*

definition
  node :: *nat* ⇒ *ubasis* ⇒ *ubasis set* ⇒ *ubasis*
where
  node i a S = *Suc* (*prod-encode* (*i*, *prod-encode* (*a*, *set-encode S*)))

lemma *node-not-0* [*simp*]: *node i a S* ≠ *0*
unfolding *node-def* **by** *simp*

lemma *node-gt-0* [*simp*]: *0* < *node i a S*
unfolding *node-def* **by** *simp*

lemma *node-inject* [*simp*]:
  ⟦*finite S*; *finite T*⟧
    ⟹ *node i a S* = *node j b T* ⟷ *i* = *j* ∧ *a* = *b* ∧ *S* = *T*
unfolding *node-def* **by** (*simp add*: *prod-encode-eq set-encode-eq*)

lemma *node-gt0*: *i* < *node i a S*
unfolding *node-def less-Suc-eq-le*
by (*rule le-prod-encode-1*)

lemma *node-gt1*: $a < node\ i\ a\ S$
unfolding *node-def less-Suc-eq-le*
by (*rule order-trans* [*OF le-prod-encode-1 le-prod-encode-2*])

lemma *nat-less-power2*: $n < 2\hat{}n$
by (*induct n*) *simp-all*

lemma *node-gt2*: $[\![finite\ S;\ b \in S]\!] \Longrightarrow b < node\ i\ a\ S$
unfolding *node-def less-Suc-eq-le set-encode-def*
apply (*rule order-trans* [*OF - le-prod-encode-2*])
apply (*rule order-trans* [*OF - le-prod-encode-2*])
apply (*rule order-trans* [**where** *y*=*setsum* (*op* ^ *2*) {*b*}])
apply (*simp add*: *nat-less-power2* [*THEN order-less-imp-le*])
apply (*erule setsum-mono2*, *simp*, *simp*)
done

lemma *eq-prod-encode-pairI*:
[*fst* (*prod-decode x*) = *a*; *snd* (*prod-decode x*) = *b*] ⟹ *x* = *prod-encode* (*a*, *b*)
by (*erule subst*, *erule subst*, *simp*)

lemma *node-cases*:
assumes *1*: $x = 0 \Longrightarrow P$
assumes *2*: $\bigwedge i\ a\ S.\ [\![finite\ S;\ x = node\ i\ a\ S]\!] \Longrightarrow P$
shows *P*
apply (*cases x*)
apply (*erule 1*)
apply (*rule 2*)
apply (*rule finite-set-decode*)
apply (*simp add*: *node-def*)
apply (*rule eq-prod-encode-pairI* [*OF refl*])
apply (*rule eq-prod-encode-pairI* [*OF refl refl*])
done

lemma *node-induct*:
assumes *1*: *P 0*
assumes *2*: $\bigwedge i\ a\ S.\ [\![P\ a;\ finite\ S;\ \forall b{\in}S.\ P\ b]\!] \Longrightarrow P\ (node\ i\ a\ S)$
shows *P x*
apply (*induct x rule*: *nat-less-induct*)
apply (*case-tac n rule*: *node-cases*)
apply (*simp add*: *1*)
apply (*simp add*: *2 node-gt1 node-gt2*)
done

25.2 Basis ordering

inductive
ubasis-le :: *nat* ⇒ *nat* ⇒ *bool*
where

ubasis-le-refl: *ubasis-le a a*
| *ubasis-le-trans*:
⟦ubasis-le a b; ubasis-le b c⟧ ⟹ ubasis-le a c
| *ubasis-le-lower*:
finite S ⟹ ubasis-le a (node i a S)
| *ubasis-le-upper*:
⟦finite S; b ∈ S; ubasis-le a b⟧ ⟹ ubasis-le (node i a S) b

lemma *ubasis-le-minimal*: *ubasis-le 0 x*
apply (*induct x rule*: *node-induct*)
apply (*rule ubasis-le-refl*)
apply (*erule ubasis-le-trans*)
apply (*erule ubasis-le-lower*)
done

25.2.1 Generic take function

function
ubasis-until :: (*ubasis ⇒ bool*) ⇒ *ubasis ⇒ ubasis*
where
ubasis-until P 0 = 0
| *finite S ⟹ ubasis-until P (node i a S) =*
(*if P (node i a S) then node i a S else ubasis-until P a*)
apply *clarify*
apply (*rule-tac x=b* **in** *node-cases*)
apply *simp*
apply *simp*
apply *fast*
apply *simp*
apply *simp*
apply *simp*
done

termination *ubasis-until*
apply (*relation measure snd*)
apply (*rule wf-measure*)
apply (*simp add*: *node-gt1*)
done

lemma *ubasis-until*: *P 0 ⟹ P* (*ubasis-until P x*)
by (*induct x rule*: *node-induct*) *simp-all*

lemma *ubasis-until'*: *0 < ubasis-until P x ⟹ P* (*ubasis-until P x*)
by (*induct x rule*: *node-induct*) *auto*

lemma *ubasis-until-same*: *P x ⟹ ubasis-until P x = x*
by (*induct x rule*: *node-induct*) *simp-all*

lemma *ubasis-until-idem*:

P 0 $\Longrightarrow$ *ubasis-until P* (*ubasis-until P x*) = *ubasis-until P x*
by (*rule ubasis-until-same* [*OF ubasis-until*])

lemma *ubasis-until-0*:
$\forall x.\ x \neq 0 \longrightarrow \neg\ P\ x \Longrightarrow$ *ubasis-until P x* = *0*
by (*induct x rule*: *node-induct*) *simp-all*

lemma *ubasis-until-less*: *ubasis-le* (*ubasis-until P x*) *x*
apply (*induct x rule*: *node-induct*)
apply (*simp add*: *ubasis-le-refl*)
apply (*simp add*: *ubasis-le-refl*)
apply (*rule impI*)
apply (*erule ubasis-le-trans*)
apply (*erule ubasis-le-lower*)
done

lemma *ubasis-until-chain*:
assumes *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows *ubasis-le* (*ubasis-until P x*) (*ubasis-until Q x*)
apply (*induct x rule*: *node-induct*)
apply (*simp add*: *ubasis-le-refl*)
apply (*simp add*: *ubasis-le-refl*)
apply (*simp add*: *PQ*)
apply *clarify*
apply (*rule ubasis-le-trans*)
apply (*rule ubasis-until-less*)
apply (*erule ubasis-le-lower*)
done

lemma *ubasis-until-mono*:
assumes $\bigwedge i\ a\ S\ b.\ [\![$*finite S*; *P* (*node i a S*); *b* $\in$ *S*; *ubasis-le a b*$]\!] \Longrightarrow$ *P b*
shows *ubasis-le a b* $\Longrightarrow$ *ubasis-le* (*ubasis-until P a*) (*ubasis-until P b*)
proof (*induct set*: *ubasis-le*)
case (*ubasis-le-refl a*) **show** *?case* **by** (*rule ubasis-le.ubasis-le-refl*)
next
case (*ubasis-le-trans a b c*) **thus** *?case* **by** − (*rule ubasis-le.ubasis-le-trans*)
next
case (*ubasis-le-lower S a i*) **thus** *?case*
apply (*clarsimp simp add*: *ubasis-le-refl*)
apply (*rule ubasis-le-trans* [*OF ubasis-until-less*])
apply (*erule ubasis-le.ubasis-le-lower*)
done
next
case (*ubasis-le-upper S b a i*) **thus** *?case*
apply *clarsimp*
apply (*subst ubasis-until-same*)
apply (*erule* (*3*) *prems*)
apply (*erule* (*2*) *ubasis-le.ubasis-le-upper*)
done

qed

lemma *finite-range-ubasis-until*:
finite {*x*. *P x*} ⟹ *finite* (*range* (*ubasis-until P*))
apply (*rule finite-subset* [**where** *B*=*insert 0* {*x*. *P x*}])
apply (*clarsimp simp add*: *ubasis-until'*)
apply *simp*
done

25.2.2 Take function for *ubasis*

definition
ubasis-take :: *nat* ⇒ *ubasis* ⇒ *ubasis*
where
ubasis-take n = *ubasis-until* (λ*x*. *x* ≤ *n*)

lemma *ubasis-take-le*: *ubasis-take n x* ≤ *n*
unfolding *ubasis-take-def* **by** (*rule ubasis-until*, *rule le0*)

lemma *ubasis-take-same*: *x* ≤ *n* ⟹ *ubasis-take n x* = *x*
unfolding *ubasis-take-def* **by** (*rule ubasis-until-same*)

lemma *ubasis-take-idem*: *ubasis-take n* (*ubasis-take n x*) = *ubasis-take n x*
by (*rule ubasis-take-same* [*OF ubasis-take-le*])

lemma *ubasis-take-0* [*simp*]: *ubasis-take 0 x* = *0*
unfolding *ubasis-take-def* **by** (*simp add*: *ubasis-until-0*)

lemma *ubasis-take-less*: *ubasis-le* (*ubasis-take n x*) *x*
unfolding *ubasis-take-def* **by** (*rule ubasis-until-less*)

lemma *ubasis-take-chain*: *ubasis-le* (*ubasis-take n x*) (*ubasis-take* (*Suc n*) *x*)
unfolding *ubasis-take-def* **by** (*rule ubasis-until-chain*) *simp*

lemma *ubasis-take-mono*:
assumes *ubasis-le x y*
shows *ubasis-le* (*ubasis-take n x*) (*ubasis-take n y*)
unfolding *ubasis-take-def*
apply (*rule ubasis-until-mono* [*OF - prems*])
apply (*frule* (*2*) *order-less-le-trans* [*OF node-gt2*])
apply (*erule order-less-imp-le*)
done

lemma *finite-range-ubasis-take*: *finite* (*range* (*ubasis-take n*))
apply (*rule finite-subset* [**where** *B*={..*n*}])
apply (*simp add*: *subset-eq ubasis-take-le*)
apply *simp*
done

lemma *ubasis-take-covers*: $\exists\, n.\ \mathit{ubasis\text{-}take}\ n\ x = x$
apply (*rule exI* [**where** $x{=}x$])
apply (*simp add*: *ubasis-take-same*)
done

interpretation *udom*: *preorder ubasis-le*
apply *default*
apply (*rule ubasis-le-refl*)
apply (*erule* (*1*) *ubasis-le-trans*)
done

interpretation *udom*: *basis-take ubasis-le ubasis-take*
apply *default*
apply (*rule ubasis-take-less*)
apply (*rule ubasis-take-idem*)
apply (*erule ubasis-take-mono*)
apply (*rule ubasis-take-chain*)
apply (*rule finite-range-ubasis-take*)
apply (*rule ubasis-take-covers*)
done

25.3 Defining the universal domain by ideal completion

typedef (**open**) *udom* = $\{S.\ \mathit{udom.ideal}\ S\}$
by (*fast intro*: *udom.ideal-principal*)

instantiation *udom* :: *below*
begin

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

instance ..
end

instance *udom* :: *po*
by (*rule udom.typedef-ideal-po*
[*OF type-definition-udom below-udom-def*])

instance *udom* :: *cpo*
by (*rule udom.typedef-ideal-cpo*
[*OF type-definition-udom below-udom-def*])

lemma *Rep-udom-lub*:
$\mathit{chain}\ Y \Longrightarrow \mathit{Rep\text{-}udom}\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ \mathit{Rep\text{-}udom}\ (Y\ i))$
by (*rule udom.typedef-ideal-rep-contlub*
[*OF type-definition-udom below-udom-def*])

lemma *ideal-Rep-udom*: *udom.ideal* (*Rep-udom xs*)

by (*rule Rep-udom* [*unfolded mem-Collect-eq*])

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

lemma *Rep-udom-principal*:
Rep-udom (*udom-principal t*) = {*u*. *ubasis-le u t*}
unfolding *udom-principal-def*
by (*simp add*: *Abs-udom-inverse udom.ideal-principal*)

interpretation *udom*:
ideal-completion ubasis-le ubasis-take udom-principal Rep-udom
apply *unfold-locales*
apply (*rule ideal-Rep-udom*)
apply (*erule Rep-udom-lub*)
apply (*rule Rep-udom-principal*)
apply (*simp only*: *below-udom-def*)
done

Universal domain is pointed

lemma *udom-minimal*: *udom-principal 0* ⊑ *x*
apply (*induct x rule*: *udom.principal-induct*)
apply (*simp*, *simp add*: *ubasis-le-minimal*)
done

instance *udom* :: *pcpo*
by *intro-classes* (*fast intro*: *udom-minimal*)

lemma *inst-udom-pcpo*: ⊥ = *udom-principal 0*
by (*rule udom-minimal* [*THEN UU-I*, *symmetric*])

Universal domain is bifinite

instantiation *udom* :: *bifinite*
begin

definition
approx-udom-def: *approx* = *udom.completion-approx*

instance
apply (*intro-classes*, *unfold approx-udom-def*)
apply (*rule udom.chain-completion-approx*)
apply (*rule udom.lub-completion-approx*)
apply (*rule udom.completion-approx-idem*)
apply (*rule udom.finite-fixes-completion-approx*)
done

end

lemma *approx-udom-principal* [*simp*]:
  approx n·(*udom-principal x*) = *udom-principal* (*ubasis-take n x*)
unfolding *approx-udom-def*
by (*rule udom.completion-approx-principal*)

lemma *approx-eq-udom-principal*:
  ∃ *a*∈*Rep-udom x*. *approx n·x* = *udom-principal* (*ubasis-take n a*)
unfolding *approx-udom-def*
by (*rule udom.completion-approx-eq-principal*)

25.4 Universality of *udom*

default-sort *bifinite*

25.4.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
  fixes *A* :: *'a*::*po set*
  shows ⟦*finite A*; *A* ≠ {}⟧ ⟹ ∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*
proof (*induct rule*: *finite-ne-induct*)
  case (*singleton x*)
    show *?case* **by** *simp*
next
  case (*insert a A*)
  from ‹∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*›
  obtain *x* **where** *x*: *x* ∈ *A*
          and *x-eq*: ⋀*y*. ⟦*y* ∈ *A*; *x* ⊑ *y*⟧ ⟹ *x* = *y* **by** *fast*
  show *?case*
  proof (*intro bexI ballI impI*)
    fix *y*
    assume *y* ∈ *insert a A* **and** (*if x* ⊑ *a then a else x*) ⊑ *y*
    thus (*if x* ⊑ *a then a else x*) = *y*
      apply *auto*
      apply (*frule* (*1*) *below-trans*)
      apply (*frule* (*1*) *x-eq*)
      apply (*rule below-antisym*, *assumption*)
      apply *simp*
      apply (*erule* (*1*) *x-eq*)
      done
  next
    show (*if x* ⊑ *a then a else x*) ∈ *insert a A*
      by (*simp add*: *x*)
  qed
qed

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

lemma *choose-lemma*:
⟦*finite A*; *A* ≠ {}⟧ ⟹ *choose A* ∈ {*x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*}
unfolding *choose-def*
apply (*rule someI-ex*)
apply (*frule* (*1*) *finite-has-maximal*, *fast*)
done

lemma *maximal-choose*:
⟦*finite A*; *y* ∈ *A*; *choose A* ⊑ *y*⟧ ⟹ *choose A* = *y*
apply (*cases A* = {}, *simp*)
apply (*frule* (*1*) *choose-lemma*, *simp*)
done

lemma *choose-in*: ⟦*finite A*; *A* ≠ {}⟧ ⟹ *choose A* ∈ *A*
by (*frule* (*1*) *choose-lemma*, *simp*)

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

termination *choose-pos*
apply (*relation measure* (*card* ∘ *fst*), *simp*)
apply *clarsimp*
apply (*rule card-Diff1-less*)
apply *assumption*
apply (*erule choose-in*)
apply *clarsimp*
done

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

lemma *choose-pos-choose*: *finite A* ⟹ *choose-pos A* (*choose A*) = *0*
by (*simp add*: *choose-pos.simps*)

lemma *inj-on-choose-pos* [*OF refl*]:
⟦*card A* = *n*; *finite A*⟧ ⟹ *inj-on* (*choose-pos A*) *A*
apply (*induct n arbitrary*: *A*)
apply *simp*
apply (*case-tac A* = {}, *simp*)
apply (*frule* (*1*) *choose-in*)
apply (*rule inj-onI*)
apply (*drule-tac x*=*A* − {*choose A*} **in** *meta-spec*, *simp*)
apply (*simp add*: *choose-pos.simps*)
apply (*simp split*: *split-if-asm*)
apply (*erule* (*1*) *inj-onD*, *simp*, *simp*)

done

lemma *choose-pos-bounded* [*OF refl*]:
⟦*card A = n*; *finite A*; *x ∈ A*⟧ ⟹ *choose-pos A x < n*
apply (*induct n arbitrary*: *A*)
apply *simp*
apply (*case-tac A = {}*, *simp*)
apply (*frule* (*1*) *choose-in*)
apply (*subst choose-pos.simps*)
apply *simp*
done

lemma *choose-pos-lessD*:
⟦*choose-pos A x < choose-pos A y*; *finite A*; *x ∈ A*; *y ∈ A*⟧ ⟹ ¬ *x ⊑ y*
apply (*induct A x arbitrary*: *y rule*: *choose-pos.induct*)
apply *simp*
apply (*case-tac x = choose A*)
apply *simp*
apply (*rule notI*)
apply (*frule* (*2*) *maximal-choose*)
apply *simp*
apply (*case-tac y = choose A*)
apply (*simp add*: *choose-pos-choose*)
apply (*drule-tac x=y* **in** *meta-spec*)
apply *simp*
apply (*erule meta-mp*)
apply (*simp add*: *choose-pos.simps*)
done

25.4.2 Rank of basis elements

primrec
cb-take :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis*
where
cb-take 0 = (λ*x*. *compact-bot*)
| *cb-take* (*Suc n*) = *compact-take n*

lemma *cb-take-covers*: ∃ *n*. *cb-take n x = x*
apply (*rule exE* [*OF compact-basis.take-covers* [**where** *a=x*]])
apply (*rename-tac n*, *rule-tac x=Suc n* **in** *exI*, *simp*)
done

lemma *cb-take-less*: *cb-take n x ⊑ x*
by (*cases n*, *simp*, *simp add*: *compact-basis.take-less*)

lemma *cb-take-idem*: *cb-take n* (*cb-take n x*) = *cb-take n x*
by (*cases n*, *simp*, *simp add*: *compact-basis.take-take*)

lemma *cb-take-mono*: *x ⊑ y* ⟹ *cb-take n x ⊑ cb-take n y*

by (*cases n*, *simp*, *simp add*: *compact-basis.take-mono*)

lemma *cb-take-chain-le*: $m \leq n \Longrightarrow$ *cb-take m x* $\sqsubseteq$ *cb-take n x*
apply (*cases m*, *simp*)
apply (*cases n*, *simp*)
apply (*simp add*: *compact-basis.take-chain-le*)
done

lemma *range-const*: *range* $(\lambda x.\ c) = \{c\}$
by *auto*

lemma *finite-range-cb-take*: *finite* (*range* (*cb-take n*))
apply (*cases n*)
apply (*simp add*: *range-const*)
apply (*simp add*: *compact-basis.finite-range-take*)
done

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

lemma *compact-approx-rank*: *cb-take* (*rank x*) *x* = *x*
unfolding *rank-def*
apply (*rule LeastI-ex*)
apply (*rule cb-take-covers*)
done

lemma *rank-leD*: *rank x* $\leq n \Longrightarrow$ *cb-take n x* = *x*
apply (*rule below-antisym* [*OF cb-take-less*])
apply (*subst compact-approx-rank* [*symmetric*])
apply (*erule cb-take-chain-le*)
done

lemma *rank-leI*: *cb-take n x* = *x* $\Longrightarrow$ *rank x* $\leq n$
unfolding *rank-def* **by** (*rule Least-le*)

lemma *rank-le-iff*: *rank x* $\leq n \longleftrightarrow$ *cb-take n x* = *x*
by (*rule iffI* [*OF rank-leD rank-leI*])

lemma *rank-compact-bot* [*simp*]: *rank compact-bot* = *0*
using *rank-leI* [*of 0 compact-bot*] **by** *simp*

lemma *rank-eq-0-iff* [*simp*]: *rank x* = *0* $\longleftrightarrow$ *x* = *compact-bot*
using *rank-le-iff* [*of x 0*] **by** *auto*

definition
rank-le :: *'a compact-basis* $\Rightarrow$ *'a compact-basis set*
where

rank-le x = {*y*. *rank y* ≤ *rank x*}

definition
rank-lt :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-lt x = {*y*. *rank y* < *rank x*}

definition
rank-eq :: *'a compact-basis* ⇒ *'a compact-basis set*
where
rank-eq x = {*y*. *rank y* = *rank x*}

lemma *rank-eq-cong*: *rank x* = *rank y* ⟹ *rank-eq x* = *rank-eq y*
unfolding *rank-eq-def* **by** *simp*

lemma *rank-lt-cong*: *rank x* = *rank y* ⟹ *rank-lt x* = *rank-lt y*
unfolding *rank-lt-def* **by** *simp*

lemma *rank-eq-subset*: *rank-eq x* ⊆ *rank-le x*
unfolding *rank-eq-def rank-le-def* **by** *auto*

lemma *rank-lt-subset*: *rank-lt x* ⊆ *rank-le x*
unfolding *rank-lt-def rank-le-def* **by** *auto*

lemma *finite-rank-le*: *finite* (*rank-le x*)
unfolding *rank-le-def*
apply (*rule finite-subset* [**where** *B*=*range* (*cb-take* (*rank x*))])
apply *clarify*
apply (*rule range-eqI*)
apply (*erule rank-leD* [*symmetric*])
apply (*rule finite-range-cb-take*)
done

lemma *finite-rank-eq*: *finite* (*rank-eq x*)
by (*rule finite-subset* [*OF rank-eq-subset finite-rank-le*])

lemma *finite-rank-lt*: *finite* (*rank-lt x*)
by (*rule finite-subset* [*OF rank-lt-subset finite-rank-le*])

lemma *rank-lt-Int-rank-eq*: *rank-lt x* ∩ *rank-eq x* = {}
unfolding *rank-lt-def rank-eq-def rank-le-def* **by** *auto*

lemma *rank-lt-Un-rank-eq*: *rank-lt x* ∪ *rank-eq x* = *rank-le x*
unfolding *rank-lt-def rank-eq-def rank-le-def* **by** *auto*

25.4.3 Sequencing basis elements

definition
place :: *'a compact-basis* ⇒ *nat*

where
place x = card (rank-lt x) + choose-pos (rank-eq x) x

lemma *place-bounded*: *place x < card (rank-le x)*
unfolding *place-def*
apply (*rule ord-less-eq-trans*)
apply (*rule add-strict-left-mono*)
apply (*rule choose-pos-bounded*)
apply (*rule finite-rank-eq*)
apply (*simp add*: *rank-eq-def*)
apply (*subst card-Un-disjoint* [*symmetric*])
apply (*rule finite-rank-lt*)
apply (*rule finite-rank-eq*)
apply (*rule rank-lt-Int-rank-eq*)
apply (*simp add*: *rank-lt-Un-rank-eq*)
done

lemma *place-ge*: *card (rank-lt x) ≤ place x*
unfolding *place-def* **by** *simp*

lemma *place-rank-mono*:
fixes *x y* :: *'a compact-basis*
shows *rank x < rank y ⟹ place x < place y*
apply (*rule less-le-trans* [*OF place-bounded*])
apply (*rule order-trans* [*OF - place-ge*])
apply (*rule card-mono*)
apply (*rule finite-rank-lt*)
apply (*simp add*: *rank-le-def rank-lt-def subset-eq*)
done

lemma *place-eqD*: *place x = place y ⟹ x = y*
apply (*rule linorder-cases* [**where** *x=rank x* **and** *y=rank y*])
apply (*drule place-rank-mono*, *simp*)
apply (*simp add*: *place-def*)
apply (*rule inj-on-choose-pos* [**where** *A=rank-eq x*, *THEN inj-onD*])
apply (*rule finite-rank-eq*)
apply (*simp cong*: *rank-lt-cong rank-eq-cong*)
apply (*simp add*: *rank-eq-def*)
apply (*simp add*: *rank-eq-def*)
apply (*drule place-rank-mono*, *simp*)
done

lemma *inj-place*: *inj place*
by (*rule inj-onI*, *erule place-eqD*)

25.4.4 Embedding and projection on basis elements

definition
sub :: *'a compact-basis ⇒ 'a compact-basis*

where
sub x = (*case rank x of 0* ⇒ *compact-bot* | *Suc k* ⇒ *cb-take k x*)

lemma *rank-sub-less*: *x* ≠ *compact-bot* ⟹ *rank* (*sub x*) < *rank x*
unfolding *sub-def*
apply (*cases rank x*, *simp*)
apply (*simp add*: *less-Suc-eq-le*)
apply (*rule rank-leI*)
apply (*rule cb-take-idem*)
done

lemma *place-sub-less*: *x* ≠ *compact-bot* ⟹ *place* (*sub x*) < *place x*
apply (*rule place-rank-mono*)
apply (*erule rank-sub-less*)
done

lemma *sub-below*: *sub x* ⊑ *x*
unfolding *sub-def* **by** (*cases rank x*, *simp-all add*: *cb-take-less*)

lemma *rank-less-imp-below-sub*: ⟦*x* ⊑ *y*; *rank x* < *rank y*⟧ ⟹ *x* ⊑ *sub y*
unfolding *sub-def*
apply (*cases rank y*, *simp*)
apply (*simp add*: *less-Suc-eq-le*)
apply (*subgoal-tac cb-take nat x* ⊑ *cb-take nat y*)
apply (*simp add*: *rank-leD*)
apply (*erule cb-take-mono*)
done

function
basis-emb :: *'a compact-basis* ⇒ *ubasis*
where
basis-emb x = (*if x* = *compact-bot then 0 else*
node (*place x*) (*basis-emb* (*sub x*))
(*basis-emb* ‘ {*y*. *place y* < *place x* ∧ *x* ⊑ *y*}))
by *auto*

termination *basis-emb*
apply (*relation measure place*, *simp*)
apply (*simp add*: *place-sub-less*)
apply *simp*
done

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

lemma *basis-emb-compact-bot* [*simp*]: *basis-emb compact-bot* = *0*
by (*simp add*: *basis-emb.simps*)

lemma *fin1*: *finite* {*y*. *place y* < *place x* ∧ *x* ⊑ *y*}
apply (*subst Collect-conj-eq*)

apply (*rule finite-Int*)
apply (*rule disjI1*)
apply (*subgoal-tac finite* (*place* −‘ {*n*. *n* < *place x*}), *simp*)
apply (*rule finite-vimageI* [*OF - inj-place*])
apply (*simp add*: *lessThan-def* [*symmetric*])
done

lemma *fin2*: *finite* (*basis-emb* ‘ {*y*. *place y* < *place x* ∧ *x* ⊑ *y*})
by (*rule finite-imageI* [*OF fin1*])

lemma *rank-place-mono*:
⟦*place x* < *place y*; *x* ⊑ *y*⟧ ⟹ *rank x* < *rank y*
apply (*rule linorder-cases*, *assumption*)
apply (*simp add*: *place-def cong*: *rank-lt-cong rank-eq-cong*)
apply (*drule choose-pos-lessD*)
apply (*rule finite-rank-eq*)
apply (*simp add*: *rank-eq-def*)
apply (*simp add*: *rank-eq-def*)
apply *simp*
apply (*drule place-rank-mono*, *simp*)
done

lemma *basis-emb-mono*:
x ⊑ *y* ⟹ *ubasis-le* (*basis-emb x*) (*basis-emb y*)
proof (*induct max* (*place x*) (*place y*) *arbitrary*: *x y rule*: *less-induct*)
case *less*
show *?case* **proof** (*rule linorder-cases*)
assume *place x* < *place y*
then have *rank x* < *rank y*
using ‹*x* ⊑ *y*› **by** (*rule rank-place-mono*)
with ‹*place x* < *place y*› **show** *?case*
apply (*case-tac y* = *compact-bot*, *simp*)
apply (*simp add*: *basis-emb.simps* [*of y*])
apply (*rule ubasis-le-trans* [*OF - ubasis-le-lower* [*OF fin2*]])
apply (*rule less*)
apply (*simp add*: *less-max-iff-disj*)
apply (*erule place-sub-less*)
apply (*erule rank-less-imp-below-sub* [*OF* ‹*x* ⊑ *y*›])
done
next
assume *place x* = *place y*
hence *x* = *y* **by** (*rule place-eqD*)
thus *?case* **by** (*simp add*: *ubasis-le-refl*)
next
assume *place x* > *place y*
with ‹*x* ⊑ *y*› **show** *?case*
apply (*case-tac x* = *compact-bot*, *simp add*: *ubasis-le-minimal*)
apply (*simp add*: *basis-emb.simps* [*of x*])
apply (*rule ubasis-le-upper* [*OF fin2*], *simp*)

```
    apply (rule less)
     apply (simp add: less-max-iff-disj)
     apply (erule place-sub-less)
    apply (erule rev-below-trans)
    apply (rule sub-below)
    done
  qed
qed

lemma inj-basis-emb: inj basis-emb
 apply (rule inj-onI)
 apply (case-tac x = compact-bot)
  apply (case-tac [!] y = compact-bot)
    apply simp
   apply (simp add: basis-emb.simps)
  apply (simp add: basis-emb.simps)
 apply (simp add: basis-emb.simps)
 apply (simp add: fin2 inj-eq [OF inj-place])
done

definition
  basis-prj :: ubasis ⇒ 'a compact-basis
where
  basis-prj x = inv basis-emb
    (ubasis-until (λx. x ∈ range (basis-emb :: 'a compact-basis ⇒ ubasis)) x)

lemma basis-prj-basis-emb: ⋀x. basis-prj (basis-emb x) = x
unfolding basis-prj-def
 apply (subst ubasis-until-same)
  apply (rule rangeI)
 apply (rule inv-f-f)
 apply (rule inj-basis-emb)
done

lemma basis-prj-node:
  ⟦finite S; node i a S ∉ range (basis-emb :: 'a compact-basis ⇒ nat)⟧
    ⟹ basis-prj (node i a S) = (basis-prj a :: 'a compact-basis)
unfolding basis-prj-def by simp

lemma basis-prj-0: basis-prj 0 = compact-bot
apply (subst basis-emb-compact-bot [symmetric])
apply (rule basis-prj-basis-emb)
done

lemma node-eq-basis-emb-iff:
  finite S ⟹ node i a S = basis-emb x ⟷
    x ≠ compact-bot ∧ i = place x ∧ a = basis-emb (sub x) ∧
        S = basis-emb ` {y. place y < place x ∧ x ⊑ y}
apply (cases x = compact-bot, simp)
```

apply (*simp add*: *basis-emb.simps* [*of x*])
apply (*simp add*: *fin2*)
done

lemma *basis-prj-mono*: *ubasis-le a b* ⟹ *basis-prj a* ⊑ *basis-prj b*
proof (*induct a b rule*: *ubasis-le.induct*)
case (*ubasis-le-refl a*) **show** *?case* **by** (*rule below-refl*)
next
case (*ubasis-le-trans a b c*) **thus** *?case* **by** − (*rule below-trans*)
next
case (*ubasis-le-lower S a i*) **thus** *?case*
apply (*cases node i a S* ∈ *range* (*basis-emb* :: *'a compact-basis* ⇒ *nat*))
apply (*erule rangeE*, *rename-tac x*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*simp add*: *node-eq-basis-emb-iff*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*rule sub-below*)
apply (*simp add*: *basis-prj-node*)
done
next
case (*ubasis-le-upper S b a i*) **thus** *?case*
apply (*cases node i a S* ∈ *range* (*basis-emb* :: *'a compact-basis* ⇒ *nat*))
apply (*erule rangeE*, *rename-tac x*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*clarsimp simp add*: *node-eq-basis-emb-iff*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*simp add*: *basis-prj-node*)
done
qed

lemma *basis-emb-prj-less*: *ubasis-le* (*basis-emb* (*basis-prj x*)) *x*
unfolding *basis-prj-def*
apply (*subst f-inv-into-f* [**where** *f*=*basis-emb*])
apply (*rule ubasis-until*)
apply (*rule range-eqI* [**where** *x*=*compact-bot*])
apply *simp*
apply (*rule ubasis-until-less*)
done

hide-const (**open**)
node
choose
choose-pos
place
sub

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

definition

udom-emb :: *'a*::*bifinite* → *udom*
where
udom-emb = *compact-basis.basis-fun* (λ*x*. *udom-principal* (*basis-emb x*))

definition
udom-prj :: *udom* → *'a*::*bifinite*
where
udom-prj = *udom.basis-fun* (λ*x*. *Rep-compact-basis* (*basis-prj x*))

lemma *udom-emb-principal*:
udom-emb·(*Rep-compact-basis x*) = *udom-principal* (*basis-emb x*)
unfolding *udom-emb-def*
apply (*rule compact-basis.basis-fun-principal*)
apply (*rule udom.principal-mono*)
apply (*erule basis-emb-mono*)
done

lemma *udom-prj-principal*:
udom-prj·(*udom-principal x*) = *Rep-compact-basis* (*basis-prj x*)
unfolding *udom-prj-def*
apply (*rule udom.basis-fun-principal*)
apply (*rule compact-basis.principal-mono*)
apply (*erule basis-prj-mono*)
done

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
apply *default*
apply (*rule compact-basis.principal-induct*, *simp*)
apply (*simp add*: *udom-emb-principal udom-prj-principal*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*rule udom.principal-induct*, *simp*)
apply (*simp add*: *udom-emb-principal udom-prj-principal*)
apply (*rule basis-emb-prj-less*)
done

end

26 Domain-Aux: Domain package support

theory *Domain-Aux*
imports *Ssum Sprod Fixrec*
uses
(*Tools/Domain/domain-take-proofs.ML*)
begin

26.1 Continuous isomorphisms

A locale for continuous isomorphisms

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

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

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

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

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

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

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

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

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

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

by (*rule iso.abs-defin'* [*OF swap*])

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

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

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

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

lemma *casedist-rule*: $rep{\cdot}x = \bot \vee P \Longrightarrow x = \bot \vee P$
by (*simp add*: *rep-defined-iff*)

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

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

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

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

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

end

26.2 Proofs about take functions

This section contains lemmas that are used in a module that supports the domain isomorphism package; the module contains proofs related to take functions and the finiteness predicate.

lemma *deflation-abs-rep*:
  fixes *abs* **and** *rep* **and** *d*
  assumes *abs-iso*: $\bigwedge x.\ rep{\cdot}(abs{\cdot}x) = x$
  assumes *rep-iso*: $\bigwedge y.\ abs{\cdot}(rep{\cdot}y) = y$
  shows *deflation d* $\Longrightarrow$ *deflation* (*abs oo d oo rep*)
by (*rule ep-pair.deflation-e-d-p*) (*simp add*: *ep-pair.intro assms*)

lemma *deflation-chain-min*:
  assumes *chain*: *chain d*
  assumes *defl*: $\bigwedge n.$ *deflation* (*d n*)
  shows $d\ m{\cdot}(d\ n{\cdot}x) = d\ (min\ m\ n){\cdot}x$
proof (*rule linorder-le-cases*)
  assume $m \leq n$
  with *chain* **have** $d\ m \sqsubseteq d\ n$ **by** (*rule chain-mono*)
  then have $d\ m{\cdot}(d\ n{\cdot}x) = d\ m{\cdot}x$
    by (*rule deflation-below-comp1* [*OF defl defl*])
  moreover from ‹$m \leq n$› **have** $min\ m\ n = m$ **by** *simp*
  ultimately show *?thesis* **by** *simp*
next
  assume $n \leq m$
  with *chain* **have** $d\ n \sqsubseteq d\ m$ **by** (*rule chain-mono*)
  then have $d\ m{\cdot}(d\ n{\cdot}x) = d\ n{\cdot}x$
    by (*rule deflation-below-comp2* [*OF defl defl*])
  moreover from ‹$n \leq m$› **have** $min\ m\ n = n$ **by** *simp*
  ultimately show *?thesis* **by** *simp*
qed

lemma *lub-ID-take-lemma*:
  assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
  assumes $\bigwedge n.\ t\ n{\cdot}x = t\ n{\cdot}y$ **shows** $x = y$
proof −
  have $(\bigsqcup n.\ t\ n{\cdot}x) = (\bigsqcup n.\ t\ n{\cdot}y)$
    using *assms*(*3*) **by** *simp*
  then have $(\bigsqcup n.\ t\ n){\cdot}x = (\bigsqcup n.\ t\ n){\cdot}y$
    using *assms*(*1*) **by** (*simp add*: *lub-distribs*)
  then show $x = y$
    using *assms*(*2*) **by** *simp*
qed

lemma *lub-ID-reach*:
  assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
  shows $(\bigsqcup n.\ t\ n{\cdot}x) = x$
using *assms* **by** (*simp add*: *lub-distribs*)

lemma *lub-ID-take-induct*:
assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
assumes *adm P* **and** $\bigwedge n.\ P\ (t\ n{\cdot}x)$ **shows** $P\ x$
proof −
from ‹*chain t*› **have** *chain* $(\lambda n.\ t\ n{\cdot}x)$ **by** *simp*
from ‹*adm P*› *this* ‹$\bigwedge n.\ P\ (t\ n{\cdot}x)$› **have** $P\ (\bigsqcup n.\ t\ n{\cdot}x)$ **by** (*rule admD*)
with ‹*chain t*› ‹$(\bigsqcup n.\ t\ n) = ID$› **show** $P\ x$ **by** (*simp add*: *lub-distribs*)
qed

26.3 Finiteness

Let a "decisive" function be a deflation that maps every input to either itself or bottom. Then if a domain's take functions are all decisive, then all values in the domain are finite.

definition
decisive :: $('a{::}pcpo \to 'a) \Rightarrow bool$
where
$decisive\ d \longleftrightarrow (\forall x.\ d{\cdot}x = x \vee d{\cdot}x = \bot)$

lemma *decisiveI*: $(\bigwedge x.\ d{\cdot}x = x \vee d{\cdot}x = \bot) \Longrightarrow decisive\ d$
unfolding *decisive-def* **by** *simp*

lemma *decisive-cases*:
assumes *decisive d* **obtains** $d{\cdot}x = x \mid d{\cdot}x = \bot$
using *assms* **unfolding** *decisive-def* **by** *auto*

lemma *decisive-bottom*: $decisive\ \bot$
unfolding *decisive-def* **by** *simp*

lemma *decisive-ID*: *decisive ID*
unfolding *decisive-def* **by** *simp*

lemma *decisive-ssum-map*:
assumes *f*: *decisive f*
assumes *g*: *decisive g*
shows *decisive* (*ssum-map*·*f*·*g*)
apply (*rule decisiveI*, *rename-tac s*)
apply (*case-tac s*, *simp-all*)
apply (*rule-tac x*=*x* **in** *decisive-cases* [*OF f*], *simp-all*)
apply (*rule-tac x*=*y* **in** *decisive-cases* [*OF g*], *simp-all*)
done

lemma *decisive-sprod-map*:
assumes *f*: *decisive f*
assumes *g*: *decisive g*
shows *decisive* (*sprod-map*·*f*·*g*)
apply (*rule decisiveI*, *rename-tac s*)
apply (*case-tac s*, *simp-all*)
apply (*rule-tac x*=*x* **in** *decisive-cases* [*OF f*], *simp-all*)

apply (*rule-tac x=y* **in** *decisive-cases* [*OF g*], *simp-all*)
done

lemma *decisive-abs-rep*:
fixes *abs rep*
assumes *iso*: *iso abs rep*
assumes *d*: *decisive d*
shows *decisive* (*abs oo d oo rep*)
apply (*rule decisiveI*)
apply (*rule-tac x=rep·x* **in** *decisive-cases* [*OF d*])
apply (*simp add*: *iso.rep-iso* [*OF iso*])
apply (*simp add*: *iso.abs-strict* [*OF iso*])
done

lemma *lub-ID-finite*:
assumes *chain*: *chain d*
assumes *lub*: $(\bigsqcup n.\ d\ n) = ID$
assumes *decisive*: $\bigwedge n.$ *decisive* $(d\ n)$
shows $\exists n.\ d\ n{\cdot}x = x$
proof −
have *1*: *chain* $(\lambda n.\ d\ n{\cdot}x)$ **using** *chain* **by** *simp*
have *2*: $(\bigsqcup n.\ d\ n{\cdot}x) = x$ **using** *chain lub* **by** (*rule lub-ID-reach*)
have $\forall n.\ d\ n{\cdot}x = x \vee d\ n{\cdot}x = \bot$
using *decisive* **unfolding** *decisive-def* **by** *simp*
hence *range* $(\lambda n.\ d\ n{\cdot}x) \subseteq \{x, \bot\}$
by *auto*
hence *finite* (*range* $(\lambda n.\ d\ n{\cdot}x)$)
by (*rule finite-subset*, *simp*)
with *1* **have** *finite-chain* $(\lambda n.\ d\ n{\cdot}x)$
by (*rule finite-range-imp-finch*)
then have $\exists n.\ (\bigsqcup n.\ d\ n{\cdot}x) = d\ n{\cdot}x$
unfolding *finite-chain-def* **by** (*auto simp add*: *maxinch-is-thelub*)
with *2* **show** $\exists n.\ d\ n{\cdot}x = x$ **by** (*auto elim*: *sym*)
qed

lemma *lub-ID-finite-take-induct*:
assumes *chain d* **and** $(\bigsqcup n.\ d\ n) = ID$ **and** $\bigwedge n.$ *decisive* $(d\ n)$
shows $(\bigwedge n.\ P\ (d\ n{\cdot}x)) \Longrightarrow P\ x$
using *lub-ID-finite* [*OF assms*] **by** *metis*

26.4 ML setup

use *Tools/Domain/domain-take-proofs.ML*

end

27 Representable: Representable Types

theory *Representable*
imports *Algebraic Universal Ssum Sprod One Fixrec Domain-Aux*
uses
(*Tools/repdef.ML*)
(*Tools/Domain/domain-isomorphism.ML*)
begin

27.1 Class of representable types

Overloaded embedding and projection functions between a representable type and the universal domain.

class *rep* = *bifinite* +
fixes *emb* :: 'a::*pcpo* → *udom*
fixes *prj* :: *udom* → 'a::*pcpo*
assumes *ep-pair-emb-prj*: *ep-pair emb prj*

interpretation *rep*!:
pcpo-ep-pair
emb :: 'a::*rep* → *udom*
prj :: *udom* → 'a::*rep*
unfolding *pcpo-ep-pair-def*
by (*rule ep-pair-emb-prj*)

lemmas *emb-inverse* = *rep.e-inverse*
lemmas *emb-prj-below* = *rep.e-p-below*
lemmas *emb-eq-iff* = *rep.e-eq-iff*
lemmas *emb-strict* = *rep.e-strict*
lemmas *prj-strict* = *rep.p-strict*

27.2 Making *rep* the default class

From now on, free type variables are assumed to be in class *rep*, unless specified otherwise.

default-sort *rep*

27.3 Representations of types

A TypeRep is an algebraic deflation over the universe of values.

types *TypeRep* = *udom alg-defl*
translations (*type*) *TypeRep* ↽ (*type*) *udom alg-defl*

definition
Rep-of :: 'a::*rep itself* ⇒ *TypeRep*
where
Rep-of TYPE('a::*rep*) =

```
    (⨆ i. alg-defl-principal (Abs-fin-defl
      (emb oo (approx i :: 'a → 'a) oo prj)))

syntax -REP :: type ⇒ TypeRep  ((1REP/(1'(-'))))
translations REP('t) ⇌ CONST Rep-of TYPE('t)

lemma cast-REP:
  cast·REP('a::rep) = (emb::'a → udom) oo (prj::udom → 'a)
proof −
  let ?a = λi. emb oo approx i oo (prj::udom → 'a)
  have a: ⋀i. finite-deflation (?a i)
    apply (rule rep.finite-deflation-e-d-p)
    apply (rule finite-deflation-approx)
    done
  show ?thesis
    unfolding Rep-of-def
    apply (subst contlub-cfun-arg)
    apply (rule chainI)
    apply (rule alg-defl.principal-mono)
    apply (rule Abs-fin-defl-mono [OF a a])
    apply (rule chainE, simp)
    apply (subst cast-alg-defl-principal)
    apply (simp add: Abs-fin-defl-inverse a)
    apply (simp add: expand-cfun-eq lub-distribs)
    done
qed

lemma emb-prj: emb·((prj·x)::'a::rep) = cast·REP('a)·x
by (simp add: cast-REP)

lemma in-REP-iff:
  x ::: REP('a::rep) ⟷ emb·((prj·x)::'a) = x
by (simp add: in-deflation-def cast-REP)

lemma prj-inverse:
  x ::: REP('a::rep) ⟹ emb·((prj·x)::'a) = x
by (simp only: in-REP-iff)

lemma emb-in-REP [simp]:
  emb·(x::'a::rep) ::: REP('a)
by (simp add: in-REP-iff)
```

27.4 Coerce operator

```
definition coerce :: 'a → 'b
where coerce = prj oo emb

lemma beta-coerce: coerce·x = prj·(emb·x)
by (simp add: coerce-def)
```

lemma *prj-emb*: $prj{\cdot}(emb{\cdot}x) = coerce{\cdot}x$
by (*simp add*: *coerce-def*)

lemma *coerce-strict* [*simp*]: $coerce{\cdot}\bot = \bot$
by (*simp add*: *coerce-def*)

lemma *coerce-eq-ID* [*simp*]: $(coerce :: 'a \to 'a) = ID$
by (*rule ext-cfun*, *simp add*: *beta-coerce*)

lemma *emb-coerce*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow emb{\cdot}((coerce{::}'a \to 'b){\cdot}x) = emb{\cdot}x$
apply (*simp add*: *beta-coerce*)
apply (*rule prj-inverse*)
apply (*erule subdeflationD*)
apply (*rule emb-in-REP*)
done

lemma *coerce-prj*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow (coerce{::}'b \to 'a){\cdot}(prj{\cdot}x) = prj{\cdot}x$
apply (*simp add*: *coerce-def*)
apply (*rule emb-eq-iff* [*THEN iffD1*])
apply (*simp only*: *emb-prj*)
apply (*rule deflation-below-comp1*)
apply (*rule deflation-cast*)
apply (*rule deflation-cast*)
apply (*erule monofun-cfun-arg*)
done

lemma *coerce-coerce* [*simp*]:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow coerce{\cdot}((coerce{::}'a \to 'b){\cdot}x) = coerce{\cdot}x$
by (*simp add*: *beta-coerce prj-inverse subdeflationD*)

lemma *coerce-inverse*:
$emb{\cdot}(x{::}'a) ::: REP('b) \Longrightarrow coerce{\cdot}(coerce{\cdot}x :: 'b) = x$
by (*simp only*: *beta-coerce prj-inverse emb-inverse*)

lemma *coerce-type*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow emb{\cdot}((coerce{::}'a \to 'b){\cdot}x) ::: REP('a)$
by (*simp add*: *beta-coerce prj-inverse subdeflationD*)

lemma *ep-pair-coerce*:
$REP('a) \sqsubseteq REP('b)$
$\Longrightarrow ep\text{-}pair\ (coerce{::}'a \to 'b)\ (coerce{::}'b \to 'a)$
apply (*rule ep-pair.intro*)

apply *simp*
apply (*simp only*: *beta-coerce*)
apply (*rule below-trans*)
apply (*rule monofun-cfun-arg*)
apply (*rule emb-prj-below*)
apply *simp*
done

Isomorphism lemmas used internally by the domain package:

lemma *domain-abs-iso*:
fixes *abs* **and** *rep*
assumes *REP*: *REP*('*b*) = *REP*('*a*)
assumes *abs-def*: *abs* ≡ (*coerce* :: '*a* → '*b*)
assumes *rep-def*: *rep* ≡ (*coerce* :: '*b* → '*a*)
shows *rep*·(*abs*·*x*) = *x*
unfolding *abs-def rep-def* **by** (*simp add*: *REP*)

lemma *domain-rep-iso*:
fixes *abs* **and** *rep*
assumes *REP*: *REP*('*b*) = *REP*('*a*)
assumes *abs-def*: *abs* ≡ (*coerce* :: '*a* → '*b*)
assumes *rep-def*: *rep* ≡ (*coerce* :: '*b* → '*a*)
shows *abs*·(*rep*·*x*) = *x*
unfolding *abs-def rep-def* **by** (*simp add*: *REP* [*symmetric*])

27.5 Proving a subtype is representable

Temporarily relax type constraints for *approx*, *emb*, and *prj*.

setup ⟪ *Sign.add-const-constraint*
(@{*const-name approx*}, *SOME* @{*typ nat* ⇒ '*a*::*cpo* → '*a*}) ⟫

setup ⟪ *Sign.add-const-constraint*
(@{*const-name emb*}, *SOME* @{*typ* '*a*::*pcpo* → *udom*}) ⟫

setup ⟪ *Sign.add-const-constraint*
(@{*const-name prj*}, *SOME* @{*typ udom* → '*a*::*pcpo*}) ⟫

definition
repdef-approx ::
('*a*::*pcpo* ⇒ *udom*) ⇒ (*udom* ⇒ '*a*) ⇒ *udom alg-defl* ⇒ *nat* ⇒ '*a* → '*a*
where
repdef-approx Rep Abs t = (λ*i*. Λ *x*. *Abs* (*cast*·(*approx i*·*t*)·(*Rep x*)))

lemma *typedef-rep-class*:
fixes *Rep* :: '*a*::*pcpo* ⇒ *udom*
fixes *Abs* :: *udom* ⇒ '*a*::*pcpo*
fixes *t* :: *TypeRep*
assumes *type*: *type-definition Rep Abs* {*x*. *x* ::: *t*}
assumes *below*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*

assumes *emb*: $emb \equiv (\Lambda\ x.\ Rep\ x)$
assumes *prj*: $prj \equiv (\Lambda\ x.\ Abs\ (cast{\cdot}t{\cdot}x))$
assumes *approx*: $(approx :: nat \Rightarrow 'a \to 'a) \equiv repdef\text{-}approx\ Rep\ Abs\ t$
shows *OFCLASS*('*a*, *rep-class*)
proof
have *adm*: $adm\ (\lambda x.\ x \in \{x.\ x ::: t\})$
by (*simp add*: *adm-in-deflation*)
have *emb-beta*: $\bigwedge x.\ emb{\cdot}x = Rep\ x$
unfolding *emb*
apply (*rule beta-cfun*)
apply (*rule typedef-cont-Rep* [*OF type below adm*])
done
have *prj-beta*: $\bigwedge y.\ prj{\cdot}y = Abs\ (cast{\cdot}t{\cdot}y)$
unfolding *prj*
apply (*rule beta-cfun*)
apply (*rule typedef-cont-Abs* [*OF type below adm*])
apply *simp-all*
done
have *cast-cast-approx*:
$\bigwedge i\ x.\ cast{\cdot}t{\cdot}(cast{\cdot}(approx\ i{\cdot}t){\cdot}x) = cast{\cdot}(approx\ i{\cdot}t){\cdot}x$
apply (*rule cast-fixed*)
apply (*rule subdeflationD*)
apply (*rule approx.below*)
apply (*rule cast-in-deflation*)
done
have *approx'*:
$approx = (\lambda i.\ \Lambda(x::'a).\ prj{\cdot}(cast{\cdot}(approx\ i{\cdot}t){\cdot}(emb{\cdot}x)))$
unfolding *approx repdef-approx-def*
apply (*subst cast-cast-approx* [*symmetric*])
apply (*simp add*: *prj-beta* [*symmetric*] *emb-beta* [*symmetric*])
done
have *emb-in-deflation*: $\bigwedge x::'a.\ emb{\cdot}x ::: t$
using *type-definition.Rep* [*OF type*]
by (*simp add*: *emb-beta*)
have *prj-emb*: $\bigwedge x::'a.\ prj{\cdot}(emb{\cdot}x) = x$
unfolding *prj-beta*
apply (*simp add*: *cast-fixed* [*OF emb-in-deflation*])
apply (*simp add*: *emb-beta type-definition.Rep-inverse* [*OF type*])
done
have *emb-prj*: $\bigwedge y.\ emb{\cdot}(prj{\cdot}y :: 'a) = cast{\cdot}t{\cdot}y$
unfolding *prj-beta emb-beta*
by (*simp add*: *type-definition.Abs-inverse* [*OF type*])
show *ep-pair* $(emb :: 'a \to udom)$ *prj*
apply *default*
apply (*simp add*: *prj-emb*)
apply (*simp add*: *emb-prj cast.below*)
done
show *chain* $(approx :: nat \Rightarrow 'a \to 'a)$
unfolding *approx'* **by** *simp*

```
  show ⋀x::'a. (⨆ i. approx i·x) = x
    unfolding approx'
    apply (simp add: lub-distribs)
    apply (subst cast-fixed [OF emb-in-deflation])
    apply (rule prj-emb)
    done
  show ⋀(i::nat) (x::'a). approx i·(approx i·x) = approx i·x
    unfolding approx'
    apply simp
    apply (simp add: emb-prj)
    apply (simp add: cast-cast-approx)
    done
  show ⋀i::nat. finite {x::'a. approx i·x = x}
    apply (rule-tac B=(λx. prj·x) ` {x. cast·(approx i·t)·x = x}
          in finite-subset)
    apply (clarsimp simp add: approx')
    apply (drule-tac f=λx. emb·x in arg-cong)
    apply (rule image-eqI)
    apply (rule prj-emb [symmetric])
    apply (simp add: emb-prj)
    apply (simp add: cast-cast-approx)
    apply (rule finite-imageI)
    apply (simp add: cast-approx-fixed-iff)
    apply (simp add: Collect-conj-eq)
    apply (simp add: finite-fixes-approx)
    done
qed
```

Restore original typing constraints

```
setup ⟪ Sign.add-const-constraint
  (@{const-name approx}, SOME @{typ nat ⇒ 'a::profinite → 'a}) ⟫

setup ⟪ Sign.add-const-constraint
  (@{const-name emb}, SOME @{typ 'a::rep → udom}) ⟫

setup ⟪ Sign.add-const-constraint
  (@{const-name prj}, SOME @{typ udom → 'a::rep}) ⟫

lemma typedef-REP:
  fixes Rep :: 'a::rep ⇒ udom
  fixes Abs :: udom ⇒ 'a::rep
  fixes t :: TypeRep
  assumes type: type-definition Rep Abs {x. x ::: t}
  assumes below: op ⊑ ≡ λx y. Rep x ⊑ Rep y
  assumes emb: emb ≡ (Λ x. Rep x)
  assumes prj: prj ≡ (Λ x. Abs (cast·t·x))
  shows REP('a) = t
proof −
  have adm: adm (λx. x ∈ {x. x ::: t})
```

```
    by (simp add: adm-in-deflation)
  have emb-beta: ⋀x. emb·x = Rep x
    unfolding emb
    apply (rule beta-cfun)
    apply (rule typedef-cont-Rep [OF type below adm])
    done
  have prj-beta: ⋀y. prj·y = Abs (cast·t·y)
    unfolding prj
    apply (rule beta-cfun)
    apply (rule typedef-cont-Abs [OF type below adm])
    apply simp-all
    done
  have emb-in-deflation: ⋀x::'a. emb·x ::: t
    using type-definition.Rep [OF type]
    by (simp add: emb-beta)
  have prj-emb: ⋀x::'a. prj·(emb·x) = x
    unfolding prj-beta
    apply (simp add: cast-fixed [OF emb-in-deflation])
    apply (simp add: emb-beta type-definition.Rep-inverse [OF type])
    done
  have emb-prj: ⋀y. emb·(prj·y :: 'a) = cast·t·y
    unfolding prj-beta emb-beta
    by (simp add: type-definition.Abs-inverse [OF type])
  show REP('a) = t
    apply (rule cast-eq-imp-eq, rule ext-cfun)
    apply (simp add: cast-REP emb-prj)
    done
qed

lemma adm-mem-Collect-in-deflation: adm (λx. x ∈ {x. x ::: A})
unfolding mem-Collect-eq by (rule adm-in-deflation)

use Tools/repdef.ML
```

27.6 Instances of class *rep*

27.6.1 Universal Domain

The Universal Domain itself is trivially representable.

```
instantiation udom :: rep
begin

definition emb-udom-def [simp]: emb = (ID :: udom → udom)
definition prj-udom-def [simp]: prj = (ID :: udom → udom)

instance
 apply (intro-classes)
 apply (simp-all add: ep-pair.intro)
done
```

end

27.6.2 Lifted types

instantiation *lift* :: (*countable*) *rep*
begin

definition *emb-lift-def*:
emb = *udom-emb oo* (*FLIFT x*. *Def* (*to-nat x*))

definition *prj-lift-def*:
prj = (*FLIFT n*. *if* (∃ *x*::'*a*::*countable*. *n* = *to-nat x*)
then Def (*THE x*::'*a*. *n* = *to-nat x*) *else* ⊥) *oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-lift-def prj-lift-def*)
apply (*rule ep-pair-comp* [*OF* - *ep-pair-udom*])
apply (*rule ep-pair.intro*)
apply (*case-tac x*, *simp*, *simp*)
apply (*case-tac y*, *simp*, *clarsimp*)
done

end

27.6.3 Representable type constructors

Functions between representable types are representable.

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

definition *emb-cfun-def*: *emb* = *udom-emb oo cfun-map*·*prj*·*emb*
definition *prj-cfun-def*: *prj* = *cfun-map*·*emb*·*prj oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-cfun-def prj-cfun-def*)
apply (*intro ep-pair-comp ep-pair-cfun-map ep-pair-emb-prj ep-pair-udom*)
done

end

Strict products of representable types are representable.

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

definition *emb-sprod-def*: *emb* = *udom-emb oo sprod-map*·*emb*·*emb*
definition *prj-sprod-def*: *prj* = *sprod-map*·*prj*·*prj oo udom-prj*

instance

apply (*intro-classes*, *unfold emb-sprod-def prj-sprod-def*)
apply (*intro ep-pair-comp ep-pair-sprod-map ep-pair-emb-prj ep-pair-udom*)
done

end

Strict sums of representable types are representable.

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

definition *emb-ssum-def*: *emb* = *udom-emb oo ssum-map·emb·emb*
definition *prj-ssum-def*: *prj* = *ssum-map·prj·prj oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-ssum-def prj-ssum-def*)
apply (*intro ep-pair-comp ep-pair-ssum-map ep-pair-emb-prj ep-pair-udom*)
done

end

Up of a representable type is representable.

instantiation *u* :: (*rep*) *rep*
begin

definition *emb-u-def*: *emb* = *udom-emb oo u-map·emb*
definition *prj-u-def*: *prj* = *u-map·prj oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-u-def prj-u-def*)
apply (*intro ep-pair-comp ep-pair-u-map ep-pair-emb-prj ep-pair-udom*)
done

end

Cartesian products of representable types are representable.

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

definition *emb-cprod-def*: *emb* = *udom-emb oo cprod-map·emb·emb*
definition *prj-cprod-def*: *prj* = *cprod-map·prj·prj oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-cprod-def prj-cprod-def*)
apply (*intro ep-pair-comp ep-pair-cprod-map ep-pair-emb-prj ep-pair-udom*)
done

end

27.7 Type combinators

definition
TypeRep-fun1 ::
((*udom* → *udom*) → (′*a* → ′*a*))
⇒ (*TypeRep* → *TypeRep*)
where
TypeRep-fun1 f =
alg-defl.basis-fun (λ*a*.
alg-defl-principal (
Abs-fin-defl (*udom-emb oo f*·(*Rep-fin-defl a*) *oo udom-prj*)))

definition
TypeRep-fun2 ::
((*udom* → *udom*) → (*udom* → *udom*) → (′*a* → ′*a*))
⇒ (*TypeRep* → *TypeRep* → *TypeRep*)
where
TypeRep-fun2 f =
alg-defl.basis-fun (λ*a*.
alg-defl.basis-fun (λ*b*.
alg-defl-principal (
Abs-fin-defl (*udom-emb oo*
f·(*Rep-fin-defl a*)·(*Rep-fin-defl b*) *oo udom-prj*))))

definition *cfun-defl* = *TypeRep-fun2 cfun-map*
definition *ssum-defl* = *TypeRep-fun2 ssum-map*
definition *sprod-defl* = *TypeRep-fun2 sprod-map*
definition *cprod-defl* = *TypeRep-fun2 cprod-map*
definition *u-defl* = *TypeRep-fun1 u-map*

lemma *Rep-fin-defl-mono*: *a* ⊑ *b* ⟹ *Rep-fin-defl a* ⊑ *Rep-fin-defl b*
unfolding *below-fin-defl-def* .

lemma *cast-TypeRep-fun1*:
assumes *f*: ⋀*a*. *finite-deflation a* ⟹ *finite-deflation* (*f*·*a*)
shows *cast*·(*TypeRep-fun1 f*·*A*) = *udom-emb oo f*·(*cast*·*A*) *oo udom-prj*
proof −
have *1*: ⋀*a*. *finite-deflation* (*udom-emb oo f*·(*Rep-fin-defl a*) *oo udom-prj*)
apply (*rule ep-pair.finite-deflation-e-d-p* [*OF ep-pair-udom*])
apply (*rule f*, *rule finite-deflation-Rep-fin-defl*)
done
show *?thesis*
by (*induct A rule*: *alg-defl.principal-induct*, *simp*)
(*simp only*: *TypeRep-fun1-def*
alg-defl.basis-fun-principal
alg-defl.basis-fun-mono
alg-defl.principal-mono
Abs-fin-defl-mono [*OF 1 1*]
monofun-cfun below-refl
Rep-fin-defl-mono

```
                cast-alg-defl-principal
                Abs-fin-defl-inverse [unfolded mem-Collect-eq, OF 1])
qed

lemma cast-TypeRep-fun2:
  assumes f: ⋀a b. finite-deflation a ⟹ finite-deflation b ⟹
              finite-deflation (f·a·b)
  shows cast·(TypeRep-fun2 f·A·B) =
    udom-emb oo f·(cast·A)·(cast·B) oo udom-prj
proof −
  have 1: ⋀a b. finite-deflation
          (udom-emb oo f·(Rep-fin-defl a)·(Rep-fin-defl b) oo udom-prj)
    apply (rule ep-pair.finite-deflation-e-d-p [OF ep-pair-udom])
    apply (rule f, (rule finite-deflation-Rep-fin-defl)+)
    done
  show ?thesis
    by (induct A B rule: alg-defl.principal-induct2, simp, simp)
      (simp only: TypeRep-fun2-def
                alg-defl.basis-fun-principal
                alg-defl.basis-fun-mono
                alg-defl.principal-mono
                Abs-fin-defl-mono [OF 1 1]
                monofun-cfun below-refl
                Rep-fin-defl-mono
                cast-alg-defl-principal
                Abs-fin-defl-inverse [unfolded mem-Collect-eq, OF 1])
qed

lemma cast-cfun-defl:
  cast·(cfun-defl·A·B) = udom-emb oo cfun-map·(cast·A)·(cast·B) oo udom-prj
unfolding cfun-defl-def
apply (rule cast-TypeRep-fun2)
apply (erule (1) finite-deflation-cfun-map)
done

lemma cast-ssum-defl:
  cast·(ssum-defl·A·B) = udom-emb oo ssum-map·(cast·A)·(cast·B) oo udom-prj
unfolding ssum-defl-def
apply (rule cast-TypeRep-fun2)
apply (erule (1) finite-deflation-ssum-map)
done

lemma cast-sprod-defl:
  cast·(sprod-defl·A·B) = udom-emb oo sprod-map·(cast·A)·(cast·B) oo udom-prj
unfolding sprod-defl-def
apply (rule cast-TypeRep-fun2)
apply (erule (1) finite-deflation-sprod-map)
done
```

lemma *cast-cprod-defl*:
cast·(cprod-defl·A·B) = udom-emb oo cprod-map·(cast·A)·(cast·B) oo udom-prj
unfolding *cprod-defl-def*
apply (*rule cast-TypeRep-fun2*)
apply (*erule (1) finite-deflation-cprod-map*)
done

lemma *cast-u-defl*:
cast·(u-defl·A) = udom-emb oo u-map·(cast·A) oo udom-prj
unfolding *u-defl-def*
apply (*rule cast-TypeRep-fun1*)
apply (*erule finite-deflation-u-map*)
done

REP of type constructor = type combinator

lemma *REP-cfun*: *REP('a → 'b) = cfun-defl·REP('a)·REP('b)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)
apply (*simp add*: *cast-REP cast-cfun-defl*)
apply (*simp add*: *cfun-map-def*)
apply (*simp only*: *prj-cfun-def emb-cfun-def*)
apply (*simp add*: *expand-cfun-eq ep-pair.e-eq-iff* [*OF ep-pair-udom*])
done

lemma *REP-ssum*: *REP('a ⊕ 'b) = ssum-defl·REP('a)·REP('b)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)
apply (*simp add*: *cast-REP cast-ssum-defl*)
apply (*simp add*: *prj-ssum-def*)
apply (*simp add*: *emb-ssum-def*)
apply (*simp add*: *ssum-map-map cfcomp1*)
done

lemma *REP-sprod*: *REP('a ⊗ 'b) = sprod-defl·REP('a)·REP('b)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)
apply (*simp add*: *cast-REP cast-sprod-defl*)
apply (*simp add*: *prj-sprod-def*)
apply (*simp add*: *emb-sprod-def*)
apply (*simp add*: *sprod-map-map cfcomp1*)
done

lemma *REP-cprod*: *REP('a × 'b) = cprod-defl·REP('a)·REP('b)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)
apply (*simp add*: *cast-REP cast-cprod-defl*)
apply (*simp add*: *prj-cprod-def*)
apply (*simp add*: *emb-cprod-def*)
apply (*simp add*: *cprod-map-map cfcomp1*)
done

lemma *REP-up*: *REP('a u) = u-defl·REP('a)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)

apply (*simp add*: *cast-REP cast-u-defl*)
apply (*simp add*: *prj-u-def*)
apply (*simp add*: *emb-u-def*)
apply (*simp add*: *u-map-map cfcomp1*)
done

lemmas *REP-simps* =
REP-cfun
REP-ssum
REP-sprod
REP-cprod
REP-up

27.8 Isomorphic deflations

definition
isodefl :: ('*a*::*rep* → '*a*) ⇒ *udom alg-defl* ⇒ *bool*
where
isodefl d t ⟷ *cast*·*t* = *emb oo d oo prj*

lemma *isodeflI*: (⋀*x*. *cast*·*t*·*x* = *emb*·(*d*·(*prj*·*x*))) ⟹ *isodefl d t*
unfolding *isodefl-def* **by** (*simp add*: *ext-cfun*)

lemma *cast-isodefl*: *isodefl d t* ⟹ *cast*·*t* = (Λ *x*. *emb*·(*d*·(*prj*·*x*)))
unfolding *isodefl-def* **by** (*simp add*: *ext-cfun*)

lemma *isodefl-strict*: *isodefl d t* ⟹ *d*·⊥ = ⊥
unfolding *isodefl-def*
by (*drule cfun-fun-cong* [**where** *x*=⊥], *simp*)

lemma *isodefl-imp-deflation*:
fixes *d* :: '*a*::*rep* → '*a*
assumes *isodefl d t* **shows** *deflation d*
proof
note *prems* [*unfolded isodefl-def*, *simp*]
fix *x* :: '*a*
show *d*·(*d*·*x*) = *d*·*x*
using *cast.idem* [*of t emb*·*x*] **by** *simp*
show *d*·*x* ⊑ *x*
using *cast.below* [*of t emb*·*x*] **by** *simp*
qed

lemma *isodefl-ID-REP*: *isodefl* (*ID* :: '*a* → '*a*) *REP*('*a*)
unfolding *isodefl-def* **by** (*simp add*: *cast-REP*)

lemma *isodefl-REP-imp-ID*: *isodefl* (*d* :: '*a* → '*a*) *REP*('*a*) ⟹ *d* = *ID*
unfolding *isodefl-def*
apply (*simp add*: *cast-REP*)
apply (*simp add*: *expand-cfun-eq*)

apply (*rule allI*)
apply (*drule-tac x=emb·x* **in** *spec*)
apply *simp*
done

lemma *isodefl-bottom*: *isodefl* ⊥ ⊥
unfolding *isodefl-def* **by** (*simp add*: *expand-cfun-eq*)

lemma *adm-isodefl*:
  cont f ⟹ *cont g* ⟹ *adm* (λ*x*. *isodefl* (*f x*) (*g x*))
unfolding *isodefl-def* **by** *simp*

lemma *isodefl-lub*:
  assumes *chain d* **and** *chain t*
  assumes ⋀*i*. *isodefl* (*d i*) (*t i*)
  shows *isodefl* (⨆*i*. *d i*) (⨆*i*. *t i*)
using *prems* **unfolding** *isodefl-def*
by (*simp add*: *contlub-cfun-arg contlub-cfun-fun*)

lemma *isodefl-fix*:
  assumes ⋀*d t*. *isodefl d t* ⟹ *isodefl* (*f*·*d*) (*g*·*t*)
  shows *isodefl* (*fix*·*f*) (*fix*·*g*)
unfolding *fix-def2*
apply (*rule isodefl-lub*, *simp*, *simp*)
apply (*induct-tac i*)
apply (*simp add*: *isodefl-bottom*)
apply (*simp add*: *prems*)
done

lemma *isodefl-coerce*:
  fixes *d* :: *'a* → *'a*
  assumes *REP*: *REP*(*'b*) = *REP*(*'a*)
  shows *isodefl d t* ⟹ *isodefl* (*coerce oo d oo coerce* :: *'b* → *'b*) *t*
unfolding *isodefl-def*
apply (*simp add*: *expand-cfun-eq*)
apply (*simp add*: *emb-coerce coerce-prj REP*)
done

lemma *isodefl-abs-rep*:
  fixes *abs* **and** *rep* **and** *d*
  assumes *REP*: *REP*(*'b*) = *REP*(*'a*)
  assumes *abs-def*: *abs* ≡ (*coerce* :: *'a* → *'b*)
  assumes *rep-def*: *rep* ≡ (*coerce* :: *'b* → *'a*)
  shows *isodefl d t* ⟹ *isodefl* (*abs oo d oo rep*) *t*
unfolding *abs-def rep-def* **using** *REP* **by** (*rule isodefl-coerce*)

lemma *isodefl-cfun*:
  isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
    isodefl (*cfun-map*·*d1*·*d2*) (*cfun-defl*·*t1*·*t2*)

apply (*rule isodeflI*)
apply (*simp add*: *cast-cfun-defl cast-isodefl*)
apply (*simp add*: *emb-cfun-def prj-cfun-def*)
apply (*simp add*: *cfun-map-map cfcomp1*)
done

lemma *isodefl-ssum*:
isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
isodefl (*ssum-map*·*d1*·*d2*) (*ssum-defl*·*t1*·*t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-ssum-defl cast-isodefl*)
apply (*simp add*: *emb-ssum-def prj-ssum-def*)
apply (*simp add*: *ssum-map-map isodefl-strict*)
done

lemma *isodefl-sprod*:
isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
isodefl (*sprod-map*·*d1*·*d2*) (*sprod-defl*·*t1*·*t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-sprod-defl cast-isodefl*)
apply (*simp add*: *emb-sprod-def prj-sprod-def*)
apply (*simp add*: *sprod-map-map isodefl-strict*)
done

lemma *isodefl-cprod*:
isodefl d1 t1 $\Longrightarrow$ *isodefl d2 t2* $\Longrightarrow$
isodefl (*cprod-map*·*d1*·*d2*) (*cprod-defl*·*t1*·*t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-cprod-defl cast-isodefl*)
apply (*simp add*: *emb-cprod-def prj-cprod-def*)
apply (*simp add*: *cprod-map-map cfcomp1*)
done

lemma *isodefl-u*:
isodefl d t $\Longrightarrow$ *isodefl* (*u-map*·*d*) (*u-defl*·*t*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-u-defl cast-isodefl*)
apply (*simp add*: *emb-u-def prj-u-def*)
apply (*simp add*: *u-map-map*)
done

27.9 Constructing Domain Isomorphisms

use *Tools/Domain/domain-isomorphism.ML*

setup ⟪
fold Domain-Isomorphism.add-type-constructor
[(@{*type-name cfun*}, @{*term cfun-defl*}, @{*const-name cfun-map*}, @{*thm REP-cfun*},

@{*thm isodefl-cfun*}, @{*thm cfun-map-ID*}, @{*thm deflation-cfun-map*}),

(@{*type-name ssum*}, @{*term ssum-defl*}, @{*const-name ssum-map*}, @{*thm REP-ssum*},
@{*thm isodefl-ssum*}, @{*thm ssum-map-ID*}, @{*thm deflation-ssum-map*}),

(@{*type-name sprod*}, @{*term sprod-defl*}, @{*const-name sprod-map*}, @{*thm REP-sprod*},
@{*thm isodefl-sprod*}, @{*thm sprod-map-ID*}, @{*thm deflation-sprod-map*}),

(@{*type-name* ∗}, @{*term cprod-defl*}, @{*const-name cprod-map*}, @{*thm REP-cprod*},
@{*thm isodefl-cprod*}, @{*thm cprod-map-ID*}, @{*thm deflation-cprod-map*}),

(@{*type-name u*}, @{*term u-defl*}, @{*const-name u-map*}, @{*thm REP-up*},
@{*thm isodefl-u*}, @{*thm u-map-ID*}, @{*thm deflation-u-map*})]
⟫

end

28 Domain: Domain package

theory *Domain*
imports *Ssum Sprod Up One Tr Fixrec Representable*
uses
(*Tools/cont-consts.ML*)
(*Tools/cont-proc.ML*)
(*Tools/Domain/domain-constructors.ML*)
(*Tools/Domain/domain-library.ML*)
(*Tools/Domain/domain-axioms.ML*)
(*Tools/Domain/domain-theorems.ML*)
(*Tools/Domain/domain-extender.ML*)
begin

default-sort *pcpo*

28.1 Casedist

lemma *ex-one-defined-iff*:
(∃ *x*. *P x* ∧ *x* ≠ ⊥) = *P ONE*
apply *safe*
apply (*rule-tac p*=*x* **in** *oneE*)
apply *simp*
apply *simp*
apply *force*
done

lemma *ex-up-defined-iff*:

```
 (∃ x. P x ∧ x ≠ ⊥) = (∃ x. P (up·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:
(∃ y. P y ∧ y ≠ ⊥) =
 (∃ x y. (P (:x, y:) ∧ x ≠ ⊥) ∧ y ≠ ⊥)
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:
(∃ y. P y ∧ y ≠ ⊥) =
 (∃ x y. P (:up·x, y:) ∧ y ≠ ⊥)
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:
(∃ x. P x ∧ x ≠ ⊥) =
((∃ x. P (sinl·x) ∧ x ≠ ⊥) ∨
 (∃ x. P (sinr·x) ∧ x ≠ ⊥))
apply (rule iffI)
 apply (erule exE)
 apply (erule conjE)
 apply (rule-tac p=x in ssumE)
   apply simp
  apply (rule disjI1, fast)
 apply (rule disjI2, fast)
apply (erule disjE)
 apply force
apply force
done

lemma exh-start: p = ⊥ ∨ (∃ x. p = x ∧ x ≠ ⊥)
 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*

28.2 Combinators for building copy functions

lemmas *domain-map-stricts* =
ssum-map-strict sprod-map-strict u-map-strict

lemmas *domain-map-simps* =
ssum-map-sinl ssum-map-sinr sprod-map-spair u-map-up

28.3 Installing the domain package

lemmas *con-strict-rules* =
sinl-strict sinr-strict spair-strict1 spair-strict2

lemmas *con-defin-rules* =
sinl-defined sinr-defined spair-defined up-defined ONE-defined

lemmas *con-defined-iff-rules* =
sinl-defined-iff sinr-defined-iff spair-strict-iff up-defined ONE-defined

lemmas *con-below-iff-rules* =
sinl-below sinr-below sinl-below-sinr sinr-below-sinl con-defined-iff-rules

lemmas *con-eq-iff-rules* =
sinl-eq sinr-eq sinl-eq-sinr sinr-eq-sinl con-defined-iff-rules

lemmas *sel-strict-rules* =
cfcomp2 sscase1 sfst-strict ssnd-strict fup1

lemma *sel-app-extra-rules*:
sscase·ID·⊥·(sinr·x) = ⊥
sscase·ID·⊥·(sinl·x) = x
sscase·⊥·ID·(sinl·x) = ⊥
sscase·⊥·ID·(sinr·x) = x
fup·ID·(up·x) = x
by (*cases x = ⊥*, *simp*, *simp*)+

lemmas *sel-app-rules* =
sel-strict-rules sel-app-extra-rules
ssnd-spair sfst-spair up-defined spair-defined

lemmas *sel-defined-iff-rules* =
cfcomp2 sfst-defined-iff ssnd-defined-iff

lemmas *take-con-rules* =
ssum-map-sinl' ssum-map-sinr' sprod-map-spair' u-map-up
deflation-strict deflation-ID ID1 cfcomp2

use *Tools/cont-consts.ML*
use *Tools/cont-proc.ML*
use *Tools/Domain/domain-library.ML*
use *Tools/Domain/domain-axioms.ML*
use *Tools/Domain/domain-constructors.ML*
use *Tools/Domain/domain-theorems.ML*
use *Tools/Domain/domain-extender.ML*

end

29 UpperPD: Upper powerdomain

theory *UpperPD*
imports *CompactBasis*
begin

29.1 Basis preorder

definition
upper-le :: *'a pd-basis ⇒ 'a pd-basis ⇒ bool* (**infix** $\leq\sharp$ *50*) **where**
upper-le = (*λu v. ∀ y∈Rep-pd-basis v. ∃ x∈Rep-pd-basis u. x ⊑ y*)

lemma *upper-le-refl* [*simp*]: $t \leq\sharp t$
unfolding *upper-le-def* **by** *fast*

lemma *upper-le-trans*: $[\![t \leq\sharp u; u \leq\sharp v]\!] \Longrightarrow t \leq\sharp v$
unfolding *upper-le-def*
apply (*rule ballI*)

apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*erule rev-bexI*)
apply (*erule* (*1*) *below-trans*)
done

interpretation *upper-le*: *preorder upper-le*
by (*rule preorder.intro*, *rule upper-le-refl*, *rule upper-le-trans*)

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\sharp$ *t*
unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-upper-mono*: $x \sqsubseteq y \Longrightarrow$ *PDUnit x* $\leq\sharp$ *PDUnit y*
unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDPlus-upper-mono*: $[\![s \leq\sharp t;\ u \leq\sharp v]\!] \Longrightarrow$ *PDPlus s u* $\leq\sharp$ *PDPlus t v*
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-upper-le*: *PDPlus t u* $\leq\sharp$ *t*
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* $\leq\sharp$ *PDUnit b*) = $a \sqsubseteq b$
unfolding *upper-le-def Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-PDUnit-iff*:
(*PDPlus t u* $\leq\sharp$ *PDUnit a*) = (*t* $\leq\sharp$ *PDUnit a* $\vee$ *u* $\leq\sharp$ *PDUnit a*)
unfolding *upper-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-iff*: (*t* $\leq\sharp$ *PDPlus u v*) = (*t* $\leq\sharp$ *u* $\wedge$ *t* $\leq\sharp$ *v*)
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-induct* [*induct set*: *upper-le*]:
assumes *le*: *t* $\leq\sharp$ *u*
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P$ (*PDUnit a*) (*PDUnit b*)
assumes *2*: $\bigwedge t\ u\ a.\ P\ t$ (*PDUnit a*) $\Longrightarrow P$ (*PDPlus t u*) (*PDUnit a*)
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ t\ v]\!] \Longrightarrow P\ t$ (*PDPlus u v*)
shows *P t u*
using *le* **apply** (*induct u arbitrary*: *t rule*: *pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp add*: *1*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply (*simp add*: *2*)
apply (*subst PDPlus-commute*)
apply (*simp add*: *2*)
apply (*simp add*: *upper-le-PDPlus-iff 3*)
done

lemma *pd-take-upper-chain*:
  pd-take n t $\leq\sharp$ *pd-take* (*Suc n*) *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-chain*)
apply (*simp add*: *PDPlus-upper-mono*)
done

lemma *pd-take-upper-le*: *pd-take i t* $\leq\sharp$ *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-less*)
apply (*simp add*: *PDPlus-upper-mono*)
done

lemma *pd-take-upper-mono*:
  t $\leq\sharp$ *u* $\Longrightarrow$ *pd-take n t* $\leq\sharp$ *pd-take n u*
apply (*erule upper-le-induct*)
apply (*simp add*: *compact-basis.take-mono*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply (*simp add*: *upper-le-PDPlus-iff*)
done

29.2 Type definition

typedef (**open**) *'a upper-pd* =
  {*S*::*'a pd-basis set*. *upper-le.ideal S*}
by (*fast intro*: *upper-le.ideal-principal*)

instantiation *upper-pd* :: (*profinite*) *below*
begin

definition
  x $\sqsubseteq$ *y* $\longleftrightarrow$ *Rep-upper-pd x* $\subseteq$ *Rep-upper-pd y*

instance ..
end

instance *upper-pd* :: (*profinite*) *po*
by (*rule upper-le.typedef-ideal-po*
    [*OF type-definition-upper-pd below-upper-pd-def*])

instance *upper-pd* :: (*profinite*) *cpo*
by (*rule upper-le.typedef-ideal-cpo*
    [*OF type-definition-upper-pd below-upper-pd-def*])

lemma *Rep-upper-pd-lub*:
  chain Y $\Longrightarrow$ *Rep-upper-pd* ($\bigsqcup$ *i*. *Y i*) = ($\bigcup$ *i*. *Rep-upper-pd* (*Y i*))
by (*rule upper-le.typedef-ideal-rep-contlub*
    [*OF type-definition-upper-pd below-upper-pd-def*])

lemma *ideal-Rep-upper-pd*: *upper-le.ideal* (*Rep-upper-pd xs*)
by (*rule Rep-upper-pd* [*unfolded mem-Collect-eq*])

definition
upper-principal :: *'a pd-basis* ⇒ *'a upper-pd* **where**
upper-principal t = *Abs-upper-pd* {*u*. *u* ≤♯ *t*}

lemma *Rep-upper-principal*:
Rep-upper-pd (*upper-principal t*) = {*u*. *u* ≤♯ *t*}
unfolding *upper-principal-def*
by (*simp add*: *Abs-upper-pd-inverse upper-le.ideal-principal*)

interpretation *upper-pd*:
ideal-completion upper-le pd-take upper-principal Rep-upper-pd
apply *unfold-locales*
apply (*rule pd-take-upper-le*)
apply (*rule pd-take-idem*)
apply (*erule pd-take-upper-mono*)
apply (*rule pd-take-upper-chain*)
apply (*rule finite-range-pd-take*)
apply (*rule pd-take-covers*)
apply (*rule ideal-Rep-upper-pd*)
apply (*erule Rep-upper-pd-lub*)
apply (*rule Rep-upper-principal*)
apply (*simp only*: *below-upper-pd-def*)
done

Upper powerdomain is pointed

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) ⊑ *ys*
by (*induct ys rule*: *upper-pd.principal-induct*, *simp*, *simp*)

instance *upper-pd* :: (*bifinite*) *pcpo*
by *intro-classes* (*fast intro*: *upper-pd-minimal*)

lemma *inst-upper-pd-pcpo*: ⊥ = *upper-principal* (*PDUnit compact-bot*)
by (*rule upper-pd-minimal* [*THEN UU-I*, *symmetric*])

Upper powerdomain is profinite

instantiation *upper-pd* :: (*profinite*) *profinite*
begin

definition
approx-upper-pd-def: *approx* = *upper-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-upper-pd-def*)
apply (*rule upper-pd.chain-completion-approx*)
apply (*rule upper-pd.lub-completion-approx*)
apply (*rule upper-pd.completion-approx-idem*)

apply (*rule upper-pd.finite-fixes-completion-approx*)
done

end

instance *upper-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-upper-principal* [*simp*]:
approx n·(*upper-principal t*) = *upper-principal* (*pd-take n t*)
unfolding *approx-upper-pd-def*
by (*rule upper-pd.completion-approx-principal*)

lemma *approx-eq-upper-principal*:
∃ *t*∈*Rep-upper-pd xs*. *approx n·xs* = *upper-principal* (*pd-take n t*)
unfolding *approx-upper-pd-def*
by (*rule upper-pd.completion-approx-eq-principal*)

29.3 Monadic unit and plus

definition
upper-unit :: *'a* → *'a upper-pd* **where**
upper-unit = *compact-basis.basis-fun* (λ*a*. *upper-principal* (*PDUnit a*))

definition
upper-plus :: *'a upper-pd* → *'a upper-pd* → *'a upper-pd* **where**
upper-plus = *upper-pd.basis-fun* (λ*t*. *upper-pd.basis-fun* (λ*u*.
upper-principal (*PDPlus t u*)))

abbreviation
upper-add :: *'a upper-pd* ⇒ *'a upper-pd* ⇒ *'a upper-pd*
(**infixl** +♯ *65*) **where**
xs +♯ *ys* == *upper-plus·xs·ys*

syntax
-upper-pd :: *args* ⇒ *'a upper-pd* ({-}♯)

translations
{*x*,*xs*}♯ == {*x*}♯ +♯ {*xs*}♯
{*x*}♯ == *CONST upper-unit·x*

lemma *upper-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♯ = *upper-principal* (*PDUnit a*)
unfolding *upper-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal PDUnit-upper-mono*)

lemma *upper-plus-principal* [*simp*]:
upper-principal t +♯ *upper-principal u* = *upper-principal* (*PDPlus t u*)
unfolding *upper-plus-def*
by (*simp add*: *upper-pd.basis-fun-principal*

upper-pd.basis-fun-mono PDPlus-upper-mono)

lemma *approx-upper-unit* [*simp*]:
approx n·{x}♯ = {approx n·x}♯
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-upper-plus* [*simp*]:
approx n·(xs +♯ ys) = (approx n·xs) +♯ (approx n·ys)
by (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*, *simp*)

interpretation *upper-add*!: *semilattice upper-add* **proof**
fix *xs ys zs* :: *'a upper-pd*
show (*xs +♯ ys*) *+♯ zs = xs +♯* (*ys +♯ zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *upper-pd.principal-induct2*, *simp*, *simp*)
apply (*rule-tac x=zs* **in** *upper-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done
show *xs +♯ ys = ys +♯ xs*
apply (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done
show *xs +♯ xs = xs*
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done
qed

lemmas *upper-plus-assoc = upper-add.assoc*
lemmas *upper-plus-commute = upper-add.commute*
lemmas *upper-plus-absorb = upper-add.idem*
lemmas *upper-plus-left-commute = upper-add.left-commute*
lemmas *upper-plus-left-absorb = upper-add.left-idem*

Useful for *simp add*: *upper-plus-ac*

lemmas *upper-plus-ac* =
upper-plus-assoc upper-plus-commute upper-plus-left-commute

Useful for *simp only*: *upper-plus-aci*

lemmas *upper-plus-aci* =
upper-plus-ac upper-plus-absorb upper-plus-left-absorb

lemma *upper-plus-below1*: *xs +♯ ys ⊑ xs*
apply (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-upper-le*)
done

lemma *upper-plus-below2*: *xs +♯ ys ⊑ ys*

by (*subst upper-plus-commute*, *rule upper-plus-below1*)

lemma *upper-plus-greatest*: $[\![xs \sqsubseteq ys;\ xs \sqsubseteq zs]\!] \Longrightarrow xs \sqsubseteq ys +\sharp zs$
apply (*subst upper-plus-absorb* [*of xs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *upper-below-plus-iff*:
$xs \sqsubseteq ys +\sharp zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
apply *safe*
apply (*erule below-trans* [*OF - upper-plus-below1*])
apply (*erule below-trans* [*OF - upper-plus-below2*])
apply (*erule* (*1*) *upper-plus-greatest*)
done

lemma *upper-plus-below-unit-iff*:
$xs +\sharp ys \sqsubseteq \{z\}\sharp \longleftrightarrow xs \sqsubseteq \{z\}\sharp \vee ys \sqsubseteq \{z\}\sharp$
apply (*rule iffI*)
apply (*subgoal-tac*
adm ($\lambda f.\ f{\cdot}xs \sqsubseteq f{\cdot}\{z\}\sharp \vee f{\cdot}ys \sqsubseteq f{\cdot}\{z\}\sharp$))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x=approx i·xs* **in** *upper-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·ys* **in** *upper-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·z* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*clarify*, *simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply *simp*
apply *simp*
apply (*erule disjE*)
apply (*erule below-trans* [*OF upper-plus-below1*])
apply (*erule below-trans* [*OF upper-plus-below2*])
done

lemma *upper-unit-below-iff* [*simp*]: $\{x\}\sharp \sqsubseteq \{y\}\sharp \longleftrightarrow x \sqsubseteq y$
apply (*rule iffI*)
apply (*rule profinite-below-ext*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
apply (*cut-tac x=approx i·x* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·y* **in** *compact-basis.compact-imp-principal*, *simp*)
apply *clarsimp*
apply (*erule monofun-cfun-arg*)
done

lemmas *upper-pd-below-simps* =
upper-unit-below-iff
upper-below-plus-iff
upper-plus-below-unit-iff

lemma *upper-unit-eq-iff* [*simp*]: $\{x\}\sharp = \{y\}\sharp \longleftrightarrow x = y$

unfolding *po-eq-conv* **by** *simp*

lemma *upper-unit-strict* [*simp*]: $\{\bot\}\sharp = \bot$
unfolding *inst-upper-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *upper-plus-strict1* [*simp*]: $\bot +\sharp ys = \bot$
by (*rule UU-I*, *rule upper-plus-below1*)

lemma *upper-plus-strict2* [*simp*]: $xs +\sharp \bot = \bot$
by (*rule UU-I*, *rule upper-plus-below2*)

lemma *upper-unit-strict-iff* [*simp*]: $\{x\}\sharp = \bot \longleftrightarrow x = \bot$
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-unit-eq-iff*)

lemma *upper-plus-strict-iff* [*simp*]:
$xs +\sharp ys = \bot \longleftrightarrow xs = \bot \vee ys = \bot$
apply (*rule iffI*)
apply (*erule rev-mp*)
apply (*rule upper-pd.principal-induct2* [**where** *x*=*xs* **and** *y*=*ys*], *simp*, *simp*)
apply (*simp add*: *inst-upper-pd-pcpo upper-pd.principal-eq-iff*
upper-le-PDPlus-PDUnit-iff)
apply *auto*
done

lemma *compact-upper-unit-iff* [*simp*]: *compact* $\{x\}\sharp \longleftrightarrow$ *compact x*
unfolding *profinite-compact-iff* **by** *simp*

lemma *compact-upper-plus* [*simp*]:
$[\![compact\ xs;\ compact\ ys]\!] \Longrightarrow compact\ (xs +\sharp ys)$
by (*auto dest*!: *upper-pd.compact-imp-principal*)

29.4 Induction rules

lemma *upper-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\sharp;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\sharp +\sharp ys)$
shows *P* (*xs*::'*a upper-pd*)
apply (*induct xs rule*: *upper-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct1*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*]
upper-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *upper-pd-induct*:
assumes *P*: *adm P*

assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\sharp ys)$
shows $P\ (xs{::}'a\ upper\text{-}pd)$
apply (*induct xs rule*: *upper-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only*: *upper-plus-principal* [*symmetric*] *plus*)
done

29.5 Monadic bind

definition
upper-bind-basis ::
$'a\ pd\text{-}basis \Rightarrow ('a \to 'b\ upper\text{-}pd) \to 'b\ upper\text{-}pd$ **where**
upper-bind-basis = *fold-pd*
$(\lambda a.\ \Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\sharp y{\cdot}f)$

lemma *ACI-upper-bind*:
class.ab-semigroup-idem-mult $(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\sharp y{\cdot}f)$
apply *unfold-locales*
apply (*simp add*: *upper-plus-assoc*)
apply (*simp add*: *upper-plus-commute*)
apply (*simp add*: *eta-cfun*)
done

lemma *upper-bind-basis-simps* [*simp*]:
upper-bind-basis $(PDUnit\ a) =$
$(\Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
upper-bind-basis $(PDPlus\ t\ u) =$
$(\Lambda\ f.\ upper\text{-}bind\text{-}basis\ t{\cdot}f +\sharp upper\text{-}bind\text{-}basis\ u{\cdot}f)$
unfolding *upper-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-upper-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-upper-bind*])
done

lemma *upper-bind-basis-mono*:
$t \leq\sharp u \Longrightarrow upper\text{-}bind\text{-}basis\ t \sqsubseteq upper\text{-}bind\text{-}basis\ u$
unfolding *expand-cfun-below*
apply (*erule upper-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *below-trans* [*OF upper-plus-below1*])
apply (*simp add*: *upper-below-plus-iff*)
done

definition
upper-bind :: $'a\ upper\text{-}pd \to ('a \to 'b\ upper\text{-}pd) \to 'b\ upper\text{-}pd$ **where**
upper-bind = *upper-pd.basis-fun upper-bind-basis*

lemma *upper-bind-principal* [*simp*]:
upper-bind·(*upper-principal t*) = *upper-bind-basis t*
unfolding *upper-bind-def*
apply (*rule upper-pd.basis-fun-principal*)
apply (*erule upper-bind-basis-mono*)
done

lemma *upper-bind-unit* [*simp*]:
upper-bind·{*x*}♯·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *upper-bind-plus* [*simp*]:
upper-bind·(*xs* +♯ *ys*)·*f* = *upper-bind*·*xs*·*f* +♯ *upper-bind*·*ys*·*f*
by (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *upper-bind-strict* [*simp*]: *upper-bind*·⊥·*f* = *f*·⊥
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-bind-unit*)

29.6 Map and join

definition
upper-map :: ('*a* → '*b*) → '*a upper-pd* → '*b upper-pd* **where**
upper-map = (Λ *f xs*. *upper-bind*·*xs*·(Λ *x*. {*f*·*x*}♯))

definition
upper-join :: '*a upper-pd upper-pd* → '*a upper-pd* **where**
upper-join = (Λ *xss*. *upper-bind*·*xss*·(Λ *xs*. *xs*))

lemma *upper-map-unit* [*simp*]:
upper-map·*f*·{*x*}♯ = {*f*·*x*}♯
unfolding *upper-map-def* **by** *simp*

lemma *upper-map-plus* [*simp*]:
upper-map·*f*·(*xs* +♯ *ys*) = *upper-map*·*f*·*xs* +♯ *upper-map*·*f*·*ys*
unfolding *upper-map-def* **by** *simp*

lemma *upper-join-unit* [*simp*]:
upper-join·{*xs*}♯ = *xs*
unfolding *upper-join-def* **by** *simp*

lemma *upper-join-plus* [*simp*]:
upper-join·(*xss* +♯ *yss*) = *upper-join*·*xss* +♯ *upper-join*·*yss*
unfolding *upper-join-def* **by** *simp*

lemma *upper-map-ident*: *upper-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-map-ID*: *upper-map*·*ID* = *ID*

by (*simp add*: *expand-cfun-eq ID-def upper-map-ident*)

lemma *upper-map-map*:
upper-map·*f*·(*upper-map*·*g*·*xs*) = *upper-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-unit*:
upper-join·(*upper-map*·*upper-unit*·*xs*) = *xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-join*:
upper-join·(*upper-map*·*upper-join*·*xsss*) = *upper-join*·(*upper-join*·*xsss*)
by (*induct xsss rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-map*:
upper-join·(*upper-map*·(*upper-map*·*f*)·*xss*) =
upper-map·*f*·(*upper-join*·*xss*)
by (*induct xss rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-map-approx*: *upper-map*·(*approx n*)·*xs* = *approx n*·*xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *ep-pair-upper-map*: *ep-pair e p* ⟹ *ep-pair* (*upper-map*·*e*) (*upper-map*·*p*)
apply *default*
apply (*induct-tac x rule*: *upper-pd-induct*, *simp-all add*: *ep-pair.e-inverse*)
apply (*induct-tac y rule*: *upper-pd-induct*)
apply (*simp-all add*: *ep-pair.e-p-below monofun-cfun*)
done

lemma *deflation-upper-map*: *deflation d* ⟹ *deflation* (*upper-map*·*d*)
apply *default*
apply (*induct-tac x rule*: *upper-pd-induct*, *simp-all add*: *deflation.idem*)
apply (*induct-tac x rule*: *upper-pd-induct*)
apply (*simp-all add*: *deflation.below monofun-cfun*)
done

end

30 LowerPD: Lower powerdomain

theory *LowerPD*
imports *CompactBasis*
begin

30.1 Basis preorder

definition
lower-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♭ *50*) **where**

lower-le = $(\lambda u\ v.\ \forall x{\in}$*Rep-pd-basis* $u.\ \exists y{\in}$*Rep-pd-basis* $v.\ x \sqsubseteq y)$

lemma *lower-le-refl* [*simp*]: $t \leq\flat\ t$
unfolding *lower-le-def* **by** *fast*

lemma *lower-le-trans*: $[\![t \leq\flat\ u;\ u \leq\flat\ v]\!] \Longrightarrow t \leq\flat\ v$
unfolding *lower-le-def*
apply (*rule ballI*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*erule rev-bexI*)
apply (*erule* (*1*) *below-trans*)
done

interpretation *lower-le*: *preorder lower-le*
by (*rule preorder.intro*, *rule lower-le-refl*, *rule lower-le-trans*)

lemma *lower-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\flat\ t$
unfolding *lower-le-def Rep-PDUnit*
by (*simp*, *rule Rep-pd-basis-nonempty* [*folded ex-in-conv*])

lemma *PDUnit-lower-mono*: $x \sqsubseteq y \Longrightarrow$ *PDUnit* $x \leq\flat$ *PDUnit* y
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *PDPlus-lower-mono*: $[\![s \leq\flat\ t;\ u \leq\flat\ v]\!] \Longrightarrow$ *PDPlus* $s\ u \leq\flat$ *PDPlus* $t\ v$
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-lower-le*: $t \leq\flat$ *PDPlus* $t\ u$
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit* $a \leq\flat$ *PDUnit* $b) = a \sqsubseteq b$
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *lower-le-PDUnit-PDPlus-iff*:
(*PDUnit* $a \leq\flat$ *PDPlus* $t\ u) =$ (*PDUnit* $a \leq\flat\ t \vee$ *PDUnit* $a \leq\flat\ u$)
unfolding *lower-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *lower-le-PDPlus-iff*: (*PDPlus* $t\ u \leq\flat\ v) = (t \leq\flat\ v \wedge u \leq\flat\ v)$
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-induct* [*induct set*: *lower-le*]:
assumes *le*: $t \leq\flat\ u$
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P$ (*PDUnit* a) (*PDUnit* b)
assumes *2*: $\bigwedge t\ u\ a.\ P$ (*PDUnit* a) $t \Longrightarrow P$ (*PDUnit* a) (*PDPlus* $t\ u$)
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ v;\ P\ u\ v]\!] \Longrightarrow P$ (*PDPlus* $t\ u$) v
shows $P\ t\ u$
using *le*
apply (*induct t arbitrary*: *u rule*: *pd-basis-induct*)

apply (*erule rev-mp*)
apply (*induct-tac u rule*: *pd-basis-induct*)
apply (*simp add*: *1*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
apply (*simp add*: *2*)
apply (*subst PDPlus-commute*)
apply (*simp add*: *2*)
apply (*simp add*: *lower-le-PDPlus-iff 3*)
done

lemma *pd-take-lower-chain*:
pd-take n t $\leq\flat$ *pd-take* (*Suc n*) *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-chain*)
apply (*simp add*: *PDPlus-lower-mono*)
done

lemma *pd-take-lower-le*: *pd-take i t* $\leq\flat$ *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-less*)
apply (*simp add*: *PDPlus-lower-mono*)
done

lemma *pd-take-lower-mono*:
t $\leq\flat$ *u* $\Longrightarrow$ *pd-take n t* $\leq\flat$ *pd-take n u*
apply (*erule lower-le-induct*)
apply (*simp add*: *compact-basis.take-mono*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
apply (*simp add*: *lower-le-PDPlus-iff*)
done

30.2 Type definition

typedef (**open**) *'a lower-pd* =
{*S*::*'a pd-basis set*. *lower-le.ideal S*}
by (*fast intro*: *lower-le.ideal-principal*)

instantiation *lower-pd* :: (*profinite*) *below*
begin

definition
x $\sqsubseteq$ *y* $\longleftrightarrow$ *Rep-lower-pd x* $\subseteq$ *Rep-lower-pd y*

instance ..
end

instance *lower-pd* :: (*profinite*) *po*
by (*rule lower-le.typedef-ideal-po*
[*OF type-definition-lower-pd below-lower-pd-def*])

instance *lower-pd* :: (*profinite*) *cpo*
by (*rule lower-le.typedef-ideal-cpo*
[*OF type-definition-lower-pd below-lower-pd-def*])

lemma *Rep-lower-pd-lub*:
chain Y $\Longrightarrow$ *Rep-lower-pd* $(\bigsqcup i.\ Y\ i) = (\bigcup i.\ \textit{Rep-lower-pd}\ (Y\ i))$
by (*rule lower-le.typedef-ideal-rep-contlub*
[*OF type-definition-lower-pd below-lower-pd-def*])

lemma *ideal-Rep-lower-pd*: *lower-le.ideal* (*Rep-lower-pd xs*)
by (*rule Rep-lower-pd* [*unfolded mem-Collect-eq*])

definition
lower-principal :: *'a pd-basis* $\Rightarrow$ *'a lower-pd* **where**
lower-principal t = *Abs-lower-pd* $\{u.\ u \leq\flat\ t\}$

lemma *Rep-lower-principal*:
Rep-lower-pd (*lower-principal t*) = $\{u.\ u \leq\flat\ t\}$
unfolding *lower-principal-def*
by (*simp add*: *Abs-lower-pd-inverse lower-le.ideal-principal*)

interpretation *lower-pd*:
ideal-completion lower-le pd-take lower-principal Rep-lower-pd
apply *unfold-locales*
apply (*rule pd-take-lower-le*)
apply (*rule pd-take-idem*)
apply (*erule pd-take-lower-mono*)
apply (*rule pd-take-lower-chain*)
apply (*rule finite-range-pd-take*)
apply (*rule pd-take-covers*)
apply (*rule ideal-Rep-lower-pd*)
apply (*erule Rep-lower-pd-lub*)
apply (*rule Rep-lower-principal*)
apply (*simp only*: *below-lower-pd-def*)
done

Lower powerdomain is pointed

lemma *lower-pd-minimal*: *lower-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*
by (*induct ys rule*: *lower-pd.principal-induct*, *simp*, *simp*)

instance *lower-pd* :: (*bifinite*) *pcpo*
by *intro-classes* (*fast intro*: *lower-pd-minimal*)

lemma *inst-lower-pd-pcpo*: $\bot$ = *lower-principal* (*PDUnit compact-bot*)
by (*rule lower-pd-minimal* [*THEN UU-I*, *symmetric*])

Lower powerdomain is profinite

instantiation *lower-pd* :: (*profinite*) *profinite*

begin

definition
approx-lower-pd-def: *approx* = *lower-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-lower-pd-def*)
apply (*rule lower-pd.chain-completion-approx*)
apply (*rule lower-pd.lub-completion-approx*)
apply (*rule lower-pd.completion-approx-idem*)
apply (*rule lower-pd.finite-fixes-completion-approx*)
done

end

instance *lower-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-lower-principal* [*simp*]:
approx n·(*lower-principal t*) = *lower-principal* (*pd-take n t*)
unfolding *approx-lower-pd-def*
by (*rule lower-pd.completion-approx-principal*)

lemma *approx-eq-lower-principal*:
∃ *t*∈*Rep-lower-pd xs*. *approx n*·*xs* = *lower-principal* (*pd-take n t*)
unfolding *approx-lower-pd-def*
by (*rule lower-pd.completion-approx-eq-principal*)

30.3 Monadic unit and plus

definition
lower-unit :: *'a* → *'a lower-pd* **where**
lower-unit = *compact-basis.basis-fun* (λ*a*. *lower-principal* (*PDUnit a*))

definition
lower-plus :: *'a lower-pd* → *'a lower-pd* → *'a lower-pd* **where**
lower-plus = *lower-pd.basis-fun* (λ*t*. *lower-pd.basis-fun* (λ*u*.
lower-principal (*PDPlus t u*)))

abbreviation
lower-add :: *'a lower-pd* ⇒ *'a lower-pd* ⇒ *'a lower-pd*
(**infixl** +♭ *65*) **where**
xs +♭ *ys* == *lower-plus*·*xs*·*ys*

syntax
-lower-pd :: *args* ⇒ *'a lower-pd* ({-}♭)

translations
{*x*,*xs*}♭ == {*x*}♭ +♭ {*xs*}♭
{*x*}♭ == *CONST lower-unit*·*x*

lemma *lower-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♭ = *lower-principal* (*PDUnit a*)
unfolding *lower-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal PDUnit-lower-mono*)

lemma *lower-plus-principal* [*simp*]:
lower-principal t +♭ *lower-principal u* = *lower-principal* (*PDPlus t u*)
unfolding *lower-plus-def*
by (*simp add*: *lower-pd.basis-fun-principal*
lower-pd.basis-fun-mono PDPlus-lower-mono)

lemma *approx-lower-unit* [*simp*]:
approx n·{*x*}♭ = {*approx n*·*x*}♭
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-lower-plus* [*simp*]:
approx n·(*xs* +♭ *ys*) = (*approx n*·*xs*) +♭ (*approx n*·*ys*)
by (*induct xs ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*, *simp*)

interpretation *lower-add*!: *semilattice lower-add* **proof**
fix *xs ys zs* :: *'a lower-pd*
show (*xs* +♭ *ys*) +♭ *zs* = *xs* +♭ (*ys* +♭ *zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *lower-pd.principal-induct2*, *simp*, *simp*)
apply (*rule-tac x*=*zs* **in** *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done
show *xs* +♭ *ys* = *ys* +♭ *xs*
apply (*induct xs ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done
show *xs* +♭ *xs* = *xs*
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done
qed

lemmas *lower-plus-assoc* = *lower-add.assoc*
lemmas *lower-plus-commute* = *lower-add.commute*
lemmas *lower-plus-absorb* = *lower-add.idem*
lemmas *lower-plus-left-commute* = *lower-add.left-commute*
lemmas *lower-plus-left-absorb* = *lower-add.left-idem*

Useful for *simp add*: *lower-plus-ac*

lemmas *lower-plus-ac* =
lower-plus-assoc lower-plus-commute lower-plus-left-commute

Useful for *simp only*: *lower-plus-aci*

lemmas *lower-plus-aci* =
  lower-plus-ac lower-plus-absorb lower-plus-left-absorb

lemma *lower-plus-below1*: $xs \sqsubseteq xs +\flat ys$
apply (*induct xs ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-lower-le*)
done

lemma *lower-plus-below2*: $ys \sqsubseteq xs +\flat ys$
by (*subst lower-plus-commute*, *rule lower-plus-below1*)

lemma *lower-plus-least*: $[\![xs \sqsubseteq zs; ys \sqsubseteq zs]\!] \Longrightarrow xs +\flat ys \sqsubseteq zs$
apply (*subst lower-plus-absorb* [*of zs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *lower-plus-below-iff*:
  $xs +\flat ys \sqsubseteq zs \longleftrightarrow xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
apply *safe*
apply (*erule below-trans* [*OF lower-plus-below1*])
apply (*erule below-trans* [*OF lower-plus-below2*])
apply (*erule* (*1*) *lower-plus-least*)
done

lemma *lower-unit-below-plus-iff*:
  $\{x\}\flat \sqsubseteq ys +\flat zs \longleftrightarrow \{x\}\flat \sqsubseteq ys \vee \{x\}\flat \sqsubseteq zs$
 apply (*rule iffI*)
  apply (*subgoal-tac*
    adm ($\lambda f.\ f{\cdot}\{x\}\flat \sqsubseteq f{\cdot}ys \vee f{\cdot}\{x\}\flat \sqsubseteq f{\cdot}zs$))
   apply (*drule admD*, *rule chain-approx*)
    apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
    apply (*cut-tac x*=*approx i*·*x* **in** *compact-basis.compact-imp-principal*, *simp*)
    apply (*cut-tac x*=*approx i*·*ys* **in** *lower-pd.compact-imp-principal*, *simp*)
    apply (*cut-tac x*=*approx i*·*zs* **in** *lower-pd.compact-imp-principal*, *simp*)
    apply (*clarify*, *simp add*: *lower-le-PDUnit-PDPlus-iff*)
   apply *simp*
  apply *simp*
 apply (*erule disjE*)
  apply (*erule below-trans* [*OF - lower-plus-below1*])
 apply (*erule below-trans* [*OF - lower-plus-below2*])
done

lemma *lower-unit-below-iff* [*simp*]: $\{x\}\flat \sqsubseteq \{y\}\flat \longleftrightarrow x \sqsubseteq y$
 apply (*rule iffI*)
  apply (*rule profinite-below-ext*)
  apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*, *simp*)
  apply (*cut-tac x*=*approx i*·*x* **in** *compact-basis.compact-imp-principal*, *simp*)
  apply (*cut-tac x*=*approx i*·*y* **in** *compact-basis.compact-imp-principal*, *simp*)
  apply *clarsimp*

apply (*erule monofun-cfun-arg*)
done

lemmas *lower-pd-below-simps* =
lower-unit-below-iff
lower-plus-below-iff
lower-unit-below-plus-iff

lemma *lower-unit-eq-iff* [*simp*]: $\{x\}\flat = \{y\}\flat \longleftrightarrow x = y$
by (*simp add*: *po-eq-conv*)

lemma *lower-unit-strict* [*simp*]: $\{\bot\}\flat = \bot$
unfolding *inst-lower-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *lower-unit-strict-iff* [*simp*]: $\{x\}\flat = \bot \longleftrightarrow x = \bot$
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-unit-eq-iff*)

lemma *lower-plus-strict-iff* [*simp*]:
$xs +\flat ys = \bot \longleftrightarrow xs = \bot \wedge ys = \bot$
apply *safe*
apply (*rule UU-I*, *erule subst*, *rule lower-plus-below1*)
apply (*rule UU-I*, *erule subst*, *rule lower-plus-below2*)
apply (*rule lower-plus-absorb*)
done

lemma *lower-plus-strict1* [*simp*]: $\bot +\flat ys = ys$
apply (*rule below-antisym* [*OF - lower-plus-below2*])
apply (*simp add*: *lower-plus-least*)
done

lemma *lower-plus-strict2* [*simp*]: $xs +\flat \bot = xs$
apply (*rule below-antisym* [*OF - lower-plus-below1*])
apply (*simp add*: *lower-plus-least*)
done

lemma *compact-lower-unit-iff* [*simp*]: *compact* $\{x\}\flat \longleftrightarrow$ *compact* x
unfolding *profinite-compact-iff* **by** *simp*

lemma *compact-lower-plus* [*simp*]:
$[\![$*compact* xs; *compact* $ys]\!] \Longrightarrow$ *compact* $(xs +\flat ys)$
by (*auto dest*!: *lower-pd.compact-imp-principal*)

30.4 Induction rules

lemma *lower-pd-induct1*:
assumes P: *adm* P
assumes *unit*: $\bigwedge x.\ P\ \{x\}\flat$
assumes *insert*:
$\bigwedge x\ ys.\ [\![P\ \{x\}\flat; P\ ys]\!] \Longrightarrow P\ (\{x\}\flat +\flat ys)$

shows P (xs::$'a$ *lower-pd*)
apply (*induct xs rule*: *lower-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct1*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*]
lower-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *lower-pd-induct*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\flat$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\flat\ ys)$
shows P (xs::$'a$ *lower-pd*)
apply (*induct xs rule*: *lower-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only*: *lower-plus-principal* [*symmetric*] *plus*)
done

30.5 Monadic bind

definition
lower-bind-basis ::
$'a$ *pd-basis* $\Rightarrow$ ($'a \to 'b$ *lower-pd*) $\to$ $'b$ *lower-pd* **where**
lower-bind-basis = *fold-pd*
($\lambda a.\ \Lambda\ f.\ f{\cdot}$(*Rep-compact-basis a*))
($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f\ +\flat\ y{\cdot}f$)

lemma *ACI-lower-bind*:
class.ab-semigroup-idem-mult ($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f\ +\flat\ y{\cdot}f$)
apply *unfold-locales*
apply (*simp add*: *lower-plus-assoc*)
apply (*simp add*: *lower-plus-commute*)
apply (*simp add*: *eta-cfun*)
done

lemma *lower-bind-basis-simps* [*simp*]:
lower-bind-basis (*PDUnit a*) =
($\Lambda\ f.\ f{\cdot}$(*Rep-compact-basis a*))
lower-bind-basis (*PDPlus t u*) =
($\Lambda\ f.$ *lower-bind-basis* $t{\cdot}f\ +\flat$ *lower-bind-basis* $u{\cdot}f$)
unfolding *lower-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-lower-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-lower-bind*])
done

lemma *lower-bind-basis-mono*:
t $\leq\flat$ *u* $\Longrightarrow$ *lower-bind-basis t* $\sqsubseteq$ *lower-bind-basis u*
unfolding *expand-cfun-below*
apply (*erule lower-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *rev-below-trans* [*OF lower-plus-below1*])
apply (*simp add*: *lower-plus-below-iff*)
done

definition
lower-bind :: *'a lower-pd* $\rightarrow$ (*'a* $\rightarrow$ *'b lower-pd*) $\rightarrow$ *'b lower-pd* **where**
lower-bind = *lower-pd.basis-fun lower-bind-basis*

lemma *lower-bind-principal* [*simp*]:
lower-bind·(*lower-principal t*) = *lower-bind-basis t*
unfolding *lower-bind-def*
apply (*rule lower-pd.basis-fun-principal*)
apply (*erule lower-bind-basis-mono*)
done

lemma *lower-bind-unit* [*simp*]:
lower-bind·{*x*}$\flat$·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *lower-bind-plus* [*simp*]:
lower-bind·(*xs* +$\flat$ *ys*)·*f* = *lower-bind*·*xs*·*f* +$\flat$ *lower-bind*·*ys*·*f*
by (*induct xs ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *lower-bind-strict* [*simp*]: *lower-bind*·$\bot$·*f* = *f*·$\bot$
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-bind-unit*)

30.6 Map and join

definition
lower-map :: (*'a* $\rightarrow$ *'b*) $\rightarrow$ *'a lower-pd* $\rightarrow$ *'b lower-pd* **where**
lower-map = (Λ *f xs*. *lower-bind*·*xs*·(Λ *x*. {*f*·*x*}$\flat$))

definition
lower-join :: *'a lower-pd lower-pd* $\rightarrow$ *'a lower-pd* **where**
lower-join = (Λ *xss*. *lower-bind*·*xss*·(Λ *xs*. *xs*))

lemma *lower-map-unit* [*simp*]:
lower-map·*f*·{*x*}$\flat$ = {*f*·*x*}$\flat$
unfolding *lower-map-def* **by** *simp*

lemma *lower-map-plus* [*simp*]:
lower-map·*f*·(*xs* +$\flat$ *ys*) = *lower-map*·*f*·*xs* +$\flat$ *lower-map*·*f*·*ys*
unfolding *lower-map-def* **by** *simp*

lemma *lower-join-unit* [*simp*]:
  lower-join·{*xs*}♭ = *xs*
unfolding *lower-join-def* **by** *simp*

lemma *lower-join-plus* [*simp*]:
  lower-join·(*xss* +♭ *yss*) = *lower-join*·*xss* +♭ *lower-join*·*yss*
unfolding *lower-join-def* **by** *simp*

lemma *lower-map-ident*: *lower-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-map-ID*: *lower-map*·*ID* = *ID*
by (*simp add*: *expand-cfun-eq ID-def lower-map-ident*)

lemma *lower-map-map*:
  lower-map·*f*·(*lower-map*·*g*·*xs*) = *lower-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-unit*:
  lower-join·(*lower-map*·*lower-unit*·*xs*) = *xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-join*:
  lower-join·(*lower-map*·*lower-join*·*xsss*) = *lower-join*·(*lower-join*·*xsss*)
by (*induct xsss rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-map*:
  lower-join·(*lower-map*·(*lower-map*·*f*)·*xss*) =
   lower-map·*f*·(*lower-join*·*xss*)
by (*induct xss rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-map-approx*: *lower-map*·(*approx n*)·*xs* = *approx n*·*xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *ep-pair-lower-map*: *ep-pair e p* ⟹ *ep-pair* (*lower-map*·*e*) (*lower-map*·*p*)
apply *default*
apply (*induct-tac x rule*: *lower-pd-induct*, *simp-all add*: *ep-pair.e-inverse*)
apply (*induct-tac y rule*: *lower-pd-induct*)
apply (*simp-all add*: *ep-pair.e-p-below monofun-cfun*)
done

lemma *deflation-lower-map*: *deflation d* ⟹ *deflation* (*lower-map*·*d*)
apply *default*
apply (*induct-tac x rule*: *lower-pd-induct*, *simp-all add*: *deflation.idem*)
apply (*induct-tac x rule*: *lower-pd-induct*)
apply (*simp-all add*: *deflation.below monofun-cfun*)
done

end

31 ConvexPD: Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

31.1 Basis preorder

definition
convex-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♮ *50*) **where**
convex-le = (λ*u v*. *u* ≤♯ *v* ∧ *u* ≤♭ *v*)

lemma *convex-le-refl* [*simp*]: *t* ≤♮ *t*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-refl lower-le-refl*)

lemma *convex-le-trans*: ⟦*t* ≤♮ *u*; *u* ≤♮ *v*⟧ ⟹ *t* ≤♮ *v*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-trans lower-le-trans*)

interpretation *convex-le*: *preorder convex-le*
by (*rule preorder.intro*, *rule convex-le-refl*, *rule convex-le-trans*)

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* ≤♮ *t*
unfolding *convex-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-convex-mono*: *x* ⊑ *y* ⟹ *PDUnit x* ≤♮ *PDUnit y*
unfolding *convex-le-def* **by** (*fast intro*: *PDUnit-upper-mono PDUnit-lower-mono*)

lemma *PDPlus-convex-mono*: ⟦*s* ≤♮ *t*; *u* ≤♮ *v*⟧ ⟹ *PDPlus s u* ≤♮ *PDPlus t v*
unfolding *convex-le-def* **by** (*fast intro*: *PDPlus-upper-mono PDPlus-lower-mono*)

lemma *convex-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* ≤♮ *PDUnit b*) = *a* ⊑ *b*
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit* **by** *fast*

lemma *convex-le-PDUnit-lemma1*:
(*PDUnit a* ≤♮ *t*) = (∀ *b*∈*Rep-pd-basis t*. *a* ⊑ *b*)
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit*
using *Rep-pd-basis-nonempty* [*of t*, *folded ex-in-conv*] **by** *fast*

lemma *convex-le-PDUnit-PDPlus-iff* [*simp*]:
(*PDUnit a* ≤♮ *PDPlus t u*) = (*PDUnit a* ≤♮ *t* ∧ *PDUnit a* ≤♮ *u*)
unfolding *convex-le-PDUnit-lemma1 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDUnit-lemma2*:
(*t* ≤♮ *PDUnit b*) = (∀ *a*∈*Rep-pd-basis t*. *a* ⊑ *b*)
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit*
using *Rep-pd-basis-nonempty* [*of t*, *folded ex-in-conv*] **by** *fast*

lemma *convex-le-PDPlus-PDUnit-iff* [*simp*]:
(*PDPlus t u* $\leq\natural$ *PDUnit a*) = (*t* $\leq\natural$ *PDUnit a* $\wedge$ *u* $\leq\natural$ *PDUnit a*)
unfolding *convex-le-PDUnit-lemma2 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDPlus-lemma*:
assumes *z*: *PDPlus t u* $\leq\natural$ *z*
shows $\exists$ *v w*. *z* = *PDPlus v w* $\wedge$ *t* $\leq\natural$ *v* $\wedge$ *u* $\leq\natural$ *w*
proof (*intro exI conjI*)
let *?A* = {*b*$\in$*Rep-pd-basis z*. $\exists$ *a*$\in$*Rep-pd-basis t*. *a* $\sqsubseteq$ *b*}
let *?B* = {*b*$\in$*Rep-pd-basis z*. $\exists$ *a*$\in$*Rep-pd-basis u*. *a* $\sqsubseteq$ *b*}
let *?v* = *Abs-pd-basis ?A*
let *?w* = *Abs-pd-basis ?B*
have *Rep-v*: *Rep-pd-basis ?v* = *?A*
apply (*rule Abs-pd-basis-inverse*)
apply (*rule Rep-pd-basis-nonempty* [*of t*, *folded ex-in-conv*, *THEN exE*])
apply (*cut-tac z*, *simp only*: *convex-le-def lower-le-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *bspec*, *simp add*: *Rep-PDPlus*, *erule bexE*)
apply (*simp add*: *pd-basis-def*)
apply *fast*
done
have *Rep-w*: *Rep-pd-basis ?w* = *?B*
apply (*rule Abs-pd-basis-inverse*)
apply (*rule Rep-pd-basis-nonempty* [*of u*, *folded ex-in-conv*, *THEN exE*])
apply (*cut-tac z*, *simp only*: *convex-le-def lower-le-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *bspec*, *simp add*: *Rep-PDPlus*, *erule bexE*)
apply (*simp add*: *pd-basis-def*)
apply *fast*
done
show *z* = *PDPlus ?v ?w*
apply (*insert z*)
apply (*simp add*: *convex-le-def*, *erule conjE*)
apply (*simp add*: *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus*)
apply (*simp add*: *Rep-v Rep-w*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp only*: *upper-le-def*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*simp add*: *Rep-PDPlus*)
apply *fast*
apply *fast*
done
show *t* $\leq\natural$ *?v u* $\leq\natural$ *?w*
apply (*insert z*)
apply (*simp-all add*: *convex-le-def upper-le-def lower-le-def Rep-PDPlus Rep-v Rep-w*)
apply *fast+*
done
qed

lemma *convex-le-induct* [*induct set*: *convex-le*]:
  assumes *le*: $t \leq\natural u$
  assumes *2*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ u\ v]\!] \Longrightarrow P\ t\ v$
  assumes *3*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
  assumes *4*: $\bigwedge t\ u\ v\ w.\ [\![P\ t\ v;\ P\ u\ w]\!] \Longrightarrow P\ (PDPlus\ t\ u)\ (PDPlus\ v\ w)$
  shows *P t u*
using *le* **apply** (*induct t arbitrary*: *u rule*: *pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac u rule*: *pd-basis-induct1*)
apply (*simp add*: *3*)
apply (*simp*, *clarify*, *rename-tac a b t*)
apply (*subgoal-tac P* (*PDPlus* (*PDUnit a*) (*PDUnit a*)) (*PDPlus* (*PDUnit b*) *t*))
apply (*simp add*: *PDPlus-absorb*)
apply (*erule* (*1*) *4* [*OF 3*])
apply (*drule convex-le-PDPlus-lemma*, *clarify*)
apply (*simp add*: *4*)
done

lemma *pd-take-convex-chain*:
  pd-take n t $\leq\natural$ *pd-take* (*Suc n*) *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-chain*)
apply (*simp add*: *PDPlus-convex-mono*)
done

lemma *pd-take-convex-le*: *pd-take i t* $\leq\natural$ *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-less*)
apply (*simp add*: *PDPlus-convex-mono*)
done

lemma *pd-take-convex-mono*:
  t $\leq\natural$ *u* $\Longrightarrow$ *pd-take n t* $\leq\natural$ *pd-take n u*
apply (*erule convex-le-induct*)
apply (*erule* (*1*) *convex-le-trans*)
apply (*simp add*: *compact-basis.take-mono*)
apply (*simp add*: *PDPlus-convex-mono*)
done

31.2 Type definition

typedef (**open**) *'a convex-pd* =
  {*S*::*'a pd-basis set*. *convex-le.ideal S*}
by (*fast intro*: *convex-le.ideal-principal*)

instantiation *convex-pd* :: (*profinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-convex-pd* $x \subseteq$ *Rep-convex-pd* y

instance ..
end

instance *convex-pd* :: (*profinite*) *po*
by (*rule convex-le.typedef-ideal-po*
[*OF type-definition-convex-pd below-convex-pd-def*])

instance *convex-pd* :: (*profinite*) *cpo*
by (*rule convex-le.typedef-ideal-cpo*
[*OF type-definition-convex-pd below-convex-pd-def*])

lemma *Rep-convex-pd-lub*:
chain $Y \Longrightarrow$ *Rep-convex-pd* $(\bigsqcup i.\ Y\ i) = (\bigcup i.$ *Rep-convex-pd* $(Y\ i))$
by (*rule convex-le.typedef-ideal-rep-contlub*
[*OF type-definition-convex-pd below-convex-pd-def*])

lemma *ideal-Rep-convex-pd*: *convex-le.ideal* (*Rep-convex-pd xs*)
by (*rule Rep-convex-pd* [*unfolded mem-Collect-eq*])

definition
convex-principal :: *'a pd-basis* $\Rightarrow$ *'a convex-pd* **where**
convex-principal $t =$ *Abs-convex-pd* $\{u.\ u \leq\natural t\}$

lemma *Rep-convex-principal*:
Rep-convex-pd (*convex-principal* t) $= \{u.\ u \leq\natural t\}$
unfolding *convex-principal-def*
by (*simp add*: *Abs-convex-pd-inverse convex-le.ideal-principal*)

interpretation *convex-pd*:
ideal-completion convex-le pd-take convex-principal Rep-convex-pd
apply *unfold-locales*
apply (*rule pd-take-convex-le*)
apply (*rule pd-take-idem*)
apply (*erule pd-take-convex-mono*)
apply (*rule pd-take-convex-chain*)
apply (*rule finite-range-pd-take*)
apply (*rule pd-take-covers*)
apply (*rule ideal-Rep-convex-pd*)
apply (*erule Rep-convex-pd-lub*)
apply (*rule Rep-convex-principal*)
apply (*simp only*: *below-convex-pd-def*)
done

Convex powerdomain is pointed

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*
by (*induct ys rule*: *convex-pd.principal-induct*, *simp*, *simp*)

instance *convex-pd* :: (*bifinite*) *pcpo*
by *intro-classes* (*fast intro*: *convex-pd-minimal*)

lemma *inst-convex-pd-pcpo*: $\bot$ = *convex-principal* (*PDUnit compact-bot*)
by (*rule convex-pd-minimal* [*THEN UU-I*, *symmetric*])

Convex powerdomain is profinite

instantiation *convex-pd* :: (*profinite*) *profinite*
begin

definition
approx-convex-pd-def: *approx* = *convex-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-convex-pd-def*)
apply (*rule convex-pd.chain-completion-approx*)
apply (*rule convex-pd.lub-completion-approx*)
apply (*rule convex-pd.completion-approx-idem*)
apply (*rule convex-pd.finite-fixes-completion-approx*)
done

end

instance *convex-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-convex-principal* [*simp*]:
approx $n{\cdot}$(*convex-principal* t) = *convex-principal* (*pd-take* n t)
unfolding *approx-convex-pd-def*
by (*rule convex-pd.completion-approx-principal*)

lemma *approx-eq-convex-principal*:
$\exists\, t{\in}$*Rep-convex-pd xs*. *approx* $n{\cdot}$*xs* = *convex-principal* (*pd-take* n t)
unfolding *approx-convex-pd-def*
by (*rule convex-pd.completion-approx-eq-principal*)

31.3 Monadic unit and plus

definition
convex-unit :: $'a \to$ $'a$ *convex-pd* **where**
convex-unit = *compact-basis.basis-fun* (λa. *convex-principal* (*PDUnit* a))

definition
convex-plus :: $'a$ *convex-pd* $\to$ $'a$ *convex-pd* $\to$ $'a$ *convex-pd* **where**
convex-plus = *convex-pd.basis-fun* (λt. *convex-pd.basis-fun* (λu.
convex-principal (*PDPlus* t u)))

abbreviation
convex-add :: $'a$ *convex-pd* $\Rightarrow$ $'a$ *convex-pd* $\Rightarrow$ $'a$ *convex-pd*

(**infixl** +♮ *65*) **where**
xs +♮ *ys* == *convex-plus·xs·ys*

syntax
-convex-pd :: *args* ⇒ *'a convex-pd* ({-}♮)

translations
{*x*,*xs*}♮ == {*x*}♮ +♮ {*xs*}♮
{*x*}♮ == *CONST convex-unit·x*

lemma *convex-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♮ = *convex-principal* (*PDUnit a*)
unfolding *convex-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal PDUnit-convex-mono*)

lemma *convex-plus-principal* [*simp*]:
convex-principal t +♮ *convex-principal u* = *convex-principal* (*PDPlus t u*)
unfolding *convex-plus-def*
by (*simp add*: *convex-pd.basis-fun-principal*
convex-pd.basis-fun-mono PDPlus-convex-mono)

lemma *approx-convex-unit* [*simp*]:
approx n·{*x*}♮ = {*approx n·x*}♮
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-convex-plus* [*simp*]:
approx n·(*xs* +♮ *ys*) = *approx n·xs* +♮ *approx n·ys*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

interpretation *convex-add*!: *semilattice convex-add* **proof**
fix *xs ys zs* :: *'a convex-pd*
show (*xs* +♮ *ys*) +♮ *zs* = *xs* +♮ (*ys* +♮ *zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *convex-pd.principal-induct2*, *simp*, *simp*)
apply (*rule-tac x=zs* **in** *convex-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done
show *xs* +♮ *ys* = *ys* +♮ *xs*
apply (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done
show *xs* +♮ *xs* = *xs*
apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done
qed

lemmas *convex-plus-assoc* = *convex-add.assoc*

lemmas *convex-plus-commute = convex-add.commute*
lemmas *convex-plus-absorb = convex-add.idem*
lemmas *convex-plus-left-commute = convex-add.left-commute*
lemmas *convex-plus-left-absorb = convex-add.left-idem*

Useful for *simp add*: *convex-plus-ac*

lemmas *convex-plus-ac =*
convex-plus-assoc convex-plus-commute convex-plus-left-commute

Useful for *simp only*: *convex-plus-aci*

lemmas *convex-plus-aci =*
convex-plus-ac convex-plus-absorb convex-plus-left-absorb

lemma *convex-unit-below-plus-iff* [*simp*]:
$\{x\}\natural \sqsubseteq ys +\natural zs \longleftrightarrow \{x\}\natural \sqsubseteq ys \wedge \{x\}\natural \sqsubseteq zs$
apply (*rule iffI*)
apply (*subgoal-tac*
adm ($\lambda f.\ f{\cdot}\{x\}\natural \sqsubseteq f{\cdot}ys \wedge f{\cdot}\{x\}\natural \sqsubseteq f{\cdot}zs$))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x=approx i·x* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·ys* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·zs* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*clarify*, *simp*)
apply *simp*
apply *simp*
apply (*erule conjE*)
apply (*subst convex-plus-absorb* [*of* $\{x\}\natural$, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *convex-plus-below-unit-iff* [*simp*]:
$xs +\natural ys \sqsubseteq \{z\}\natural \longleftrightarrow xs \sqsubseteq \{z\}\natural \wedge ys \sqsubseteq \{z\}\natural$
apply (*rule iffI*)
apply (*subgoal-tac*
adm ($\lambda f.\ f{\cdot}xs \sqsubseteq f{\cdot}\{z\}\natural \wedge f{\cdot}ys \sqsubseteq f{\cdot}\{z\}\natural$))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x=approx i·xs* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·ys* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·z* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*clarify*, *simp*)
apply *simp*
apply *simp*
apply (*erule conjE*)
apply (*subst convex-plus-absorb* [*of* $\{z\}\natural$, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *convex-unit-below-iff* [*simp*]: $\{x\}\natural \sqsubseteq \{y\}\natural \longleftrightarrow x \sqsubseteq y$
apply (*rule iffI*)
apply (*rule profinite-below-ext*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
apply (*cut-tac x=approx i·x* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·y* **in** *compact-basis.compact-imp-principal*, *simp*)
apply *clarsimp*
apply (*erule monofun-cfun-arg*)
done

lemma *convex-unit-eq-iff* [*simp*]: $\{x\}\natural = \{y\}\natural \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *convex-unit-strict* [*simp*]: $\{\bot\}\natural = \bot$
unfolding *inst-convex-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *convex-unit-strict-iff* [*simp*]: $\{x\}\natural = \bot \longleftrightarrow x = \bot$
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-unit-eq-iff*)

lemma *compact-convex-unit-iff* [*simp*]:
compact $\{x\}\natural \longleftrightarrow$ *compact x*
unfolding *profinite-compact-iff* **by** *simp*

lemma *compact-convex-plus* [*simp*]:
$[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* (*xs* $+\natural$ *ys*)
by (*auto dest*!: *convex-pd.compact-imp-principal*)

31.4 Induction rules

lemma *convex-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.$ *P* $\{x\}\natural$
assumes *insert*: $\bigwedge x$ *ys*. $[\![P\ \{x\}\natural; P\ ys]\!] \Longrightarrow P\ (\{x\}\natural +\natural ys)$
shows *P* (*xs*::'*a convex-pd*)
apply (*induct xs rule*: *convex-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct1*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*]
convex-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *convex-pd-induct*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.$ *P* $\{x\}\natural$
assumes *plus*: $\bigwedge$*xs ys*. $[\![P\ xs; P\ ys]\!] \Longrightarrow P\ (xs +\natural ys)$
shows *P* (*xs*::'*a convex-pd*)
apply (*induct xs rule*: *convex-pd.principal-induct*, *rule P*)

apply (*induct-tac a rule*: *pd-basis-induct*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only*: *convex-plus-principal* [*symmetric*] *plus*)
done

31.5 Monadic bind

definition
convex-bind-basis ::
'a pd-basis ⇒ (*'a* → *'b convex-pd*) → *'b convex-pd* **where**
convex-bind-basis = *fold-pd*
(λ*a*. Λ *f*. *f*·(*Rep-compact-basis a*))
(λ*x y*. Λ *f*. *x*·*f* +♮ *y*·*f*)

lemma *ACI-convex-bind*:
class.ab-semigroup-idem-mult (λ*x y*. Λ *f*. *x*·*f* +♮ *y*·*f*)
apply *unfold-locales*
apply (*simp add*: *convex-plus-assoc*)
apply (*simp add*: *convex-plus-commute*)
apply (*simp add*: *eta-cfun*)
done

lemma *convex-bind-basis-simps* [*simp*]:
convex-bind-basis (*PDUnit a*) =
(Λ *f*. *f*·(*Rep-compact-basis a*))
convex-bind-basis (*PDPlus t u*) =
(Λ *f*. *convex-bind-basis t*·*f* +♮ *convex-bind-basis u*·*f*)
unfolding *convex-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-convex-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-convex-bind*])
done

lemma *monofun-LAM*:
⟦*cont f*; *cont g*; ⋀*x*. *f x* ⊑ *g x*⟧ ⟹ (Λ *x*. *f x*) ⊑ (Λ *x*. *g x*)
by (*simp add*: *expand-cfun-below*)

lemma *convex-bind-basis-mono*:
t ≤♮ *u* ⟹ *convex-bind-basis t* ⊑ *convex-bind-basis u*
apply (*erule convex-le-induct*)
apply (*erule* (*1*) *below-trans*)
apply (*simp add*: *monofun-LAM monofun-cfun*)
apply (*simp add*: *monofun-LAM monofun-cfun*)
done

definition
convex-bind :: *'a convex-pd* → (*'a* → *'b convex-pd*) → *'b convex-pd* **where**
convex-bind = *convex-pd.basis-fun convex-bind-basis*

lemma *convex-bind-principal* [*simp*]:
convex-bind·(*convex-principal t*) = *convex-bind-basis t*
unfolding *convex-bind-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*erule convex-bind-basis-mono*)
done

lemma *convex-bind-unit* [*simp*]:
convex-bind·{*x*}♮·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-bind-plus* [*simp*]:
convex-bind·(*xs* +♮ *ys*)·*f* = *convex-bind*·*xs*·*f* +♮ *convex-bind*·*ys*·*f*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *convex-bind-strict* [*simp*]: *convex-bind*·⊥·*f* = *f*·⊥
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-bind-unit*)

31.6 Map and join

definition
convex-map :: ('*a* → '*b*) → '*a convex-pd* → '*b convex-pd* **where**
convex-map = (Λ *f xs*. *convex-bind*·*xs*·(Λ *x*. {*f*·*x*}♮))

definition
convex-join :: '*a convex-pd convex-pd* → '*a convex-pd* **where**
convex-join = (Λ *xss*. *convex-bind*·*xss*·(Λ *xs*. *xs*))

lemma *convex-map-unit* [*simp*]:
convex-map·*f*·(*convex-unit*·*x*) = *convex-unit*·(*f*·*x*)
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-plus* [*simp*]:
convex-map·*f*·(*xs* +♮ *ys*) = *convex-map*·*f*·*xs* +♮ *convex-map*·*f*·*ys*
unfolding *convex-map-def* **by** *simp*

lemma *convex-join-unit* [*simp*]:
convex-join·{*xs*}♮ = *xs*
unfolding *convex-join-def* **by** *simp*

lemma *convex-join-plus* [*simp*]:
convex-join·(*xss* +♮ *yss*) = *convex-join*·*xss* +♮ *convex-join*·*yss*
unfolding *convex-join-def* **by** *simp*

lemma *convex-map-ident*: *convex-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-map-ID*: *convex-map*·*ID* = *ID*
by (*simp add*: *expand-cfun-eq ID-def convex-map-ident*)

lemma *convex-map-map*:
convex-map·f·(convex-map·g·xs) = convex-map·(Λ x. f·(g·x))·xs
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-unit*:
convex-join·(convex-map·convex-unit·xs) = xs
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-join*:
convex-join·(convex-map·convex-join·xsss) = convex-join·(convex-join·xsss)
by (*induct xsss rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-map*:
convex-join·(convex-map·(convex-map·f)·xss) =
convex-map·f·(convex-join·xss)
by (*induct xss rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-map-approx*: *convex-map·(approx n)·xs = approx n·xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *ep-pair-convex-map*:
ep-pair e p ⟹ ep-pair (convex-map·e) (convex-map·p)
apply *default*
apply (*induct-tac x rule*: *convex-pd-induct*, *simp-all add*: *ep-pair.e-inverse*)
apply (*induct-tac y rule*: *convex-pd-induct*)
apply (*simp-all add*: *ep-pair.e-p-below monofun-cfun*)
done

lemma *deflation-convex-map*: *deflation d ⟹ deflation (convex-map·d)*
apply *default*
apply (*induct-tac x rule*: *convex-pd-induct*, *simp-all add*: *deflation.idem*)
apply (*induct-tac x rule*: *convex-pd-induct*)
apply (*simp-all add*: *deflation.below monofun-cfun*)
done

31.7 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: *t ≤♮ u ⟹ t ≤♯ u*
unfolding *convex-le-def* **by** *simp*

definition
convex-to-upper :: *'a convex-pd → 'a upper-pd* **where**
convex-to-upper = convex-pd.basis-fun upper-principal

lemma *convex-to-upper-principal* [*simp*]:
convex-to-upper·(convex-principal t) = upper-principal t
unfolding *convex-to-upper-def*

apply (*rule convex-pd.basis-fun-principal*)
apply (*rule upper-pd.principal-mono*)
apply (*erule convex-le-imp-upper-le*)
done

lemma *convex-to-upper-unit* [*simp*]:
convex-to-upper·{*x*}♮ = {*x*}♯
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-to-upper-plus* [*simp*]:
convex-to-upper·(*xs* +♮ *ys*) = *convex-to-upper*·*xs* +♯ *convex-to-upper*·*ys*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *approx-convex-to-upper*:
approx i·(*convex-to-upper*·*xs*) = *convex-to-upper*·(*approx i*·*xs*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-upper-bind* [*simp*]:
convex-to-upper·(*convex-bind*·*xs*·*f*) =
upper-bind·(*convex-to-upper*·*xs*)·(*convex-to-upper oo f*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-upper-map* [*simp*]:
convex-to-upper·(*convex-map*·*f*·*xs*) = *upper-map*·*f*·(*convex-to-upper*·*xs*)
by (*simp add*: *convex-map-def upper-map-def cfcomp-LAM*)

lemma *convex-to-upper-join* [*simp*]:
convex-to-upper·(*convex-join*·*xss*) =
upper-bind·(*convex-to-upper*·*xss*)·*convex-to-upper*
by (*simp add*: *convex-join-def upper-join-def cfcomp-LAM eta-cfun*)

Convex to lower

lemma *convex-le-imp-lower-le*: *t* ≤♮ *u* ⟹ *t* ≤♭ *u*
unfolding *convex-le-def* **by** *simp*

definition
convex-to-lower :: *'a convex-pd* → *'a lower-pd* **where**
convex-to-lower = *convex-pd.basis-fun lower-principal*

lemma *convex-to-lower-principal* [*simp*]:
convex-to-lower·(*convex-principal t*) = *lower-principal t*
unfolding *convex-to-lower-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*rule lower-pd.principal-mono*)
apply (*erule convex-le-imp-lower-le*)
done

lemma *convex-to-lower-unit* [*simp*]:
convex-to-lower·{*x*}♮ = {*x*}♭

by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-to-lower-plus* [*simp*]:
convex-to-lower·(*xs* +♮ *ys*) = *convex-to-lower*·*xs* +♭ *convex-to-lower*·*ys*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *approx-convex-to-lower*:
approx i·(*convex-to-lower*·*xs*) = *convex-to-lower*·(*approx i*·*xs*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-lower-bind* [*simp*]:
convex-to-lower·(*convex-bind*·*xs*·*f*) =
lower-bind·(*convex-to-lower*·*xs*)·(*convex-to-lower oo f*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-lower-map* [*simp*]:
convex-to-lower·(*convex-map*·*f*·*xs*) = *lower-map*·*f*·(*convex-to-lower*·*xs*)
by (*simp add*: *convex-map-def lower-map-def cfcomp-LAM*)

lemma *convex-to-lower-join* [*simp*]:
convex-to-lower·(*convex-join*·*xss*) =
lower-bind·(*convex-to-lower*·*xss*)·*convex-to-lower*
by (*simp add*: *convex-join-def lower-join-def cfcomp-LAM eta-cfun*)

Ordering property

lemma *convex-pd-below-iff*:
(*xs* ⊑ *ys*) =
(*convex-to-upper*·*xs* ⊑ *convex-to-upper*·*ys* ∧
convex-to-lower·*xs* ⊑ *convex-to-lower*·*ys*)
apply (*safe elim*!: *monofun-cfun-arg*)
apply (*rule profinite-below-ext*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x*=*approx i*·*xs* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x*=*approx i*·*ys* **in** *convex-pd.compact-imp-principal*, *simp*)
apply *clarify*
apply (*simp add*: *approx-convex-to-upper approx-convex-to-lower convex-le-def*)
done

lemmas *convex-plus-below-plus-iff* =
convex-pd-below-iff [**where** *xs*=*xs* +♮ *ys* **and** *ys*=*zs* +♮ *ws*, *standard*]

lemmas *convex-pd-below-simps* =
convex-unit-below-plus-iff
convex-plus-below-unit-iff
convex-plus-below-plus-iff
convex-unit-below-iff
convex-to-upper-unit
convex-to-upper-plus

convex-to-lower-unit
convex-to-lower-plus
upper-pd-below-simps
lower-pd-below-simps

end

32 Powerdomains: Powerdomains

theory *Powerdomains*
imports *Representable ConvexPD*
begin

32.1 Powerdomains are representable

Upper powerdomain of a representable type is representable.

instantiation *upper-pd* :: (*rep*) *rep*
begin

definition *emb-upper-pd-def*: *emb* = *udom-emb oo upper-map·emb*
definition *prj-upper-pd-def*: *prj* = *upper-map·prj oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-upper-pd-def prj-upper-pd-def*)
apply (*intro ep-pair-comp ep-pair-upper-map ep-pair-emb-prj ep-pair-udom*)
done

end

Lower powerdomain of a representable type is representable.

instantiation *lower-pd* :: (*rep*) *rep*
begin

definition *emb-lower-pd-def*: *emb* = *udom-emb oo lower-map·emb*
definition *prj-lower-pd-def*: *prj* = *lower-map·prj oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-lower-pd-def prj-lower-pd-def*)
apply (*intro ep-pair-comp ep-pair-lower-map ep-pair-emb-prj ep-pair-udom*)
done

end

Convex powerdomain of a representable type is representable.

instantiation *convex-pd* :: (*rep*) *rep*
begin

definition *emb-convex-pd-def*: *emb* = *udom-emb oo convex-map·emb*
definition *prj-convex-pd-def*: *prj* = *convex-map·prj oo udom-prj*

instance
apply (*intro-classes*, *unfold emb-convex-pd-def prj-convex-pd-def*)
apply (*intro ep-pair-comp ep-pair-convex-map ep-pair-emb-prj ep-pair-udom*)
done

end

32.2 Finite deflation lemmas

TODO: move these lemmas somewhere else

lemma *finite-compact-range-imp-finite-range*:
fixes $d :: 'a{::}profinite \to 'b{::}cpo$
assumes *finite* $((\lambda x.\ d{\cdot}x)\ `\ \{x.\ compact\ x\})$
shows *finite* $(range\ (\lambda x.\ d{\cdot}x))$
proof (*rule finite-subset* [*OF* - *prems*])
{
fix $x :: 'a$
have $range\ (\lambda i.\ d{\cdot}(approx\ i{\cdot}x)) \subseteq (\lambda x.\ d{\cdot}x)\ `\ \{x.\ compact\ x\}$
by *auto*
hence *finite* $(range\ (\lambda i.\ d{\cdot}(approx\ i{\cdot}x)))$
using *prems* **by** (*rule finite-subset*)
hence *finite-chain* $(\lambda i.\ d{\cdot}(approx\ i{\cdot}x))$
by (*simp add*: *finite-range-imp-finch*)
hence $\exists i.\ (\bigsqcup i.\ d{\cdot}(approx\ i{\cdot}x)) = d{\cdot}(approx\ i{\cdot}x)$
by (*simp add*: *finite-chain-def maxinch-is-thelub*)
hence $\exists i.\ d{\cdot}x = d{\cdot}(approx\ i{\cdot}x)$
by (*simp add*: *lub-distribs*)
hence $d{\cdot}x \in (\lambda x.\ d{\cdot}x)\ `\ \{x.\ compact\ x\}$
by *auto*
}
thus $range\ (\lambda x.\ d{\cdot}x) \subseteq (\lambda x.\ d{\cdot}x)\ `\ \{x.\ compact\ x\}$
by *clarsimp*
qed

lemma *finite-deflation-upper-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*upper-map·d*)
proof (*intro finite-deflation.intro finite-deflation-axioms.intro*)
interpret *d*: *finite-deflation d* **by** *fact*
have *deflation d* **by** *fact*
thus *deflation* (*upper-map·d*) **by** (*rule deflation-upper-map*)
have *finite* $(range\ (\lambda x.\ d{\cdot}x))$ **by** (*rule d.finite-range*)
hence *finite* $(Rep\text{-}compact\text{-}basis\ -`\ range\ (\lambda x.\ d{\cdot}x))$
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-compact-basis-inject*)
hence *finite* $(Pow\ (Rep\text{-}compact\text{-}basis\ -`\ range\ (\lambda x.\ d{\cdot}x)))$ **by** *simp*
hence *finite* $(Rep\text{-}pd\text{-}basis\ -`\ (Pow\ (Rep\text{-}compact\text{-}basis\ -`\ range\ (\lambda x.\ d{\cdot}x))))$
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-pd-basis-inject*)

hence *finite* (*upper-principal ‘ Rep-pd-basis −‘* (*Pow* (*Rep-compact-basis −‘ range* (*λx. d·x*)))) **by** *simp*
hence *finite* ((*λxs. upper-map·d·xs*) *‘ range upper-principal*)
apply (*rule finite-subset* [*COMP swap-prems-rl*])
apply (*clarsimp*, *rename-tac t*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*] *upper-map-unit*)
apply (*subgoal-tac ∃ b. d·*(*Rep-compact-basis a*) = *Rep-compact-basis b*)
apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDUnit*)
apply (*rule image-eqI*)
apply (*erule sym*)
apply *simp*
apply (*rule exI*)
apply (*rule Abs-compact-basis-inverse* [*symmetric*])
apply (*simp add*: *d.compact*)
apply (*simp only*: *upper-plus-principal* [*symmetric*] *upper-map-plus*)
apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDPlus*)
done
moreover have {*xs*::*′a upper-pd. compact xs*} = *range upper-principal*
by (*auto dest*: *upper-pd.compact-imp-principal*)
ultimately have *finite* ((*λxs. upper-map·d·xs*) *‘* {*xs*::*′a upper-pd. compact xs*})
by *simp*
hence *finite* (*range* (*λxs. upper-map·d·xs*))
by (*rule finite-compact-range-imp-finite-range*)
thus *finite* {*xs. upper-map·d·xs* = *xs*}
by (*rule finite-range-imp-finite-fixes*)
qed

lemma *finite-deflation-lower-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*lower-map·d*)
proof (*intro finite-deflation.intro finite-deflation-axioms.intro*)
interpret *d*: *finite-deflation d* **by** *fact*
have *deflation d* **by** *fact*
thus *deflation* (*lower-map·d*) **by** (*rule deflation-lower-map*)
have *finite* (*range* (*λx. d·x*)) **by** (*rule d.finite-range*)
hence *finite* (*Rep-compact-basis −‘ range* (*λx. d·x*))
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-compact-basis-inject*)
hence *finite* (*Pow* (*Rep-compact-basis −‘ range* (*λx. d·x*))) **by** *simp*
hence *finite* (*Rep-pd-basis −‘* (*Pow* (*Rep-compact-basis −‘ range* (*λx. d·x*))))
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-pd-basis-inject*)
hence *finite* (*lower-principal ‘ Rep-pd-basis −‘* (*Pow* (*Rep-compact-basis −‘ range* (*λx. d·x*)))) **by** *simp*
hence *finite* ((*λxs. lower-map·d·xs*) *‘ range lower-principal*)

apply (*rule finite-subset* [*COMP swap-prems-rl*])
apply (*clarsimp*, *rename-tac t*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *lower-map-unit*)
apply (*subgoal-tac* ∃ *b*. *d*·(*Rep-compact-basis a*) = *Rep-compact-basis b*)
apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDUnit*)
apply (*rule image-eqI*)
apply (*erule sym*)
apply *simp*
apply (*rule exI*)
apply (*rule Abs-compact-basis-inverse* [*symmetric*])
apply (*simp add*: *d.compact*)
apply (*simp only*: *lower-plus-principal* [*symmetric*] *lower-map-plus*)
apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDPlus*)
done
moreover have {*xs*::′*a lower-pd*. *compact xs*} = *range lower-principal*
by (*auto dest*: *lower-pd.compact-imp-principal*)
ultimately have *finite* ((λ*xs*. *lower-map*·*d*·*xs*) ‘ {*xs*::′*a lower-pd*. *compact xs*})
by *simp*
hence *finite* (*range* (λ*xs*. *lower-map*·*d*·*xs*))
by (*rule finite-compact-range-imp-finite-range*)
thus *finite* {*xs*. *lower-map*·*d*·*xs* = *xs*}
by (*rule finite-range-imp-finite-fixes*)
qed

lemma *finite-deflation-convex-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*convex-map*·*d*)
proof (*intro finite-deflation.intro finite-deflation-axioms.intro*)
interpret *d*: *finite-deflation d* **by** *fact*
have *deflation d* **by** *fact*
thus *deflation* (*convex-map*·*d*) **by** (*rule deflation-convex-map*)
have *finite* (*range* (λ*x*. *d*·*x*)) **by** (*rule d.finite-range*)
hence *finite* (*Rep-compact-basis* −‘ *range* (λ*x*. *d*·*x*))
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-compact-basis-inject*)
hence *finite* (*Pow* (*Rep-compact-basis* −‘ *range* (λ*x*. *d*·*x*))) **by** *simp*
hence *finite* (*Rep-pd-basis* −‘ (*Pow* (*Rep-compact-basis* −‘ *range* (λ*x*. *d*·*x*))))
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-pd-basis-inject*)
hence *finite* (*convex-principal* ‘ *Rep-pd-basis* −‘ (*Pow* (*Rep-compact-basis* −‘ *range* (λ*x*. *d*·*x*)))) **by** *simp*
hence *finite* ((λ*xs*. *convex-map*·*d*·*xs*) ‘ *range convex-principal*)
apply (*rule finite-subset* [*COMP swap-prems-rl*])
apply (*clarsimp*, *rename-tac t*)
apply (*induct-tac t rule*: *pd-basis-induct*)

apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*] *convex-map-unit*)
apply (*subgoal-tac* ∃ *b*. *d*·(*Rep-compact-basis a*) = *Rep-compact-basis b*)
apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDUnit*)
apply (*rule image-eqI*)
apply (*erule sym*)
apply *simp*
apply (*rule exI*)
apply (*rule Abs-compact-basis-inverse* [*symmetric*])
apply (*simp add*: *d.compact*)
apply (*simp only*: *convex-plus-principal* [*symmetric*] *convex-map-plus*)
apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDPlus*)
done
moreover have {*xs*::'*a convex-pd*. *compact xs*} = *range convex-principal*
by (*auto dest*: *convex-pd.compact-imp-principal*)
ultimately have *finite* ((λ*xs*. *convex-map*·*d*·*xs*) ' {*xs*::'*a convex-pd*. *compact xs*})
by *simp*
hence *finite* (*range* (λ*xs*. *convex-map*·*d*·*xs*))
by (*rule finite-compact-range-imp-finite-range*)
thus *finite* {*xs*. *convex-map*·*d*·*xs* = *xs*}
by (*rule finite-range-imp-finite-fixes*)
qed

32.3 Deflation combinators

definition *upper-defl* = *TypeRep-fun1 upper-map*
definition *lower-defl* = *TypeRep-fun1 lower-map*
definition *convex-defl* = *TypeRep-fun1 convex-map*

lemma *cast-upper-defl*:
cast·(*upper-defl*·*A*) = *udom-emb oo upper-map*·(*cast*·*A*) *oo udom-prj*
unfolding *upper-defl-def*
apply (*rule cast-TypeRep-fun1*)
apply (*erule finite-deflation-upper-map*)
done

lemma *cast-lower-defl*:
cast·(*lower-defl*·*A*) = *udom-emb oo lower-map*·(*cast*·*A*) *oo udom-prj*
unfolding *lower-defl-def*
apply (*rule cast-TypeRep-fun1*)
apply (*erule finite-deflation-lower-map*)
done

lemma *cast-convex-defl*:
cast·(convex-defl·A) = udom-emb oo convex-map·(cast·A) oo udom-prj
unfolding *convex-defl-def*
apply (*rule cast-TypeRep-fun1*)
apply (*erule finite-deflation-convex-map*)
done

lemma *REP-upper*: *REP('a upper-pd) = upper-defl·REP('a)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)
apply (*simp add*: *cast-REP cast-upper-defl*)
apply (*simp add*: *prj-upper-pd-def*)
apply (*simp add*: *emb-upper-pd-def*)
apply (*simp add*: *upper-map-map cfcomp1*)
done

lemma *REP-lower*: *REP('a lower-pd) = lower-defl·REP('a)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)
apply (*simp add*: *cast-REP cast-lower-defl*)
apply (*simp add*: *prj-lower-pd-def*)
apply (*simp add*: *emb-lower-pd-def*)
apply (*simp add*: *lower-map-map cfcomp1*)
done

lemma *REP-convex*: *REP('a convex-pd) = convex-defl·REP('a)*
apply (*rule cast-eq-imp-eq*, *rule ext-cfun*)
apply (*simp add*: *cast-REP cast-convex-defl*)
apply (*simp add*: *prj-convex-pd-def*)
apply (*simp add*: *emb-convex-pd-def*)
apply (*simp add*: *convex-map-map cfcomp1*)
done

lemma *isodefl-upper*:
isodefl d t ⟹ isodefl (upper-map·d) (upper-defl·t)
apply (*rule isodeflI*)
apply (*simp add*: *cast-upper-defl cast-isodefl*)
apply (*simp add*: *emb-upper-pd-def prj-upper-pd-def*)
apply (*simp add*: *upper-map-map*)
done

lemma *isodefl-lower*:
isodefl d t ⟹ isodefl (lower-map·d) (lower-defl·t)
apply (*rule isodeflI*)
apply (*simp add*: *cast-lower-defl cast-isodefl*)
apply (*simp add*: *emb-lower-pd-def prj-lower-pd-def*)
apply (*simp add*: *lower-map-map*)
done

lemma *isodefl-convex*:
isodefl d t ⟹ isodefl (convex-map·d) (convex-defl·t)

apply (*rule isodeflI*)
apply (*simp add*: *cast-convex-defl cast-isodefl*)
apply (*simp add*: *emb-convex-pd-def prj-convex-pd-def*)
apply (*simp add*: *convex-map-map*)
done

32.4 Domain package setup for powerdomains

setup ⟪
fold Domain-Isomorphism.add-type-constructor
[(@{*type-name upper-pd*}, @{*term upper-defl*}, @{*const-name upper-map*},
@{*thm REP-upper*}, @{*thm isodefl-upper*}, @{*thm upper-map-ID*},
@{*thm deflation-upper-map*}),

(@{*type-name lower-pd*}, @{*term lower-defl*}, @{*const-name lower-map*},
@{*thm REP-lower*}, @{*thm isodefl-lower*}, @{*thm lower-map-ID*},
@{*thm deflation-lower-map*}),

(@{*type-name convex-pd*}, @{*term convex-defl*}, @{*const-name convex-map*},
@{*thm REP-convex*}, @{*thm isodefl-convex*}, @{*thm convex-map-ID*},
@{*thm deflation-convex-map*})]
⟫

end

theory *HOLCF*
imports
Main
Domain
Powerdomains
begin

default-sort *pcpo*

ML ⟪ *path-add* ~~/*src*/*HOLCF*/*Library* ⟫

Legacy theorem names

lemmas *sq-ord-less-eq-trans* = *below-eq-trans*
lemmas *sq-ord-eq-less-trans* = *eq-below-trans*
lemmas *refl-less* = *below-refl*
lemmas *trans-less* = *below-trans*
lemmas *antisym-less* = *below-antisym*
lemmas *antisym-less-inverse* = *below-antisym-inverse*
lemmas *box-less* = *box-below*
lemmas *rev-trans-less* = *rev-below-trans*
lemmas *not-less2not-eq* = *not-below2not-eq*
lemmas *less-UU-iff* = *below-UU-iff*

lemmas *flat-less-iff = flat-below-iff*
lemmas *adm-less = adm-below*
lemmas *adm-not-less = adm-not-below*
lemmas *adm-compact-not-less = adm-compact-not-below*
lemmas *less-fun-def = below-fun-def*
lemmas *expand-fun-less = expand-fun-below*
lemmas *less-fun-ext = below-fun-ext*
lemmas *less-discr-def = below-discr-def*
lemmas *discr-less-eq = discr-below-eq*
lemmas *less-unit-def = below-unit-def*
lemmas *less-cprod-def = below-prod-def*
lemmas *prod-lessI = prod-belowI*
lemmas *Pair-less-iff = Pair-below-iff*
lemmas *fst-less-iff = fst-below-iff*
lemmas *snd-less-iff = snd-below-iff*
lemmas *expand-cfun-less = expand-cfun-below*
lemmas *less-cfun-ext = below-cfun-ext*
lemmas *injection-less = injection-below*
lemmas *approx-less = approx-below*
lemmas *profinite-less-ext = profinite-below-ext*
lemmas *less-up-def = below-up-def*
lemmas *not-Iup-less = not-Iup-below*
lemmas *Iup-less = Iup-below*
lemmas *up-less = up-below*
lemmas *Def-inject-less-eq = Def-below-Def*
lemmas *Def-less-is-eq = Def-below-iff*
lemmas *spair-less-iff = spair-below-iff*
lemmas *less-sprod = below-sprod*
lemmas *spair-less = spair-below*
lemmas *sfst-less-iff = sfst-below-iff*
lemmas *ssnd-less-iff = ssnd-below-iff*
lemmas *fix-least-less = fix-least-below*
lemmas *dist-less-one = dist-below-one*
lemmas *less-ONE = below-ONE*
lemmas *ONE-less-iff = ONE-below-iff*
lemmas *less-sinlD = below-sinlD*
lemmas *less-sinrD = below-sinrD*

end