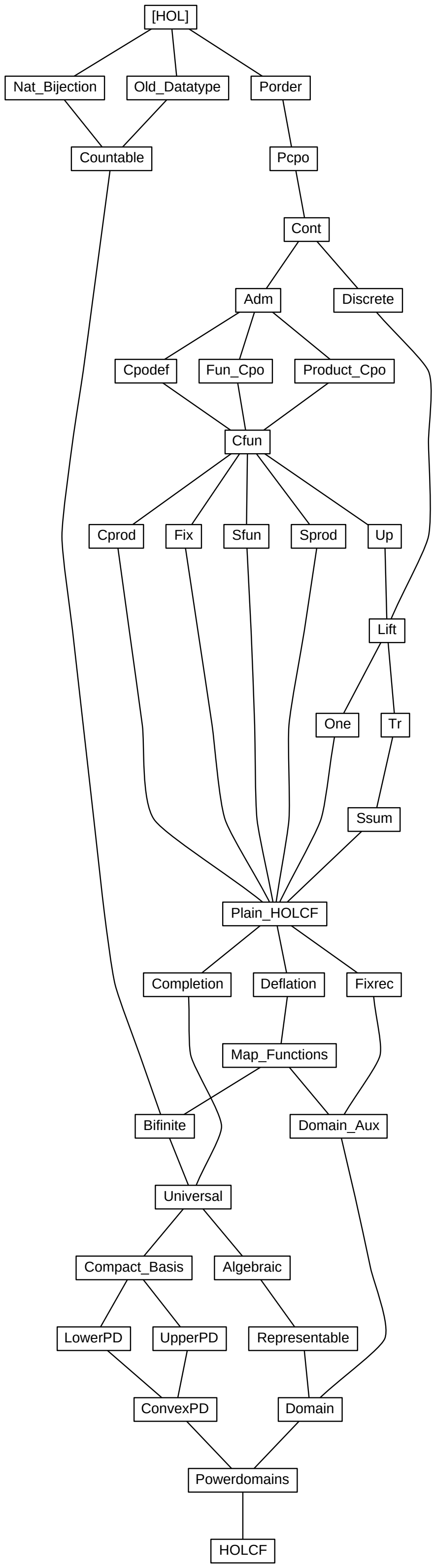
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

April 17, 2016

Contents


[HOL]
Nat_Bijection
Old_Datatype
Porder
Countable
Pcpo
Cont
Adm
Discrete
Cpodef
Fun_Cpo
Product_Cpo
Cfun
Cprod
Fix
Sfun
Sprod
Up
Lift
One
Tr
Ssum
Plain_HOLCF
Completion
Deflation
Fixrec
Map_Functions
Bifinite
Domain_Aux
Universal
Compact_Basis
Algebraic
LowerPD
UpperPD
Representable
ConvexPD
Domain
Powerdomains
HOLCF

1 Partial orders

theory *Porder*
imports *Main*
begin

declare [[*typedef-overloaded*]]

1.1 Type class for partial orders

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

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

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

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

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

lemma *below-eq-trans*: $[\![a \sqsubseteq b;\ b = c]\!] \Longrightarrow a \sqsubseteq c$
  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

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

lemma *is-ubI*: $(\bigwedge x.\ x \in S \Longrightarrow x \sqsubseteq u) \Longrightarrow S <| u$
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* ⇒ *'a* ⇒ *bool* (**infix** <<| *55*) **where**
  S <<| *x* ⟷ *S* <| *x* ∧ (∀ *u*. *S* <| *u* ⟶ *x* ⊑ *u*)

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

end

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

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

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

context *po*
begin

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

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

access to some definition as inference rule

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

lemma *is-lubD2*: ⟦*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*

lemma *is-lub-below-iff*: *S* <<| *x* ⟹ *x* ⊑ *u* ⟷ *S* <| *u*
  unfolding *is-lub-def is-ub-def* **by** (*metis below-trans*)

lubs are unique

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

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

lemma *is-lub-lub*: *M* <<| *x* ⟹ *M* <<| *lub M*
  unfolding *lub-def* **by** (*rule theI* [*OF* - *is-lub-unique*])

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

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

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

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 lub-eqI*])

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 lub-eqI*])

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*: $[\![chain\ Y; i \le j]\!] \Longrightarrow Y\ i \sqsubseteq Y\ j$
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-lub-rangeD1*: $range\ S <<| x \Longrightarrow S\ i \sqsubseteq x$
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*:
$chain\ S \Longrightarrow range\ (\lambda i.\ S\ (i + j))$ <<| $x = range\ S$ <<| x
by (*simp add*: *is-lub-def is-ub-range-shift*)

the lub of a constant chain is the constant

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

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

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

1.5 Finite chains

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

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

results about finite chains

lemma *max-in-chainI*: $(\bigwedge j.\ i \le j \Longrightarrow Y\ i = Y\ j) \Longrightarrow max\text{-}in\text{-}chain\ i\ Y$
unfolding *max-in-chain-def* **by** *fast*

lemma *max-in-chainD*: $[\![max\text{-}in\text{-}chain\ i\ Y;\ i \le j]\!] \Longrightarrow Y\ i = Y\ j$
unfolding *max-in-chain-def* **by** *fast*

lemma *finite-chainI*:
$[\![chain\ C;\ max\text{-}in\text{-}chain\ i\ C]\!] \Longrightarrow finite\text{-}chain\ C$
unfolding *finite-chain-def* **by** *fast*

lemma *finite-chainE*:
$[\![finite\text{-}chain\ C;\ \bigwedge i.\ [\![chain\ C;\ max\text{-}in\text{-}chain\ i\ C]\!] \Longrightarrow R]\!] \Longrightarrow R$
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=λi. 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 ((λx. LEAST k. f k = x) ‘ range f)
  have ?j ≤ ?k
  proof (rule Max-ge)
    show finite ((λx. LEAST k. f k = x) ‘ range f)
      using finite-range by (rule finite-imageI)
    show ?j ∈ (λx. 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*)$]\!] \Longrightarrow$ *finite-chain Y*
 apply (*subgoal-tac* $\exists k. \forall i.\ Y\ i \sqsubseteq Y\ k$)
  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*: $x \sqsubseteq y \Longrightarrow$ *chain* ($\lambda i.$ *if i=0 then x else y*)
 by (*rule chainI*, *simp*)

lemma *bin-chainmax*:
 $x \sqsubseteq y \Longrightarrow$ *max-in-chain* (*Suc 0*) ($\lambda i.$ *if i=0 then x else y*)
 unfolding *max-in-chain-def* **by** *simp*

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

the maximal element in a chain is its lub

lemma *lub-chain-maxelem*: $[\![Y\ i = c; \forall i.\ Y\ i \sqsubseteq c]\!] \Longrightarrow$ *lub* (*range Y*) = *c*
 by (*blast dest*: *ub-rangeD intro*: *lub-eqI is-lubI ub-rangeI*)

end

end

2 Classes cpo and pcpo

theory *Pcpo*
imports *Porder*
begin

2.1 Complete partial orders

The class cpo of chain complete partial orders

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

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

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

lemma *thelubE*: ⟦*chain S*; ($\bigsqcup i.$ *S i*) = *l*⟧ $\Longrightarrow$ *range S* <<| *l*
by (*blast dest*: *cpo intro*: *is-lub-lub*)

Properties of the lub

lemma *is-ub-thelub*: *chain S* $\Longrightarrow$ *S x* $\sqsubseteq$ ($\bigsqcup i.$ *S i*)
by (*blast dest*: *cpo intro*: *is-lub-lub* [*THEN is-lub-rangeD1*])

lemma *is-lub-thelub*:
⟦*chain S*; *range S* <| *x*⟧ $\Longrightarrow$ ($\bigsqcup i.$ *S i*) $\sqsubseteq$ *x*
by (*blast dest*: *cpo intro*: *is-lub-lub* [*THEN is-lubD2*])

lemma *lub-below-iff*: *chain S* $\Longrightarrow$ ($\bigsqcup i.$ *S i*) $\sqsubseteq$ *x* $\longleftrightarrow$ ($\forall i.$ *S i* $\sqsubseteq$ *x*)
by (*simp add*: *is-lub-below-iff* [*OF cpo-lubI*] *is-ub-def*)

lemma *lub-below*: ⟦*chain S*; $\bigwedge i.$ *S i* $\sqsubseteq$ *x*⟧ $\Longrightarrow$ ($\bigsqcup i.$ *S i*) $\sqsubseteq$ *x*
by (*simp add*: *lub-below-iff*)

lemma *below-lub*: ⟦*chain S*; *x* $\sqsubseteq$ *S i*⟧ $\Longrightarrow$ *x* $\sqsubseteq$ ($\bigsqcup i.$ *S i*)
by (*erule below-trans*, *erule is-ub-thelub*)

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 lub-below*)
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 lub-below*)
apply *assumption*
apply (*rule-tac i=i* **in** *below-lub*)
apply (*erule chain-shift*)

apply (*erule chain-mono*)
apply (*rule le-add1*)
done

lemma *maxinch-is-thelub*:
  $chain\ Y \Longrightarrow max\text{-}in\text{-}chain\ i\ Y = ((\bigsqcup i.\ Y\ i) = Y\ i)$
apply (*rule iffI*)
apply (*fast intro*!: *lub-eqI 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*:
  $[\![chain\ X;\ chain\ Y;\ \bigwedge i.\ X\ i \sqsubseteq Y\ i]\!]$
    $\Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
by (*fast elim*: *lub-below below-lub*)

the = relation between two chains is preserved by their lubs

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

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 lub-below [OF 4])
    apply (rule lub-below [OF 2])
    apply (rule below-lub [OF 3])
    apply (rule below-trans)
    apply (rule chain-mono [OF 1 max.cobounded1])
    apply (rule chain-mono [OF 2 max.cobounded2])
    done
  show (⨆ i. Y i i) ⊑ (⨆ i. ⨆ 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: ⋀j. chain (λi. Y i j)
  assumes 2: ⋀i. chain (λj. Y i j)
  shows (⨆ i. ⨆ j. Y i j) = (⨆ j. ⨆ 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: ∃ x. ∀ y. x ⊑ y
begin

definition bottom :: 'a  (⊥)
  where bottom = (THE x. ∀ y. x ⊑ y)

lemma minimal [iff]: ⊥ ⊑ x
unfolding bottom-def
apply (rule the1I2)
apply (rule ex-ex1I)
apply (rule least)
apply (blast intro: below-antisym)
apply simp
done

end
```

Old "UU" syntax:

```
syntax UU :: logic

translations UU => CONST bottom
```

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

```
setup ‹
```

```
  Reorient-Proc.add
    (fn Const(@{const-name bottom}, -) => true | - => false)
›
```

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

useful lemmas about $\bot$

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

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

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

lemma *lub-eq-bottom-iff*: *chain* $Y \Longrightarrow (\bigsqcup i.\ Y\ i) = \bot \longleftrightarrow (\forall i.\ Y\ i = \bot)$
by (*simp only*: *eq-bottom-iff lub-below-iff*)

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

subclass *chfin*
apply *standard*
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 $(x \sqsubseteq y) = (x = \bot \lor x = y)$
 by (*safe dest*!: *ax-flat*)

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

end

2.4 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 *chfin*
proof
 fix *S* :: $nat \Rightarrow 'a$
 assume *S*: *chain S*
 hence $\exists x.\ S = (\lambda i.\ x)$ **by** (*rule discrete-chain-const*)
 hence *max-in-chain 0 S*
  unfolding *max-in-chain-def* **by** *auto*
 thus $\exists i.$ *max-in-chain i S* **..**
qed

end

end

3 Continuity and monotonicity

theory *Cont*
imports *Pcpo*
begin

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

default-sort *po*

3.1 Definitions

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

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

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

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

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

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

3.2 Equivalence of alternate definition

monotone functions map chains to chains

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

monotone functions map upper bound to upper bounds

lemma *ub2ub-monofun*:

$[\![monofun\ f;\ range\ Y <\!|\ u]\!] \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <\!|\ f\ u$
apply (*rule ub-rangeI*)
apply (*erule monofunE*)
apply (*erule ub-rangeD*)
done

a lemma about binary chains

lemma *binchain-cont*:
$[\![cont\ f;\ x \sqsubseteq y]\!] \Longrightarrow range\ (\lambda i{::}nat.\ f\ (if\ i = 0\ then\ x\ else\ y)) <\!<\!|\ f\ y$
apply (*subgoal-tac* $f\ (\bigsqcup i{::}nat.\ if\ i = 0\ then\ x\ else\ y) = f\ y$)
apply (*erule subst*)
apply (*erule contE*)
apply (*erule bin-chain*)
apply (*rule-tac f=f* **in** *arg-cong*)
apply (*erule is-lub-bin-chain* [*THEN lub-eqI*])
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-lub-rangeD1*)
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 lub-eqI* [*symmetric*])
apply (*erule* (*1*) *contE*)
done

lemma *contI2*:
fixes $f :: 'a{::}cpo \Rightarrow 'b{::}cpo$
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*
proof (*rule contI*)
fix $Y :: nat \Rightarrow 'a$
assume *Y*: *chain Y*
with *mono* **have** *fY*: $chain\ (\lambda i.\ f\ (Y\ i))$
by (*rule ch2ch-monofun*)
have $(\bigsqcup i.\ f\ (Y\ i)) = f\ (\bigsqcup i.\ Y\ i)$
apply (*rule below-antisym*)

```
    apply (rule lub-below [OF fY])
    apply (rule monofunE [OF mono])
    apply (rule is-ub-thelub [OF Y])
    apply (rule below [OF Y fY])
    done
  with fY show range (λi. f (Y i)) <<| f (⨆i. Y i)
    by (rule thelubE)
qed
```

3.3 Collection of continuity rules

named-theorems *cont2cont continuity intro rule*

3.4 Continuity of basic functions

The identity function is continuous

```
lemma cont-id [simp, cont2cont]: cont (λx. x)
apply (rule contI)
apply (erule cpo-lubI)
done
```

constant functions are continuous

```
lemma cont-const [simp, cont2cont]: cont (λx. c)
  using is-lub-const by (rule contI)
```

application of functions is continuous

```
lemma cont-apply:
  fixes f :: 'a::cpo ⇒ 'b::cpo ⇒ 'c::cpo and t :: 'a ⇒ 'b
  assumes 1: cont (λx. t x)
  assumes 2: ⋀x. cont (λy. f x y)
  assumes 3: ⋀y. cont (λx. f x y)
  shows cont (λx. (f x) (t x))
proof (rule contI2 [OF monofunI])
  fix x y :: 'a assume x ⊑ y
  then show f x (t x) ⊑ f y (t y)
    by (auto intro: cont2monofunE [OF 1]
                    cont2monofunE [OF 2]
                    cont2monofunE [OF 3]
                    below-trans)
next
  fix Y :: nat ⇒ 'a assume chain Y
  then show f (⨆i. Y i) (t (⨆i. Y i)) ⊑ (⨆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*])

Least upper bounds preserve continuity

lemma *cont2cont-lub* [*simp*]:
assumes *chain*: $\bigwedge x.$ *chain* ($\lambda i.$ *F i x*) **and** *cont*: $\bigwedge i.$ *cont* ($\lambda x.$ *F i x*)
shows *cont* ($\lambda x. \bigsqcup i.$ *F i x*)
apply (*rule contI2*)
apply (*simp add*: *monofunI cont2monofunE* [*OF cont*] *lub-mono chain*)
apply (*simp add*: *cont2contlubE* [*OF cont*])
apply (*simp add*: *diag-lub ch2ch-cont* [*OF cont*] *chain*)
done

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*$]\!] \Longrightarrow$ *finite-chain* ($\lambda n.$ *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*$]\!] \Longrightarrow$ *finite-chain* ($\lambda n.$ *f* (*Y n*))
by (*rule cont2mono* [*THEN monofun-finch2finch*])

All monotone functions with chain-finite domain are continuous.

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

All strict functions with flat domain are continuous.

lemma *flatdom-strict2mono*: *f* $\bot = \bot \Longrightarrow$ *monofun* (*f*::'*a*::*flat* $\Rightarrow$ '*b*::*pcpo*)
apply (*rule monofunI*)

apply (*drule ax-flat*)
apply *auto*
done

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

All functions with discrete domain are continuous.

lemma *cont-discrete-cpo* [*simp*, *cont2cont*]: $cont\ (f{::}'a{::}discrete\text{-}cpo \Rightarrow 'b{::}cpo)$
apply (*rule contI*)
apply (*drule discrete-chain-const*, *clarify*)
apply (*simp add*: *is-lub-const*)
done

end

4 Admissibility and compactness

theory *Adm*
imports *Cont*
begin

default-sort *cpo*

4.1 Definitions

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

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

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

lemma *admD2*: $[\![adm\ (\lambda x.\ \neg\ P\ x);\ chain\ Y;\ P\ (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists i.\ P\ (Y\ i)$
unfolding *adm-def* **by** *fast*

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

4.2 Admissibility on chain-finite types

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

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

4.3 Admissibility of special formulae and propagation

lemma *adm-const* [*simp*]: *adm* (λ*x*. *t*)
by (*rule admI*, *simp*)

lemma *adm-conj* [*simp*]:
  ⟦*adm* (λ*x*. *P x*); *adm* (λ*x*. *Q x*)⟧ ⟹ *adm* (λ*x*. *P x* ∧ *Q x*)
by (*fast intro*: *admI elim*: *admD*)

lemma *adm-all* [*simp*]:
  (⋀*y*. *adm* (λ*x*. *P x y*)) ⟹ *adm* (λ*x*. ∀ *y*. *P x y*)
by (*fast intro*: *admI elim*: *admD*)

lemma *adm-ball* [*simp*]:
  (⋀*y*. *y* ∈ *A* ⟹ *adm* (λ*x*. *P x y*)) ⟹ *adm* (λ*x*. ∀ *y*∈*A*. *P x y*)
by (*fast intro*: *admI elim*: *admD*)

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

lemma *adm-disj-lemma1*:
  assumes *adm*: *adm P*
  assumes *chain*: *chain Y*
  assumes *P*: ∀ *i*. ∃ *j*≥*i*. *P* (*Y j*)
  shows *P* (⨆ *i*. *Y i*)
proof −
  def *f* ≡ λ*i*. *LEAST j*. *i* ≤ *j* ∧ *P* (*Y j*)
  have *chain'*: *chain* (λ*i*. *Y* (*f i*))
    unfolding *f-def*
    apply (*rule chainI*)
    apply (*rule chain-mono* [*OF chain*])
    apply (*rule Least-le*)
    apply (*rule LeastI2-ex*)
    apply (*simp-all add*: *P*)
    done
  have *f1*: ⋀*i*. *i* ≤ *f i* **and** *f2*: ⋀*i*. *P* (*Y* (*f i*))
    using *LeastI-ex* [*OF P* [*rule-format*]] **by** (*simp-all add*: *f-def*)
  have *lub-eq*: (⨆ *i*. *Y i*) = (⨆ *i*. *Y* (*f i*))
    apply (*rule below-antisym*)
    apply (*rule lub-mono* [*OF chain chain'*])
    apply (*rule chain-mono* [*OF chain f1*])
    apply (*rule lub-range-mono* [*OF - chain chain'*])
    apply *clarsimp*
    done
  show *P* (⨆ *i*. *Y i*)
    unfolding *lub-eq* **using** *adm chain' f2* **by** (*rule admD*)
qed

lemma *adm-disj-lemma2*:
  ∀ *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* [*simp*]:
$[\![adm\ (\lambda x.\ P\ x);\ adm\ (\lambda x.\ Q\ x)]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \vee Q\ x)$
apply (*rule admI*)
apply (*erule adm-disj-lemma2* [*THEN disjE*])
apply (*erule* (*2*) *adm-disj-lemma1* [*THEN disjI1*])
apply (*erule* (*2*) *adm-disj-lemma1* [*THEN disjI2*])
done

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

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

admissibility and continuity

lemma *adm-below* [*simp*]:
$[\![cont\ (\lambda x.\ u\ x);\ cont\ (\lambda x.\ v\ x)]\!] \Longrightarrow adm\ (\lambda x.\ u\ x \sqsubseteq v\ x)$
by (*simp add*: *adm-def cont2contlubE lub-mono ch2ch-cont*)

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

lemma *adm-subst*: $[\![cont\ (\lambda x.\ t\ x);\ adm\ P]\!] \Longrightarrow adm\ (\lambda x.\ P\ (t\ x))$
by (*simp add*: *adm-def cont2contlubE ch2ch-cont*)

lemma *adm-not-below* [*simp*]: $cont\ (\lambda x.\ t\ x) \Longrightarrow adm\ (\lambda x.\ t\ x \not\sqsubseteq u)$
by (*rule admI*, *simp add*: *cont2contlubE ch2ch-cont lub-below-iff*)

4.4 Compactness

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

lemma *compactI*: $adm\ (\lambda x.\ k \not\sqsubseteq x) \Longrightarrow compact\ k$
unfolding *compact-def* **.**

lemma *compactD*: $compact\ k \Longrightarrow adm\ (\lambda x.\ k \not\sqsubseteq x)$
unfolding *compact-def* **.**

lemma *compactI2*:

$(\bigwedge Y.\ [\![chain\ Y;\ x \sqsubseteq (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists\, i.\ x \sqsubseteq Y\ i) \Longrightarrow compact\ x$
unfolding *compact-def adm-def* **by** *fast*

lemma *compactD2*:
$[\![compact\ x;\ chain\ Y;\ x \sqsubseteq (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists\, i.\ x \sqsubseteq Y\ i$
unfolding *compact-def adm-def* **by** *fast*

lemma *compact-below-lub-iff*:
$[\![compact\ x;\ chain\ Y]\!] \Longrightarrow x \sqsubseteq (\bigsqcup i.\ Y\ i) \longleftrightarrow (\exists\, i.\ x \sqsubseteq Y\ i)$
by (*fast intro*: *compactD2 elim*: *below-lub*)

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

lemma *compact-imp-max-in-chain*:
$[\![chain\ Y;\ compact\ (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists\, i.\ max\text{-}in\text{-}chain\ i\ Y$
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* [*simp*]:
$[\![compact\ k;\ cont\ (\lambda x.\ t\ x)]\!] \Longrightarrow adm\ (\lambda x.\ k \not\sqsubseteq t\ x)$
unfolding *compact-def* **by** (*rule adm-subst*)

lemma *adm-neq-compact* [*simp*]:
$[\![compact\ k;\ cont\ (\lambda x.\ t\ x)]\!] \Longrightarrow adm\ (\lambda x.\ t\ x \neq k)$
by (*simp add*: *po-eq-conv*)

lemma *adm-compact-neq* [*simp*]:
$[\![compact\ k;\ cont\ (\lambda x.\ t\ x)]\!] \Longrightarrow adm\ (\lambda x.\ k \neq t\ x)$
by (*simp add*: *po-eq-conv*)

lemma *compact-bottom* [*simp*, *intro*]: *compact* ⊥
by (*rule compactI*, *simp*)

Any upward-closed predicate is admissible.

lemma *adm-upward*:
assumes *P*: $\bigwedge x\ y.\ [\![P\ x;\ x \sqsubseteq y]\!] \Longrightarrow P\ y$
shows *adm P*
by (*rule admI*, *drule spec*, *erule P*, *erule is-ub-thelub*)

lemmas *adm-lemmas* =
adm-const adm-conj adm-all adm-ball adm-disj adm-imp adm-iff
adm-below adm-eq adm-not-below

adm-compact-not-below adm-compact-neq adm-neq-compact

end

5 Subtypes of pcpos

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

5.1 Proving a subtype is a partial order

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

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* ⊑ ≡ λ*x y. Rep x* ⊑ *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*}})›

5.2 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

5.3 Proving a subtype is chain-finite

lemma *ch2ch-Rep*:

assumes *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows $chain\ S \Longrightarrow chain\ (\lambda i.\ Rep\ (S\ i))$
unfolding *chain-def below* **.**

theorem *typedef-chfin*:
fixes $Abs :: 'a::chfin \Rightarrow 'b::po$
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

5.4 Proving a subtype is complete

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

lemma *typedef-is-lubI*:
assumes *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
shows $range\ (\lambda i.\ Rep\ (S\ i)) <<|\ Rep\ x \Longrightarrow range\ S <<|\ x$
unfolding *is-lub-def is-ub-def below* **by** *simp*

lemma *Abs-inverse-lub-Rep*:
fixes $Abs :: 'a::cpo \Rightarrow 'b::po$
assumes *type*: *type-definition Rep Abs A*
and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
shows $chain\ S \Longrightarrow Rep\ (Abs\ (\bigsqcup i.\ Rep\ (S\ i))) = (\bigsqcup i.\ Rep\ (S\ i))$
apply (*rule type-definition.Abs-inverse* [*OF type*])
apply (*erule admD* [*OF adm ch2ch-Rep* [*OF below*]])
apply (*rule type-definition.Rep* [*OF type*])
done

theorem *typedef-is-lub*:
fixes $Abs :: 'a::cpo \Rightarrow 'b::po$
assumes *type*: *type-definition Rep Abs A*
and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
shows $chain\ S \Longrightarrow range\ S <<|\ Abs\ (\bigsqcup i.\ Rep\ (S\ i))$
proof −
assume *S*: *chain S*
hence $chain\ (\lambda i.\ Rep\ (S\ i))$ **by** (*rule ch2ch-Rep* [*OF below*])
hence $range\ (\lambda i.\ Rep\ (S\ i)) <<|\ (\bigsqcup i.\ Rep\ (S\ i))$ **by** (*rule cpo-lubI*)
hence $range\ (\lambda i.\ Rep\ (S\ i)) <<|\ Rep\ (Abs\ (\bigsqcup i.\ Rep\ (S\ i)))$
by (*simp only*: *Abs-inverse-lub-Rep* [*OF type below adm S*])

thus *range S <<| Abs* (⨆ *i. Rep* (*S i*))
  by (*rule typedef-is-lubI* [*OF below*])
qed

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

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* (⨆ *i. Rep* (*S i*))
    by (*rule typedef-is-lub* [*OF type below adm*])
  thus ∃ *x. range S <<| x* **..**
qed

5.4.1 Continuity of *Rep* and *Abs*

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

theorem *typedef-cont-Rep*:
  fixes *Abs* :: *'a::cpo* ⇒ *'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 *cont* (λ*x. f x*) ⟹ *cont* (λ*x. Rep* (*f x*))
 apply (*erule cont-apply* [*OF - - cont-const*])
 apply (*rule contI*)
 apply (*simp only*: *typedef-lub* [*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-cont-Abs*:
  fixes *Abs* :: *'a::cpo* ⇒ *'b::cpo*
  fixes *f* :: *'c::cpo* ⇒ *'a::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*)
    and *f-in-A*: ⋀*x. f x* ∈ *A*
  shows *cont f* ⟹ *cont* (λ*x. Abs* (*f x*))
unfolding *cont-def is-lub-def is-ub-def ball-simps below*
by (*simp add*: *type-definition.Abs-inverse* [*OF type f-in-A*])

5.5 Proving subtype elements are compact

theorem *typedef-compact*:
  fixes *Abs* :: *'a::cpo* ⇒ *'b::cpo*
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *adm*: $adm\ (\lambda x.\ x \in A)$
  shows $compact\ (Rep\ k) \Longrightarrow compact\ k$
proof (*unfold compact-def*)
  have *cont-Rep*: *cont Rep*
    by (*rule typedef-cont-Rep* [*OF type below adm cont-id*])
  assume $adm\ (\lambda x.\ Rep\ k \not\sqsubseteq x)$
  with *cont-Rep* **have** $adm\ (\lambda x.\ Rep\ k \not\sqsubseteq Rep\ x)$ **by** (*rule adm-subst*)
  thus $adm\ (\lambda x.\ k \not\sqsubseteq x)$ **by** (*unfold below*)
qed

5.6 Proving a subtype is pointed

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

theorem *typedef-pcpo-generic*:
  fixes *Abs* :: *'a::cpo* ⇒ *'b::cpo*
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *z-in-A*: $z \in A$
    and *z-least*: $\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x$
  shows *OFCLASS*(*'b*, *pcpo-class*)
 apply (*intro-classes*)
 apply (*rule-tac x=Abs z* **in** *exI*, *rule allI*)
 apply (*unfold 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*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *bottom-in-A*: $\bot \in A$
  shows *OFCLASS*(*'b*, *pcpo-class*)
by (*rule typedef-pcpo-generic* [*OF type below bottom-in-A*], *rule minimal*)

5.6.1 Strictness of *Rep* and *Abs*

For a sub-pcpo where ⊥ 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 *bottom-in-A*: $\bot \in A$
  shows $Abs\ \bot = \bot$
 apply (*rule bottomI*, *unfold below*)
 apply (*simp add*: *type-definition.Abs-inverse* [*OF type bottom-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 *bottom-in-A*: $\bot \in A$
  shows $Rep\ \bot = \bot$
 apply (*rule typedef-Abs-strict* [*OF type below bottom-in-A*, *THEN subst*])
 apply (*rule type-definition.Abs-inverse* [*OF type bottom-in-A*])
done

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

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

5.7 Proving a subtype is flat

theorem *typedef-flat*:
  fixes *Abs* :: *'a::flat* ⇒ *'b::pcpo*
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *bottom-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 bottom-in-A*])
 apply (*simp add*: *ax-flat*)

done

5.8 HOLCF type definition package

ML-file *Tools/cpodef.ML*

end

6 Class instances for the full function space

theory *Fun-Cpo*
imports *Adm*
begin

6.1 Full function space is a partial order

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

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

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 fun-eq-iff 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

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

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

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

6.2 Full function space is chain complete

Properties of chains of functions.

lemma *fun-chain-iff*: $chain\ S \longleftrightarrow (\forall x.\ chain\ (\lambda i.\ S\ i\ x))$
unfolding *chain-def fun-below-iff* **by** *auto*

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*)

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

lemma *is-lub-lambda*:
  $(\bigwedge x.\ range\ (\lambda i.\ Y\ i\ x) <<|\ f\ x) \Longrightarrow range\ Y <<|\ f$
unfolding *is-lub-def is-ub-def below-fun-def* **by** *simp*

lemma *is-lub-fun*:
  $chain\ (S{::}nat \Rightarrow\ 'a{::}type \Rightarrow\ 'b{::}cpo)$
    $\Longrightarrow range\ S <<|\ (\lambda x.\ \bigsqcup i.\ S\ i\ x)$
apply (*rule is-lub-lambda*)
apply (*rule cpo-lubI*)
apply (*erule ch2ch-fun*)
done

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

instance *fun* :: (*type*, *cpo*) *cpo*
by *intro-classes* (*rule exI*, *erule is-lub-fun*)

instance *fun* :: (*type*, *discrete-cpo*) *discrete-cpo*
proof
  fix $f\ g :: 'a \Rightarrow\ 'b$
  show $f \sqsubseteq g \longleftrightarrow f = g$
    unfolding *fun-below-iff fun-eq-iff*
    by *simp*
qed

6.3 Full function space is pointed

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

instance *fun* :: (*type*, *pcpo*) *pcpo*
by *standard* (*fast intro*: *minimal-fun*)

lemma *inst-fun-pcpo*: $\bot = (\lambda x.\ \bot)$
by (*rule minimal-fun* [*THEN bottomI*, *symmetric*])

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

lemma *lambda-strict*: $(\lambda x.\ \bot) = \bot$
by (*rule bottomI*, *rule minimal-fun*)

6.4 Propagation of monotonicity and continuity

The lub of a chain of monotone functions is monotone.

lemma *adm-monofun*: *adm monofun*
by (*rule admI*, *simp add*: *lub-fun fun-chain-iff monofun-def lub-mono*)

The lub of a chain of continuous functions is continuous.

lemma *adm-cont*: *adm cont*
by (*rule admI*, *simp add*: *lub-fun fun-chain-iff*)

Function application preserves monotonicity and continuity.

lemma *mono2mono-fun*: *monofun f* $\Longrightarrow$ *monofun* $(\lambda x.\ f\ x\ y)$
by (*simp add*: *monofun-def fun-below-iff*)

lemma *cont2cont-fun*: *cont f* $\Longrightarrow$ *cont* $(\lambda x.\ f\ x\ y)$
apply (*rule contI2*)
apply (*erule cont2mono* [*THEN mono2mono-fun*])
apply (*simp add*: *cont2contlubE lub-fun ch2ch-cont*)
done

lemma *cont-fun*: *cont* $(\lambda f.\ f\ x)$
using *cont-id* **by** (*rule cont2cont-fun*)

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

lemma *mono2mono-lambda*:
  assumes *f*: $\bigwedge y.$ *monofun* $(\lambda x.\ f\ x\ y)$ **shows** *monofun f*
using *f* **by** (*simp add*: *monofun-def fun-below-iff*)

lemma *cont2cont-lambda* [*simp*]:
  assumes *f*: $\bigwedge y.$ *cont* $(\lambda x.\ f\ x\ y)$ **shows** *cont f*
by (*rule contI*, *rule is-lub-lambda*, *rule contE* [*OF f*])

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*: *lub-fun ch2ch-lambda*)

end

7 The cpo of cartesian products

theory *Product-Cpo*
imports *Adm*
begin

default-sort *cpo*

7.1 Unit type is a pcpo

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

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

instance proof
qed *simp*

end

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

7.2 Product type is a partial order

instantiation *prod* :: (*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 *prod* :: (*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 prod-eq-iff*
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

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

lemma *prod-belowI*: ⟦*fst p* ⊑ *fst q*; *snd p* ⊑ *snd q*⟧ ⟹ *p* ⊑ *q*
unfolding *below-prod-def* **by** *simp*

lemma *Pair-below-iff* [*simp*]: (*a*, *b*) ⊑ (*c*, *d*) ⟷ *a* ⊑ *c* ∧ *b* ⊑ *d*
unfolding *below-prod-def* **by** *simp*

Pair (-,-) is monotone in both arguments

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

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

lemma *monofun-pair*:
⟦*x1* ⊑ *x2*; *y1* ⊑ *y2*⟧ ⟹ (*x1*, *y1*) ⊑ (*x2*, *y2*)
by *simp*

lemma *ch2ch-Pair* [*simp*]:
chain X ⟹ *chain Y* ⟹ *chain* (λ*i*. (*X i*, *Y i*))
by (*rule chainI*, *simp add*: *chainE*)

fst and *snd* are monotone

lemma *fst-monofun*: *x* ⊑ *y* ⟹ *fst x* ⊑ *fst y*
unfolding *below-prod-def* **by** *simp*

lemma *snd-monofun*: *x* ⊑ *y* ⟹ *snd x* ⊑ *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* = (λ*i*. (*A i*, *B i*))
proof
from ‹*chain Y*› **show** *chain* (λ*i*. *fst* (*Y i*)) **by** (*rule ch2ch-fst*)
from ‹*chain Y*› **show** *chain* (λ*i*. *snd* (*Y i*)) **by** (*rule ch2ch-snd*)

show $Y = (\lambda i.\ (fst\ (Y\ i),\ snd\ (Y\ i)))$ **by** *simp*
qed

7.4 Product type is a cpo

lemma *is-lub-Pair*:
$[\![range\ A <<|\ x;\ range\ B <<|\ y]\!] \Longrightarrow range\ (\lambda i.\ (A\ i,\ B\ i)) <<|\ (x,\ y)$
unfolding *is-lub-def is-ub-def ball-simps below-prod-def* **by** *simp*

lemma *lub-Pair*:
$[\![chain\ (A::nat \Rightarrow 'a::cpo);\ chain\ (B::nat \Rightarrow 'b::cpo)]\!]$
$\Longrightarrow (\bigsqcup i.\ (A\ i,\ B\ i)) = (\bigsqcup i.\ A\ i,\ \bigsqcup i.\ B\ i)$
by (*fast intro*: *lub-eqI is-lub-Pair elim*: *thelubE*)

lemma *is-lub-prod*:
fixes $S :: nat \Rightarrow ('a::cpo \times 'b::cpo)$
assumes S: *chain S*
shows $range\ S <<|\ (\bigsqcup i.\ fst\ (S\ i),\ \bigsqcup i.\ snd\ (S\ i))$
using S **by** (*auto elim*: *prod-chain-cases simp add*: *is-lub-Pair cpo-lubI*)

lemma *lub-prod*:
$chain\ (S::nat \Rightarrow 'a::cpo \times 'b::cpo)$
$\Longrightarrow (\bigsqcup i.\ S\ i) = (\bigsqcup i.\ fst\ (S\ i),\ \bigsqcup i.\ snd\ (S\ i))$
by (*rule is-lub-prod* [*THEN lub-eqI*])

instance *prod* :: (*cpo*, *cpo*) *cpo*
proof
fix $S :: nat \Rightarrow ('a \times 'b)$
assume *chain S*
hence $range\ S <<|\ (\bigsqcup i.\ fst\ (S\ i),\ \bigsqcup i.\ snd\ (S\ i))$
by (*rule is-lub-prod*)
thus $\exists x.\ range\ S <<|\ x$ **..**
qed

instance *prod* :: (*discrete-cpo*, *discrete-cpo*) *discrete-cpo*
proof
fix $x\ y :: 'a \times 'b$
show $x \sqsubseteq y \longleftrightarrow x = y$
unfolding *below-prod-def prod-eq-iff*
by *simp*
qed

7.5 Product type is pointed

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

instance *prod* :: (*pcpo*, *pcpo*) *pcpo*
by *intro-classes* (*fast intro*: *minimal-prod*)

lemma *inst-prod-pcpo*: ⊥ = (⊥, ⊥)
by (*rule minimal-prod* [*THEN bottomI*, *symmetric*])

lemma *Pair-bottom-iff* [*simp*]: (*x*, *y*) = ⊥ ⟷ *x* = ⊥ ∧ *y* = ⊥
unfolding *inst-prod-pcpo* **by** *simp*

lemma *fst-strict* [*simp*]: *fst* ⊥ = ⊥
unfolding *inst-prod-pcpo* **by** (*rule fst-conv*)

lemma *snd-strict* [*simp*]: *snd* ⊥ = ⊥
unfolding *inst-prod-pcpo* **by** (*rule snd-conv*)

lemma *Pair-strict* [*simp*]: (⊥, ⊥) = ⊥
by *simp*

lemma *split-strict* [*simp*]: *case-prod f* ⊥ = *f* ⊥ ⊥
unfolding *split-def* **by** *simp*

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

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

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

lemma *cont-fst*: *cont fst*
apply (*rule contI*)
apply (*simp add*: *lub-prod*)
apply (*erule cpo-lubI* [*OF ch2ch-fst*])
done

lemma *cont-snd*: *cont snd*
apply (*rule contI*)
apply (*simp add*: *lub-prod*)
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-case-prod*:
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*. *case g x of* (*a*, *b*) ⇒ *f x a b*)
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 *prod-contI*:
assumes *f1*: ⋀*y*. *cont* (λ*x*. *f* (*x*, *y*))
assumes *f2*: ⋀*x*. *cont* (λ*y*. *f* (*x*, *y*))
shows *cont f*
proof −
have *cont* (λ(*x*, *y*). *f* (*x*, *y*))
by (*intro cont2cont-case-prod f1 f2 cont2cont*)
thus *cont f*
by (*simp only*: *case-prod-eta*)
qed

lemma *prod-cont-iff*:
cont f ⟷ (∀ *y*. *cont* (λ*x*. *f* (*x*, *y*))) ∧ (∀ *x*. *cont* (λ*y*. *f* (*x*, *y*)))
apply *safe*
apply (*erule cont-compose* [*OF - cont-pair1*])
apply (*erule cont-compose* [*OF - cont-pair2*])
apply (*simp only*: *prod-contI*)
done

lemma *cont2cont-case-prod'* [*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* (λ*x*. *case-prod* (*f x*) (*g x*))
using *assms* **by** (*simp add*: *cont2cont-case-prod prod-cont-iff*)

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

lemma *cont2cont-split-simple* [*simp*, *cont2cont*]:
 assumes $\bigwedge a\ b.\ cont\ (\lambda x.\ f\ x\ a\ b)$
 shows $cont\ (\lambda x.\ case\ p\ of\ (a,\ b) \Rightarrow f\ x\ a\ b)$
using *assms* **by** (*cases p*) *auto*

Admissibility of predicates on product types.

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

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

end

8 The type of continuous functions

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

default-sort *cpo*

8.1 Definition of continuous function type

definition *cfun* = {*f*::′*a* => ′*b*. *cont f*}

cpodef (′*a*, ′*b*) *cfun* ((- →/ -) [*1*, *0*] *0*) = *cfun* :: (′*a* => ′*b*) *set*
unfolding *cfun-def* **by** (*auto intro*: *cont-const adm-cont*)

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

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

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

8.2 Syntax for continuous lambda abstraction

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

parse-translation ‹
(∗ *rewrite* (*-cabs x t*) => (*Abs-cfun* (%*x*. *t*)) ∗)
[*Syntax-Trans.mk-binder-tr* (@{*syntax-const -cabs*}, @{*const-syntax Abs-cfun*})];
›

print-translation ‹
[(@{*const-syntax Abs-cfun*}, *fn* - => *fn* [*Abs abs*] =>
let val (*x*, *t*) = *Syntax-Trans.atomic-abs-tr*′ *abs*
in Syntax.const @{*syntax-const -cabs*} $ *x* $ *t end*)]
› — To avoid eta-contraction of body

Syntax for nested abstractions

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

syntax
-Lambda :: [*cargs*, *logic*] ⇒ *logic* ((*3*Λ -./ -) [*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*] =
Ast.fold-ast-p @{*syntax-const -cabs*}
(*Ast.unfold-ast* @{*syntax-const -cargs*} (*Ast.strip-positions pats*), *body*)
| *Lambda-ast-tr asts* = *raise Ast.AST* (*Lambda-ast-tr*, *asts*);

```
  in [(@{syntax-const -Lambda}, K 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 Ast.unfold-ast-p @{syntax-const -cabs}
          (Ast.Appl (Ast.Constant @{syntax-const -cabs} :: asts)) of
        ([], -) => raise Ast.AST (cabs-ast-tr', asts)
      | (xs, body) => Ast.Appl
          [Ast.Constant @{syntax-const -Lambda},
           Ast.fold-ast @{syntax-const -cargs} xs, body]);
  in [(@{syntax-const -cabs}, K cabs-ast-tr')] end
›
```

Dummy patterns for continuous abstraction

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

8.3 Continuous function space is pointed

lemma *bottom-cfun*: ⊥ ∈ *cfun*
by (*simp add*: *cfun-def inst-fun-pcpo*)

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 bottom-cfun*])

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

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

function application is strict in its first argument

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

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

for compatibility with old HOLCF-Version

lemma *inst-cfun-pcpo*: ⊥ = (Λ *x*. ⊥)
by *simp*

8.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

Beta-reduction simproc

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

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

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

```
simproc-setup beta-cfun-proc (Rep-cfun (Abs-cfun f)) = ‹
  fn phi => fn ctxt => fn ct =>
    let
      val dest = Thm.dest-comb;
      val f = (snd o dest o snd o dest) ct;
      val [T, U] = Thm.dest-ctyp (Thm.ctyp-of-cterm f);
      val tr = Thm.instantiate' [SOME T, SOME U] [SOME f]
          (mk-meta-eq @{thm Abs-cfun-inverse2});
      val rules = Named-Theorems.get ctxt @{named-theorems cont2cont};
      val tac = SOLVED' (REPEAT-ALL-NEW (match-tac ctxt (rev rules)));
    in SOME (perhaps (SINGLE (tac 1)) tr) end
›
```

Eta-equality for continuous functions

lemma *eta-cfun*: (Λ *x*. *f*·*x*) = *f*
by (*rule Rep-cfun-inverse*)

Extensionality for continuous functions

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

lemma *cfun-eqI*: ($\bigwedge$*x*. *f*·*x* = *g*·*x*) $\Longrightarrow$ *f* = *g*
by (*simp add*: *cfun-eq-iff*)

Extensionality wrt. ordering for continuous functions

lemma *cfun-below-iff*: *f* $\sqsubseteq$ *g* $\longleftrightarrow$ ($\forall$ *x*. *f*·*x* $\sqsubseteq$ *g*·*x*)
by (*simp add*: *below-cfun-def fun-below-iff*)

lemma *cfun-belowI*: $(\bigwedge x.\ f\cdot x \sqsubseteq g\cdot x) \Longrightarrow f \sqsubseteq g$
by (*simp add*: *cfun-below-iff*)

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*

8.5 Continuity of application

lemma *cont-Rep-cfun1*: *cont* $(\lambda f.\ f\cdot x)$
by (*rule cont-Rep-cfun* [*OF cont-id*, *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*]
lemmas *monofun-Rep-cfun2* = *cont-Rep-cfun2* [*THEN cont2mono*]

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

lemma *contlub-cfun-arg*: *chain* $Y \Longrightarrow f\cdot(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\cdot(Y\ i))$
by (*rule cont-Rep-cfun2* [*THEN cont2contlubE*])

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*])

monotonicity of application

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

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 $'a \rightarrow\ 'b$

lemma *chain-monofun*: *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ f\cdot(Y\ i))$
by (*erule monofun-Rep-cfun2* [*THEN ch2ch-monofun*])

lemma *ch2ch-Rep-cfunR*: $chain\ Y \Longrightarrow chain\ (\lambda i.\ f\cdot(Y\ i))$
by (*rule monofun-Rep-cfun2* [*THEN ch2ch-monofun*])

lemma *ch2ch-Rep-cfunL*: $chain\ F \Longrightarrow chain\ (\lambda i.\ (F\ i)\cdot x)$
by (*rule monofun-Rep-cfun1* [*THEN ch2ch-monofun*])

lemma *ch2ch-Rep-cfun* [*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 cfun-below-iff*)

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

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

lemma *lub-LAM*:
$[\![\bigwedge x.\ chain\ (\lambda i.\ F\ i\ x);\ \bigwedge i.\ cont\ (\lambda x.\ F\ i\ x)]\!]$
$\Longrightarrow (\bigsqcup i.\ \Lambda\ x.\ F\ i\ x) = (\Lambda\ x.\ \bigsqcup i.\ F\ i\ x)$
by (*simp add*: *lub-cfun lub-fun ch2ch-lambda*)

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

strictness

lemma *strictI*: $f\cdot x = \bot \Longrightarrow f\cdot\bot = \bot$
apply (*rule bottomI*)
apply (*erule subst*)
apply (*rule minimal* [*THEN monofun-cfun-arg*])
done

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

lemma *lub-cfun*: $chain\ F \Longrightarrow (\bigsqcup i.\ F\ i) = (\Lambda\ x.\ \bigsqcup i.\ F\ i\cdot x)$
by (*simp add*: *lub-cfun lub-fun ch2ch-lambda*)

8.6 Continuity simplification procedure

cont2cont lemma for *Rep-cfun*

lemma *cont2cont-APP* [*simp*, *cont2cont*]:
assumes *f*: $cont\ (\lambda x.\ f\ x)$
assumes *t*: $cont\ (\lambda x.\ t\ x)$
shows $cont\ (\lambda x.\ (f\ x)\cdot(t\ x))$
proof −
have *1*: $\bigwedge y.\ cont\ (\lambda x.\ (f\ x)\cdot y)$
using *cont-Rep-cfun1 f* **by** (*rule cont-compose*)

show *cont* $(\lambda x.\ (f\ x)\cdot(t\ x))$
using *t cont-Rep-cfun2 1* **by** (*rule cont-apply*)
qed

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

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

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

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 cfun-below-iff* **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 $cont\ (\lambda x.\ \Lambda\ y.\ f\ x\ y)$
using *assms* **by** (*simp add*: *cont2cont-LAM prod-cont-iff*)

lemma *cont2cont-LAM-discrete* [*simp*, *cont2cont*]:
$(\bigwedge y::'a::discrete\text{-}cpo.\ cont\ (\lambda x.\ f\ x\ y)) \Longrightarrow cont\ (\lambda x.\ \Lambda\ y.\ f\ x\ y)$
by (*simp add*: *cont2cont-LAM*)

8.7 Miscellaneous

Monotonicity of *Abs-cfun*

lemma *monofun-LAM*:
$[\![cont\ f;\ cont\ g;\ \bigwedge x.\ f\ x \sqsubseteq g\ x]\!] \Longrightarrow (\Lambda\ x.\ f\ x) \sqsubseteq (\Lambda\ x.\ g\ x)$

by (*simp add*: *cfun-below-iff*)

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*!: *lub-eqI 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*)

8.8 Continuous injection-retraction pairs

Continuous retractions are strict.

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

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

lemma *injection-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*)

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*:
$f\cdot x = (c::'b::flat) \Longrightarrow f\cdot\bot = \bot \vee (\forall z.\ f\cdot z = c)$
apply (*case-tac* $f\cdot x = \bot$)
apply (*rule disjI1*)
apply (*rule bottomI*)
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

8.9 Identity and composition

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

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

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

lemma *ID1* [*simp*]: $ID\cdot x = x$
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*: *cfun-eq-iff*)

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 cfun-eqI*, *simp*)

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

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

8.10 Strictified functions

default-sort *pcpo*

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

lemma *cont2cont-if-bottom* [*cont2cont*, *simp*]:
  assumes *f*: *cont* (λx. *f x*) **and** *g*: *cont* (λx. *g x*)
  shows *cont* (λx. *if f x* = $\bot$ *then* $\bot$ *else g x*)
proof (*rule cont-apply* [*OF f*])
  show $\bigwedge x$. *cont* (λy. *if y* = $\bot$ *then* $\bot$ *else g x*)
    unfolding *cont-def is-lub-def is-ub-def ball-simps*
    by (*simp add*: *lub-eq-bottom-iff*)
  show $\bigwedge y$. *cont* (λx. *if y* = $\bot$ *then* $\bot$ *else g x*)
    by (*simp add*: *g*)
qed

lemma *seq-conv-if*: *seq*·*x* = (*if x* = $\bot$ *then* $\bot$ *else ID*)
unfolding *seq-def* **by** *simp*

lemma *seq-simps* [*simp*]:
  seq·$\bot$ = $\bot$
  seq·*x*·$\bot$ = $\bot$
  x $\neq \bot \Longrightarrow$ *seq*·*x* = *ID*
by (*simp-all add*: *seq-conv-if*)

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

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

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

lemma *strictify2* [*simp*]: $x \neq \bot \Longrightarrow \mathit{strictify}\cdot f\cdot x = f\cdot x$
by (*simp add*: *strictify-conv-if*)

8.11 Continuity of let-bindings

lemma *cont2cont-Let*:
  assumes *f*: *cont* ($\lambda x.\ f\ x$)
  assumes *g1*: $\bigwedge y.$ *cont* ($\lambda x.\ g\ x\ y$)
  assumes *g2*: $\bigwedge x.$ *cont* ($\lambda y.\ g\ x\ y$)
  shows *cont* ($\lambda x.$ *let* $y = f\ x$ *in* $g\ x\ y$)
unfolding *Let-def* **using** *f g2 g1* **by** (*rule cont-apply*)

lemma *cont2cont-Let'* [*simp*, *cont2cont*]:
  assumes *f*: *cont* ($\lambda x.\ f\ x$)
  assumes *g*: *cont* ($\lambda p.\ g$ (*fst p*) (*snd p*))
  shows *cont* ($\lambda x.$ *let* $y = f\ x$ *in* $g\ x\ y$)
using *f*
proof (*rule cont2cont-Let*)
  fix *x* **show** *cont* ($\lambda y.\ g\ x\ y$)
    using *g* **by** (*simp add*: *prod-cont-iff*)
next
  fix *y* **show** *cont* ($\lambda x.\ g\ x\ y$)
    using *g* **by** (*simp add*: *prod-cont-iff*)
qed

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

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

end

9 The Strict Function Type

theory *Sfun*
imports *Cfun*
begin

pcpodef (*'a*, *'b*) *sfun* (**infixr** $\rightarrow!$ *0*)
  $= \{f :: 'a \rightarrow 'b.\ f\cdot\bot = \bot\}$
by *simp-all*

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

TODO: Define nice syntax for abstraction, application.

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

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

lemma *sfun-rep-beta*: *sfun-rep*·*f* = *Rep-sfun f*
unfolding *sfun-rep-def* **by** (*simp add*: *cont-Rep-sfun*)

lemma *sfun-rep-strict1* [*simp*]: *sfun-rep*·⊥ = ⊥
unfolding *sfun-rep-beta* **by** (*rule Rep-sfun-strict*)

lemma *sfun-rep-strict2* [*simp*]: *sfun-rep*·*f*·⊥ = ⊥
unfolding *sfun-rep-beta* **by** (*rule Rep-sfun* [*simplified*])

lemma *strictify-cancel*: *f*·⊥ = ⊥ ⟹ *strictify*·*f* = *f*
by (*simp add*: *cfun-eq-iff strictify-conv-if*)

lemma *sfun-abs-sfun-rep* [*simp*]: *sfun-abs*·(*sfun-rep*·*f*) = *f*
unfolding *sfun-abs-def sfun-rep-def*
apply (*simp add*: *cont-Abs-sfun cont-Rep-sfun*)
apply (*simp add*: *Rep-sfun-inject* [*symmetric*] *Abs-sfun-inverse*)
apply (*simp add*: *cfun-eq-iff strictify-conv-if*)
apply (*simp add*: *Rep-sfun* [*simplified*])
done

lemma *sfun-rep-sfun-abs* [*simp*]: *sfun-rep*·(*sfun-abs*·*f*) = *strictify*·*f*
unfolding *sfun-abs-def sfun-rep-def*
apply (*simp add*: *cont-Abs-sfun cont-Rep-sfun*)
apply (*simp add*: *Abs-sfun-inverse*)
done

lemma *sfun-eq-iff*: *f* = *g* ⟷ *sfun-rep*·*f* = *sfun-rep*·*g*
by (*simp add*: *sfun-rep-def cont-Rep-sfun Rep-sfun-inject*)

lemma *sfun-below-iff*: *f* ⊑ *g* ⟷ *sfun-rep*·*f* ⊑ *sfun-rep*·*g*
by (*simp add*: *sfun-rep-def cont-Rep-sfun below-sfun-def*)

end

10 The cpo of cartesian products

theory *Cprod*
imports *Cfun*
begin

default-sort *cpo*

10.1 Continuous case function for unit type

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

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

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

10.2 Continuous version of split function

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

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

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

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

10.3 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

11 The type of strict products

theory *Sprod*
imports *Cfun*
begin

default-sort *pcpo*

11.1 Definition of strict product type

definition *sprod* = {*p*::′*a* × ′*b*. *p* = ⊥ ∨ (*fst p* ≠ ⊥ ∧ *snd p* ≠ ⊥)}

pcpodef (′*a*, ′*b*) *sprod* ((- ⊗/ -) [*21*,*20*] *20*) = *sprod* :: (′*a* × ′*b*) *set*
unfolding *sprod-def* **by** *simp-all*

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

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

11.2 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* (*seq*·*b*·*a*, *seq*·*a*·*b*))

definition
ssplit :: (′*a* → ′*b* → ′*c*) → (′*a* ∗∗ ′*b*) → ′*c* **where**
ssplit = (Λ *f p*. *seq*·*p*·(*f*·(*sfst*·*p*)·(*ssnd*·*p*)))

syntax
-stuple :: [*logic*, *args*] ⇒ *logic* ((*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*)

11.3 Case analysis

lemma *spair-sprod*: (*seq*·*b*·*a*, *seq*·*a*·*b*) ∈ *sprod*
by (*simp add*: *sprod-def seq-conv-if*)

lemma *Rep-sprod-spair*: *Rep-sprod* (:*a*, *b*:) = (*seq*·*b*·*a*, *seq*·*a*·*b*)
by (*simp add*: *spair-def cont-Abs-sprod Abs-sprod-inverse spair-sprod*)

lemmas *Rep-sprod-simps* =

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

lemma *sprodE* [*case-names bottom spair*, *cases type*: *sprod*]:
obtains $p = \bot \mid x\ y$ **where** $p = (:x,\ y:)$ **and** $x \neq \bot$ **and** $y \neq \bot$
using *Rep-sprod* [*of p*] **by** (*auto simp add*: *sprod-def Rep-sprod-simps*)

lemma *sprod-induct* [*case-names bottom spair*, *induct type*: *sprod*]:
$[\![P\ \bot;\ \bigwedge x\ y.\ [\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow P\ (:x,\ y:)]\!] \Longrightarrow P\ x$
by (*cases x*, *simp-all*)

11.4 Properties of *spair*

lemma *spair-strict1* [*simp*]: $(:\bot,\ y:) = \bot$
by (*simp add*: *Rep-sprod-simps*)

lemma *spair-strict2* [*simp*]: $(:x,\ \bot:) = \bot$
by (*simp add*: *Rep-sprod-simps*)

lemma *spair-bottom-iff* [*simp*]: $((:x,\ y:) = \bot) = (x = \bot \vee y = \bot)$
by (*simp add*: *Rep-sprod-simps seq-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 seq-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 seq-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-below*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) = (x \sqsubseteq a \wedge y \sqsubseteq b)$
by (*simp add*: *spair-below-iff*)

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*: $\bot = (:\bot, \bot:)$
by *simp*

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

11.5 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-bottom-iff* [*simp*]: $(sfst\cdot p = \bot) = (p = \bot)$
by (*cases p*, *simp-all*)

lemma *ssnd-bottom-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 *spair-sfst-ssnd*: $(:sfst\cdot p, ssnd\cdot p:) = p$
by (*cases p*, *simp-all*)

lemma *below-sprod*: $(x \sqsubseteq y) = (sfst\cdot x \sqsubseteq sfst\cdot y \wedge ssnd\cdot x \sqsubseteq ssnd\cdot y)$
by (*simp add*: *Rep-sprod-simps sfst-def ssnd-def cont-Rep-sprod*)

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

apply (*cases* $x = \bot$, *simp*, *cases* $y = \bot$, *simp*)
apply (*simp add*: *below-sprod*)
done

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

11.6 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 seq-conv-if*)

lemma *compact-spair-iff*:
  compact $(:x,\ y:) = (x = \bot \vee y = \bot \vee ($*compact* $x \wedge$ *compact* $y))$
apply (*safe elim!*: *compact-spair*)
apply (*drule compact-sfst*, *simp*)
apply (*drule compact-ssnd*, *simp*)
apply *simp*
apply *simp*
done

11.7 Properties of *ssplit*

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

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

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

11.8 Strict product preserves flatness

instance *sprod* :: (*flat*, *flat*) *flat*
proof
  fix $x\ y :: 'a \otimes 'b$
  assume $x \sqsubseteq y$ **thus** $x = \bot \vee x = y$
    apply (*induct* x, *simp*)
    apply (*induct* y, *simp*)
    apply (*simp add*: *spair-below-iff flat-below-iff*)
    done

qed

end

12 Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

12.1 Discrete cpo class instance

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

definition
(*op* ⊑ :: *'a discr* ⇒ *'a discr* ⇒ *bool*) = (*op* =)

instance
by *standard* (*simp add*: *below-discr-def*)

end

12.2 *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*

end

13 The type of lifted values

theory *Up*
imports *Cfun*
begin

default-sort *cpo*

13.1 Definition of new type for lifting

datatype *'a u* ((-$_\perp$) [*1000*] *999*) = *Ibottom* | *Iup 'a*

primrec *Ifup* :: (*'a* → *'b*::*pcpo*) ⇒ *'a u* ⇒ *'b* **where**
  Ifup f Ibottom = ⊥
| *Ifup f* (*Iup x*) = *f*·*x*

13.2 Ordering on lifted cpo

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

definition
 below-up-def:
  (*op* ⊑) ≡ (λ*x y*. *case x of Ibottom* ⇒ *True* | *Iup a* ⇒
   (*case y of Ibottom* ⇒ *False* | *Iup b* ⇒ *a* ⊑ *b*))

instance ..
end

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

lemma *not-Iup-below* [*iff*]: *Iup x* ⋢ *Ibottom*
by (*simp add*: *below-up-def*)

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

13.3 Lifted cpo is a partial order

instance *u* :: (*cpo*) *po*
proof
 fix *x* :: *'a u*
 show *x* ⊑ *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

13.4 Lifted cpo is a cpo

lemma *is-lub-Iup*:
  range S <<| *x* ⟹ *range* (λ*i*. *Iup* (*S i*)) <<| *Iup x*
unfolding *is-lub-def is-ub-def ball-simps*
by (*auto simp add*: *below-up-def split*: *u.split*)

lemma *up-chain-lemma*:
  assumes *Y*: *chain Y* **obtains** ∀ *i*. *Y i* = *Ibottom*
  | *A k* **where** ∀ *i*. *Iup* (*A i*) = *Y* (*i* + *k*) **and** *chain A* **and** *range Y* <<| *Iup* (⨆ *i*. *A i*)
proof (*cases* ∃ *k*. *Y k* ≠ *Ibottom*)
  case *True*
  then obtain *k* **where** *k*: *Y k* ≠ *Ibottom* **..**
  def *A* ≡ λ*i*. *THE a*. *Iup a* = *Y* (*i* + *k*)
  have *Iup-A*: ∀ *i*. *Iup* (*A i*) = *Y* (*i* + *k*)
  proof
    fix *i* :: *nat*
    from *Y le-add2* **have** *Y k* ⊑ *Y* (*i* + *k*) **by** (*rule chain-mono*)
    with *k* **have** *Y* (*i* + *k*) ≠ *Ibottom* **by** (*cases Y k*, *auto*)
    thus *Iup* (*A i*) = *Y* (*i* + *k*)
      by (*cases Y* (*i* + *k*), *simp-all add*: *A-def*)
  qed
  from *Y* **have** *chain-A*: *chain A*
    unfolding *chain-def Iup-below* [*symmetric*]
    by (*simp add*: *Iup-A*)
  hence *range A* <<| (⨆ *i*. *A i*)
    by (*rule cpo-lubI*)
  hence *range* (λ*i*. *Iup* (*A i*)) <<| *Iup* (⨆ *i*. *A i*)
    by (*rule is-lub-Iup*)
  hence *range* (λ*i*. *Y* (*i* + *k*)) <<| *Iup* (⨆ *i*. *A i*)
    by (*simp only*: *Iup-A*)
  hence *range* (λ*i*. *Y i*) <<| *Iup* (⨆ *i*. *A i*)
    by (*simp only*: *is-lub-range-shift* [*OF Y*])
  with *Iup-A chain-A* **show** *?thesis* **..**
next
  case *False*
  then have ∀ *i*. *Y i* = *Ibottom* **by** *simp*
  then show *?thesis* **..**
qed

instance *u* :: (*cpo*) *cpo*
proof
  fix *S* :: *nat* ⇒ *'a u*
  assume *S*: *chain S*
  thus ∃ *x*. *range* (λ*i*. *S i*) <<| *x*
  proof (*rule up-chain-lemma*)
    assume ∀ *i*. *S i* = *Ibottom*
    hence *range* (λ*i*. *S i*) <<| *Ibottom*
      by (*simp add*: *is-lub-const*)

```
    thus ?thesis ..
  next
    fix A :: nat ⇒ 'a
    assume range S <<| Iup (⨆ i. A i)
    thus ?thesis ..
  qed
qed
```

13.5 Lifted cpo is pointed

```
instance u :: (cpo) pcpo
by intro-classes fast
```

for compatibility with old HOLCF-Version

```
lemma inst-up-pcpo: ⊥ = Ibottom
by (rule minimal-up [THEN bottomI, symmetric])
```

13.6 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 (λf. Ifup f x)
by (induct x, simp-all)

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

lemma cont-Ifup2: cont (λx. Ifup f x)
proof (rule contI2)
  fix Y assume Y: chain Y and Y': chain (λi. Ifup f (Y i))
  from Y show Ifup f (⨆ i. Y i) ⊑ (⨆ i. Ifup f (Y i))
  proof (rule up-chain-lemma)
    fix A and k
    assume A: ∀ i. Iup (A i) = Y (i + k)
    assume chain A and range Y <<| Iup (⨆ i. A i)
    hence Ifup f (⨆ i. Y i) = (⨆ i. Ifup f (Iup (A i)))
      by (simp add: lub-eqI contlub-cfun-arg)
    also have ... = (⨆ i. Ifup f (Y (i + k)))
```

  by (*simp add*: *A*)
 also have ... = ($\bigsqcup$ *i*. *Ifup f* (*Y i*))
  using *Y'* **by** (*rule lub-range-shift*)
 finally show *?thesis* **by** *simp*
qed *simp*
qed (*rule monofun-Ifup2*)

13.7 Continuous versions of constants

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

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

translations
case l of XCONST up·x $\Rightarrow$ *t* == *CONST fup*·(Λ *x*. *t*)·*l*
case l of (*XCONST up* :: $'a$)·*x* $\Rightarrow$ *t* => *CONST fup*·(Λ *x*. *t*)·*l*
Λ(*XCONST up*·*x*). *t* == *CONST fup*·(Λ *x*. *t*)

continuous versions of lemmas for $'a_\bot$

lemma *Exh-Up*: $z = \bot \lor (\exists x.\ z = up \cdot x)$
apply (*induct z*)
apply (*simp add*: *inst-up-pcpo*)
apply (*simp add*: *up-def cont-Iup*)
done

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

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

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

lemma *not-up-less-UU*: $up \cdot x \not\sqsubseteq \bot$
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*]:
$[\![p = \bot \Longrightarrow Q;\ \bigwedge x.\ p = up \cdot x \Longrightarrow Q]\!] \Longrightarrow Q$
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 ⊥; $\bigwedge x$. P (*up*·x)⟧ ⟹ P x
by (*cases* x, *simp-all*)

lifting preserves chain-finiteness

lemma *up-chain-cases*:
assumes Y: *chain* Y **obtains** $\forall i$. $Y\ i = \bot$
| A k **where** $\forall i$. *up*·(A i) = Y ($i + k$) **and** *chain* A **and** ($\bigsqcup i$. Y i) = *up*·($\bigsqcup i$. A i)
apply (*rule up-chain-lemma* [*OF* Y])
apply (*simp-all add*: *inst-up-pcpo up-def cont-Iup lub-eqI*)
done

lemma *compact-up*: *compact* x ⟹ *compact* (*up*·x)
apply (*rule compactI2*)
apply (*erule up-chain-cases*)
apply *simp*
apply (*drule* (*1*) *compactD2*, *simp*)
apply (*erule exE*)
apply (*drule-tac f=up* **and** *x=x* **in** *monofun-cfun-arg*)
apply (*simp*, *erule exI*)
done

lemma *compact-upD*: *compact* (*up*·x) ⟹ *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·⊥ = ⊥
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*)

end

14 Lifting types of class type to flat pcpo's

theory *Lift*
imports *Discrete Up*
begin

default-sort *type*

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

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

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

14.1 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

old-rep-datatype ⊥::*'a lift Def*
  by (*erule lift-induct*) (*simp-all add*: *Def-def Abs-lift-inject inst-lift-pcpo*)

⊥ and *Def*

lemma *not-Undef-is-Def*: (*x* ≠ ⊥) = (∃ *y*. *x* = *Def y*)
  by (*cases x*) *simp-all*

lemma *lift-definedE*: ⟦*x* ≠ ⊥; ⋀*a*. *x* = *Def a* ⟹ *R*⟧ ⟹ *R*
  by (*cases x*) *simp-all*

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

method-setup *defined* = ‹
  Scan.succeed (*fn ctxt* => *SIMPLE-METHOD'*
    (*eresolve-tac ctxt* @{*thms lift-definedE*} *THEN' asm-simp-tac ctxt*))
›

lemma *DefE*: *Def x* = ⊥ ⟹ *R*
  by *simp*

lemma *DefE2*: ⟦*x* = *Def s*; *x* = ⊥⟧ ⟹ *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*)

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

14.2 Lift is flat

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

14.3 Continuity of *case-lift*

lemma *case-lift-eq*: *case-lift* $\bot$ *f x* = *fup*$\cdot$(Λ *y*. *f* (*undiscr y*))$\cdot$(*Rep-lift x*)
apply (*induct x*, *unfold lift.case*)
apply (*simp add*: *Rep-lift-strict*)
apply (*simp add*: *Def-def Abs-lift-inverse*)
done

lemma *cont2cont-case-lift* [*simp*]:
  $[\![\bigwedge y.$ *cont* ($\lambda x.$ *f x y*); *cont g*$]\!] \Longrightarrow$ *cont* ($\lambda x.$ *case-lift* $\bot$ (*f x*) (*g x*))
unfolding *case-lift-eq* **by** (*simp add*: *cont-Rep-lift*)

14.4 Further operations

definition
  flift1 :: (*'a* $\Rightarrow$ *'b::pcpo*) $\Rightarrow$ (*'a lift* $\rightarrow$ *'b*) (**binder** *FLIFT 10*) **where**
  flift1 = (λf. (Λ *x*. *case-lift* $\bot$ *f x*))

translations
  Λ(*XCONST Def x*). *t* => *CONST flift1* (λx. *t*)
  Λ(*CONST Def x*). *FLIFT y*. *t* <= *FLIFT x y*. *t*
  Λ(*CONST Def x*). *t* <= *FLIFT x*. *t*

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

lemma *flift1-Def* [*simp*]: *flift1 f*$\cdot$(*Def x*) = (*f x*)
by (*simp add*: *flift1-def*)

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

lemma *flift1-strict* [*simp*]: *flift1 f*$\cdot\bot = \bot$
by (*simp add*: *flift1-def*)

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

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

lemma *flift2-bottom-iff* [*simp*]: (*flift2* $f \cdot x = \bot$) = ($x = \bot$)
by (*cases* x, *simp-all*)

lemma *FLIFT-mono*:
($\bigwedge x.\ f\ x \sqsubseteq g\ x$) $\Longrightarrow$ (*FLIFT* $x.\ f\ x$) $\sqsubseteq$ (*FLIFT* $x.\ g\ x$)
by (*rule cfun-belowI*, *case-tac* x, *simp-all*)

lemma *cont2cont-flift1* [*simp*, *cont2cont*]:
$[\![\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)]\!] \Longrightarrow cont\ (\lambda x.\ FLIFT\ y.\ f\ x\ y)$
by (*simp add*: *flift1-def cont2cont-LAM*)

end

15 The type of lifted booleans

theory *Tr*
imports *Lift*
begin

15.1 Type definition and constructors

type-synonym
tr = *bool lift*

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

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

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

Exhaustion and Elimination for type *tr*

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

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

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

distinctness for type *tr*

lemma *dist-below-tr* [*simp*]:
  $TT \not\sqsubseteq \bot\ FF \not\sqsubseteq \bot\ TT \not\sqsubseteq FF\ FF \not\sqsubseteq TT$
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*) *simp-all*

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

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

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

15.2 Case analysis

default-sort *pcpo*

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

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

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

lemma *ifte-thms* [*simp*]:
  If $\bot$ *then e1 else e2* = $\bot$
  If FF then e1 else e2 = *e2*
  If TT then e1 else e2 = *e1*
by (*simp-all add*: *tr-case-def TT-def FF-def*)

15.3 Boolean connectives

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

tactic for tr-thms with case split

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

lemmas about andalso, orelse, neg and if

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

lemma *neg-thms* [*simp*]:
$neg{\cdot}TT = FF$
$neg{\cdot}FF = TT$
$neg{\cdot}\bot = \bot$
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*:
$P\ (If2\ Q\ x\ y) = ((Q = \bot \longrightarrow P\ \bot) \wedge (Q = TT \longrightarrow P\ x) \wedge (Q = FF \longrightarrow P\ y))$
apply (*unfold If2-def*)
apply (*cases Q*)
apply (*simp-all*)
done

ML ‹
fun split-If-tac ctxt =
simp-tac (*put-simpset HOL-basic-ss ctxt addsimps* [@{*thm If2-def*} *RS sym*])
THEN' (*split-tac ctxt* [@{*thm split-If2*}])
›

15.4 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*:
$t \neq \bot \Longrightarrow ((t\ andalso\ s) = FF) = (t = FF \vee s = FF)$
apply (*cases t*)
apply *simp-all*
done

lemma *andalso-and*:
$t \neq \bot \Longrightarrow ((t\ andalso\ s) \neq FF) = (t \neq FF \wedge s \neq FF)$
apply (*cases t*)
apply *simp-all*
done

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

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

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

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

lemma *If-and-if*:

  (*If Def P then A else B*) = (*if P then A else B*)
apply (*cases Def P*)
apply (*auto simp add*: *TT-def*[*symmetric*] *FF-def*[*symmetric*])
done

15.5 Compactness

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

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

end

16 The type of strict sums

theory *Ssum*
imports *Tr*
begin

default-sort *pcpo*

16.1 Definition of strict sum type

definition
  ssum =
    {*p* :: *tr* × (′*a* × ′*b*). *p* = ⊥ ∨
      (*fst p* = *TT* ∧ *fst* (*snd p*) ≠ ⊥ ∧ *snd* (*snd p*) = ⊥) ∨
      (*fst p* = *FF* ∧ *fst* (*snd p*) = ⊥ ∧ *snd* (*snd p*) ≠ ⊥)}

pcpodef (′*a*, ′*b*) *ssum* ((- ⊕/ -) [*21*, *20*] *20*) = *ssum* :: (*tr* × ′*a* × ′*b*) *set*
  unfolding *ssum-def* **by** *simp-all*

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

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

16.2 Definitions of constructors

definition
  sinl :: ′*a* → (′*a* ++ ′*b*) **where**
  sinl = (Λ *a*. *Abs-ssum* (*seq*·*a*·*TT*, *a*, ⊥))

definition
  sinr :: ′*b* → (′*a* ++ ′*b*) **where**
  sinr = (Λ *b*. *Abs-ssum* (*seq*·*b*·*FF*, ⊥, *b*))

lemma *sinl-ssum*: $(seq{\cdot}a{\cdot}TT,\ a,\ \bot) \in ssum$
by (*simp add*: *ssum-def seq-conv-if*)

lemma *sinr-ssum*: $(seq{\cdot}b{\cdot}FF,\ \bot,\ b) \in ssum$
by (*simp add*: *ssum-def seq-conv-if*)

lemma *Rep-ssum-sinl*: $Rep\text{-}ssum\ (sinl{\cdot}a) = (seq{\cdot}a{\cdot}TT,\ a,\ \bot)$
by (*simp add*: *sinl-def cont-Abs-ssum Abs-ssum-inverse sinl-ssum*)

lemma *Rep-ssum-sinr*: $Rep\text{-}ssum\ (sinr{\cdot}b) = (seq{\cdot}b{\cdot}FF,\ \bot,\ b)$
by (*simp add*: *sinr-def cont-Abs-ssum Abs-ssum-inverse sinr-ssum*)

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

16.3 Properties of *sinl* and *sinr*

Ordering

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

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

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

lemma *sinr-below-sinl* [*simp*]: $(sinr{\cdot}x \sqsubseteq sinl{\cdot}y) = (x = \bot)$
by (*simp add*: *Rep-ssum-simps seq-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*: *Rep-ssum-simps*)

lemma *sinr-strict* [*simp*]: $sinr \cdot \bot = \bot$
by (*simp add*: *Rep-ssum-simps*)

lemma *sinl-bottom-iff* [*simp*]: $(sinl \cdot x = \bot) = (x = \bot)$
using *sinl-eq* [*of x* $\bot$] **by** *simp*

lemma *sinr-bottom-iff* [*simp*]: $(sinr \cdot x = \bot) = (x = \bot)$
using *sinr-eq* [*of x* $\bot$] **by** *simp*

lemma *sinl-defined*: $x \neq \bot \Longrightarrow sinl \cdot x \neq \bot$
by *simp*

lemma *sinr-defined*: $x \neq \bot \Longrightarrow sinr \cdot x \neq \bot$
by *simp*

Compactness

lemma *compact-sinl*: *compact x* $\Longrightarrow$ *compact* $(sinl \cdot x)$
by (*rule compact-ssum*, *simp add*: *Rep-ssum-sinl*)

lemma *compact-sinr*: *compact x* $\Longrightarrow$ *compact* $(sinr \cdot x)$
by (*rule compact-ssum*, *simp add*: *Rep-ssum-sinr*)

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* $(sinr \cdot x) =$ *compact x*
by (*safe elim*!: *compact-sinr compact-sinrD*)

16.4 Case analysis

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

using *Rep-ssum* [*of p*] **by** (*auto simp add*: *ssum-def Rep-ssum-simps*)

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*)

16.5 Case analysis combinator

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

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

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

lemma *beta-sscase*:
$sscase\cdot f\cdot g\cdot s = (\lambda(t,\ x,\ y).\ \textit{If t then } f\cdot x \textit{ else } g\cdot y)\ (\textit{Rep-ssum } s)$
unfolding *sscase-def* **by** (*simp add*: *cont-Rep-ssum*)

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*)

16.6 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

end

17 The unit domain

theory *One*
imports *Lift*
begin

type-synonym
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*]: $ONE \not\sqsubseteq \bot$
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-case :: *'a::pcpo* → *one* → *'a* **where**
one-case = (Λ *a x*. *seq·x·a*)

translations
case x of XCONST ONE ⇒ *t* == *CONST one-case·t·x*
case x of XCONST ONE :: *'a* ⇒ *t* => *CONST one-case·t·x*
Λ (*XCONST ONE*). *t* == *CONST one-case·t*

lemma *one-case1* [*simp*]: (*case* ⊥ *of ONE* ⇒ *t*) = ⊥
by (*simp add*: *one-case-def*)

lemma *one-case2* [*simp*]: (*case ONE of ONE* ⇒ *t*) = *t*
by (*simp add*: *one-case-def*)

lemma *one-case3* [*simp*]: (*case x of ONE* ⇒ *ONE*) = *x*
by (*induct x rule*: *one-induct*) *simp-all*

end

18 Fixed point operator and admissibility

theory *Fix*
imports *Cfun*
begin

default-sort *pcpo*

18.1 Iteration

primrec *iterate* :: *nat* ⇒ (*'a::cpo* → *'a*) → (*'a* → *'a*) **where**
iterate 0 = (Λ *F x*. *x*)
| *iterate* (*Suc n*) = (Λ *F x*. *F*·(*iterate n·F·x*))

Derive inductive properties of iterate from primitive recursion

lemma *iterate-0* [*simp*]: *iterate 0·F·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* (λi. *iterate i*$\cdot F \cdot \bot$)
by (*rule chainI*, *unfold iterate-Suc2*, *rule monofun-cfun-arg*, *rule minimal*)

18.2 Least fixed point operator

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

Binder syntax for *fix*

abbreviation
fix-syn :: ($'a \Rightarrow 'a$) $\Rightarrow 'a$ (**binder** μ *10*) **where**
fix-syn (λx. *f x*) $\equiv$ *fix*$\cdot$(Λ *x*. *f x*)

notation (*ASCII*)
fix-syn (**binder** *FIX* *10*)

Properties of *fix*

direct connection between *fix* and iteration

lemma *fix-def2*: *fix*$\cdot F$ = ($\bigsqcup i$. *iterate i*$\cdot F \cdot \bot$)
unfolding *fix-def* **by** *simp*

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*: $F \cdot x \sqsubseteq x \Longrightarrow \mathit{fix} \cdot F \sqsubseteq x$
apply (*simp add*: *fix-def2*)
apply (*rule lub-below*)
apply (*rule chain-iterate*)
apply (*induct-tac i*)
apply *simp*
apply *simp*
apply (*erule rev-below-trans*)
apply (*erule monofun-cfun-arg*)
done

lemma *fix-least*: $F \cdot x = x \Longrightarrow \mathit{fix} \cdot F \sqsubseteq x$
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 $\mathit{fix} \cdot F = x$
apply (*rule below-antisym*)
apply (*rule fix-least* [*OF fixed*])
apply (*rule least* [*OF fix-eq* [*symmetric*]])
done

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

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

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

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

strictness of *fix*

lemma *fix-bottom-iff*: $(\mathit{fix} \cdot F = \bot) = (F \cdot \bot = \bot)$
apply (*rule iffI*)
apply (*erule subst*)
apply (*rule fix-eq* [*symmetric*])
apply (*erule fix-least* [*THEN bottomI*])
done

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

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

18.3 Fixed point induction

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

lemma *cont-fix-ind*:
  $[\![\mathit{cont}\ F;\ \mathit{adm}\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F\ x)]\!] \Longrightarrow P\ (\mathit{fix}{\cdot}(\mathit{Abs\text{-}cfun}\ F))$
by (*simp add*: *fix-ind*)

lemma *def-fix-ind*:
  $[\![f \equiv \mathit{fix}{\cdot}F;\ \mathit{adm}\ P;\ P\ \bot;\ \bigwedge x.\ P\ x \Longrightarrow P\ (F{\cdot}x)]\!] \Longrightarrow P\ f$
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 $P\ (\mathit{fix}{\cdot}F)$
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*: $\mathit{adm}\ (\lambda x.\ P\ (\mathit{fst}\ x)\ (\mathit{snd}\ x))$
  assumes *base*: $P\ \bot\ \bot$
  assumes *step*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (F{\cdot}x)\ (G{\cdot}y)$
  shows $P\ (\mathit{fix}{\cdot}F)\ (\mathit{fix}{\cdot}G)$
proof –

from *adm* **have** *adm′*: *adm* (*case-prod 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 $\bigwedge i.$ *case-prod P* (*iterate* $i \cdot F \cdot \bot$, *iterate* $i \cdot G \cdot \bot$)
  by *simp*
hence *case-prod P* ($\bigsqcup i.$ (*iterate* $i \cdot F \cdot \bot$, *iterate* $i \cdot G \cdot \bot$))
  by − (*rule admD* [*OF adm′*], *simp*, *assumption*)
hence *case-prod P* ($\bigsqcup i.$ *iterate* $i \cdot F \cdot \bot$, $\bigsqcup i.$ *iterate* $i \cdot G \cdot \bot$)
  by (*simp add*: *lub-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

lemma *cont-parallel-fix-ind*:
  assumes *cont F* **and** *cont G*
  assumes *adm* ($\lambda x.$ *P* (*fst x*) (*snd x*))
  assumes *P* $\bot$ $\bot$
  assumes $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (F\ x)\ (G\ y)$
  shows *P* (*fix*$\cdot$(*Abs-cfun F*)) (*fix*$\cdot$(*Abs-cfun G*))
by (*rule parallel-fix-ind*, *simp-all add*: *assms*)

18.4 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*: *prod-eq-iff*)
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* ⊑ *x* **by** (*simp add*: *fix-least-below*)
hence *snd* (*F*·(*?x*, *y*)) ⊑ *snd* (*F*·(*x*, *y*))
by (*simp add*: *snd-monofun monofun-cfun*)
hence *snd* (*F*·(*?x*, *y*)) ⊑ *y* **using** *F-y* **by** *simp*
hence *2*: *?y* ⊑ *y* **by** (*simp add*: *fix-least-below*)
show (*?x*, *?y*) ⊑ *z* **using** *z 1 2* **by** *simp*
qed

end

19 Plain HOLCF

theory *Plain-HOLCF*
imports *Cfun Sfun Cprod Sprod Ssum Up Discrete Lift One Tr Fix*
begin

Basic HOLCF concepts and types; does not include definition packages.

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

end

20 Package for defining recursive functions in HOLCF

theory *Fixrec*
imports *Plain-HOLCF*
keywords *fixrec* :: *thy-decl*
begin

20.1 Pattern-match monad

default-sort *cpo*

pcpodef *'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*)))

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*]: $succeed \cdot x \neq \bot$
by (*simp add*: *succeed-def cont-Abs-match Abs-match-bottom-iff*)

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

lemma *succeed-eq* [*simp*]: $(succeed \cdot x = succeed \cdot y) = (x = y)$
by (*simp add*: *succeed-def cont-Abs-match Abs-match-inject*)

lemma *succeed-neq-fail* [*simp*]:
$succeed \cdot x \neq fail$ $fail \neq succeed \cdot x$
by (*simp-all add*: *succeed-def fail-def cont-Abs-match Abs-match-inject*)

20.1.1 Run operator

definition
run :: *'a match* → *'a*::*pcpo* **where**
$run = (\Lambda\ m.\ sscase \cdot \bot \cdot (fup \cdot ID) \cdot (Rep\text{-}match\ m))$

rewrite rules for run

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

lemma *run-fail* [*simp*]: $run \cdot fail = \bot$
unfolding *run-def fail-def*
by (*simp add*: *cont-Rep-match Abs-match-inverse*)

lemma *run-succeed* [*simp*]: $run \cdot (succeed \cdot x) = x$
unfolding *run-def succeed-def*
by (*simp add*: *cont-Rep-match cont-Abs-match Abs-match-inverse*)

20.1.2 Monad plus operator

definition
mplus :: *'a match* → *'a match* → *'a match* **where**
$mplus = (\Lambda\ m1\ m2.\ sscase \cdot (\Lambda\ \text{-}.\ m2) \cdot (\Lambda\ \text{-}.\ m1) \cdot (Rep\text{-}match\ m1))$

abbreviation
mplus-syn :: [*'a match*, *'a match*] ⇒ *'a match* (**infixr** +++ *65*) **where**
$m1 +\!\!+\!\!+ m2 == mplus \cdot m1 \cdot m2$

rewrite rules for mplus

lemma *mplus-strict* [*simp*]: $\bot +\!\!+\!\!+ m = \bot$
unfolding *mplus-def*

by (*simp add*: *cont-Rep-match Rep-match-strict*)

lemma *mplus-fail* [*simp*]: *fail* +++ *m* = *m*
unfolding *mplus-def fail-def*
by (*simp add*: *cont-Rep-match Abs-match-inverse*)

lemma *mplus-succeed* [*simp*]: *succeed·x* +++ *m* = *succeed·x*
unfolding *mplus-def succeed-def*
by (*simp add*: *cont-Rep-match cont-Abs-match Abs-match-inverse*)

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-bottom :: *'a* → *'c match* → *'c match*
where
match-bottom = (Λ *x k*. *seq·x·fail*)

definition
match-Pair :: *'a::cpo* × *'b::cpo* → (*'a* → *'b* → *'c match*) → *'c match*
where
match-Pair = (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*)

definition
match-FF :: *tr* → *'c match* → *'c match*
where
match-FF = (Λ *x k*. *If x then fail else k*)

lemma *match-bottom-simps* [*simp*]:
match-bottom·x·k = (*if x* = ⊥ *then* ⊥ *else fail*)
by (*simp add*: *match-bottom-def*)

lemma *match-Pair-simps* [*simp*]:
match-Pair·(*x*, *y*)*·k* = *k·x·y*
by (*simp-all add*: *match-Pair-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*]:
  $match\text{-}TT \cdot TT \cdot k = k$
  $match\text{-}TT \cdot FF \cdot k = fail$
  $match\text{-}TT \cdot \bot \cdot k = \bot$
by (*simp-all add*: *match-TT-def*)

lemma *match-FF-simps* [*simp*]:
  $match\text{-}FF \cdot FF \cdot k = k$
  $match\text{-}FF \cdot TT \cdot k = fail$
  $match\text{-}FF \cdot \bot \cdot k = \bot$
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*: $[\![x \equiv fst\ p;\ y \equiv snd\ p]\!] \Longrightarrow (x,\ y) \equiv p$
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

ML-file *Tools/holcf-library.ML*
ML-file *Tools/fixrec.ML*

method-setup *fixrec-simp* = ‹
  Scan.succeed (*SIMPLE-METHOD′ o Fixrec.fixrec-simp-tac*)
› *pattern prover for fixrec constants*

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-Pair*}),
(@{*const-name ONE*}, @{*const-name match-ONE*}),
(@{*const-name TT*}, @{*const-name match-TT*}),
(@{*const-name FF*}, @{*const-name match-FF*}),
(@{*const-name bottom*}, @{*const-name match-bottom*})]
⟩

hide-const (**open**) *succeed fail run*

end

21 Continuous deflations and ep-pairs

theory *Deflation*
imports *Plain-HOLCF*
begin

default-sort *cpo*

21.1 Continuous deflations

locale *deflation* =
fixes $d :: {'a} \rightarrow {'a}$
assumes *idem*: $\bigwedge x.\ d{\cdot}(d{\cdot}x) = d{\cdot}x$
assumes *below*: $\bigwedge x.\ d{\cdot}x \sqsubseteq x$
begin

lemma *below-ID*: $d \sqsubseteq ID$
by (*rule cfun-belowI*, *simp add*: *below*)

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

lemma *fixes-eq-range*: $\{x.\ d{\cdot}x = x\} = range\ (\lambda x.\ d{\cdot}x)$
by (*auto simp add*: *eq-sym-conv idem*)

lemma *range-eq-fixes*: $range\ (\lambda x.\ d{\cdot}x) = \{x.\ d{\cdot}x = x\}$
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: $\bigwedge x.\ d{\cdot}x = x \Longrightarrow f{\cdot}x = x$ **shows** $d \sqsubseteq f$
proof (*rule cfun-belowI*)
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 bottomI*])

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

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

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

lemma *deflation-below-iff*:
[\![*deflation p*; *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·(f·x) = f·x by (rule f.idem)
  finally show f·x ⊑ f·(g·x) .
qed

lemma deflation-below-comp2:
  ⟦deflation f; deflation g; f ⊑ g⟧ ⟹ g·(f·x) = f·x
by (simp only: deflation.belowD deflation.idem)
```

21.2 Deflations with finite range

```
lemma finite-range-imp-finite-fixes:
  finite (range f) ⟹ finite {x. f x = x}
proof −
  have {x. f x = x} ⊆ 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·x = x}
begin

lemma finite-range: finite (range (λx. d·x))
by (simp add: range-eq-fixes finite-fixes)

lemma finite-image: finite ((λx. d·x) ‘ A)
by (rule finite-subset [OF image-mono [OF subset-UNIV] finite-range])

lemma compact: compact (d·x)
proof (rule compactI2)
  fix Y :: nat ⇒ 'a
  assume Y: chain Y
  have finite-chain (λi. d·(Y i))
  proof (rule finite-range-imp-finch)
    show chain (λi. d·(Y i))
      using Y by simp
    have range (λi. d·(Y i)) ⊆ range (λx. d·x)
      by clarsimp
    thus finite (range (λi. d·(Y i)))
      using finite-range by (rule finite-subset)
  qed
  hence ∃j. (⨆i. d·(Y i)) = d·(Y j)
    by (simp add: finite-chain-def maxinch-is-thelub Y)
  then obtain j where j: (⨆i. d·(Y i)) = d·(Y j) ..

  assume d·x ⊑ (⨆i. Y i)
  hence d·(d·x) ⊑ d·(⨆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

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

lemma *finite-deflation-imp-deflation*:
finite-deflation d $\Longrightarrow$ *deflation d*
unfolding *finite-deflation-def* **by** *simp*

lemma *finite-deflation-bottom*: *finite-deflation* $\bot$
by *standard simp-all*

21.3 Continuous embedding-projection pairs

locale *ep-pair* =
fixes $e :: 'a \rightarrow 'b$ **and** $p :: 'b \rightarrow 'a$
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* $(\lambda y.\ e{\cdot}x \not\sqsubseteq y)$ **by** (*rule compactD*)
  hence *adm* $(\lambda y.\ e{\cdot}x \not\sqsubseteq e{\cdot}y)$ **by** (*rule adm-subst* [*OF cont-Rep-cfun2*])
  hence *adm* $(\lambda y.\ x \not\sqsubseteq y)$ **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* $(\lambda y.\ x \not\sqsubseteq y)$ **by** (*rule compactD*)
  hence *adm* $(\lambda y.\ x \not\sqsubseteq p{\cdot}y)$ **by** (*rule adm-subst* [*OF cont-Rep-cfun2*])
  hence *adm* $(\lambda y.\ e{\cdot}x \not\sqsubseteq y)$ **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)·((e oo d oo p)·x) = (e oo d oo p)·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 (rule finite-deflation-intro)
  have deflation d ..
  thus deflation (p oo d oo e)
   using d by (rule deflation-p-d-e)
 next
  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

end
```

21.4 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 cfun-belowI)
 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 $p1 \sqsubseteq p2$
proof (*rule cfun-belowI*)
  fix *x*
  have $e\cdot(p1\cdot x) \sqsubseteq x$
    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*)

21.5 Composing ep-pairs

lemma *ep-pair-ID-ID*: *ep-pair ID ID*
by *standard 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 e p*
  for *e* :: *'a::pcpo* → *'b::pcpo*
  and *p* :: *'b::pcpo* → *'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-bottom-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

22 Map functions for various types

theory *Map-Functions*
imports *Deflation*
begin

22.1 Map operator for continuous function space

default-sort *cpo*

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

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

lemma *cfun-map-ID*: *cfun-map*$\cdot ID\cdot ID = ID$
unfolding *cfun-eq-iff* **by** *simp*

lemma *cfun-map-map*:
cfun-map$\cdot f1\cdot g1\cdot($*cfun-map*$\cdot f2\cdot g2\cdot p) =$
cfun-map$\cdot(\Lambda\ x.\ f2\cdot(f1\cdot x))\cdot(\Lambda\ x.\ g1\cdot(g2\cdot x))\cdot p$
by (*rule cfun-eqI*) *simp*

lemma *ep-pair-cfun-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*cfun-map*$\cdot p1\cdot e2$) (*cfun-map*$\cdot e1\cdot p2$)

```
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: cfun-eq-iff)
  fix g show cfun-map·p1·e2·(cfun-map·e1·p2·g) ⊑ g
    apply (rule cfun-belowI, 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: cfun-eq-iff d1.idem d2.idem)
  show cfun-map·d1·d2·f ⊑ f
    apply (rule cfun-belowI, 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 (λx. a·x))
  assumes b: finite (range (λy. b·y))
  shows finite (range (λf. cfun-map·a·b·f))  (is finite (range ?h))
proof (rule finite-imageD)
  let ?f = λg. range (λx. (a·x, g·x))
  show finite (?f ‘ range ?h)
  proof (rule finite-subset)
    let ?B = Pow (range (λx. a·x) × range (λy. 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 cfun-eqI, clarsimp)
    fix x f g
    assume range (λx. (a·x, b·(f·(a·x)))) = range (λx. (a·x, b·(g·(a·x))))
    hence range (λx. (a·x, b·(f·(a·x)))) ⊆ range (λx. (a·x, b·(g·(a·x))))
```

```
    by (rule equalityD1)
  hence (a·x, b·(f·(a·x))) ∈ range (λx. (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 (rule finite-deflation-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
```

Finite deflations are compact elements of the function space

```
lemma finite-deflation-imp-compact: finite-deflation d ⟹ compact d
apply (frule finite-deflation-imp-deflation)
apply (subgoal-tac compact (cfun-map·d·d·d))
apply (simp add: cfun-map-def deflation.idem eta-cfun)
apply (rule finite-deflation.compact)
apply (simp only: finite-deflation-cfun-map)
done
```

22.2 Map operator for product type

```
definition
 prod-map :: ('a → 'b) → ('c → 'd) → 'a × 'c → 'b × 'd
where
 prod-map = (Λ f g p. (f·(fst p), g·(snd p)))

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

lemma prod-map-ID: prod-map·ID·ID = ID
unfolding cfun-eq-iff by auto

lemma prod-map-map:
 prod-map·f1·g1·(prod-map·f2·g2·p) =
```

prod-map·(Λ *x*. *f1*·(*f2*·*x*))·(Λ *x*. *g1*·(*g2*·*x*))·*p*
by (*induct p*) *simp*

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

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

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

22.3 Map function for lifted cpo

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

lemma *u-map-strict* [*simp*]: *u-map*·*f*·⊥ = ⊥
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*: *cfun-eq-iff 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 *u-map-oo*: *u-map*·(*f oo g*) = *u-map*·*f oo u-map*·*g*
by (*simp add*: *cfcomp1 u-map-map eta-cfun*)

lemma *ep-pair-u-map*: *ep-pair e p* ⟹ *ep-pair* (*u-map*·*e*) (*u-map*·*p*)
apply *standard*
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 *standard*
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 (*rule finite-deflation-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

22.4 Map function for strict products

default-sort *pcpo*

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*]:
  $x \neq \bot \Longrightarrow y \neq \bot \Longrightarrow$ *sprod-map*$\cdot f\cdot g\cdot(:x,\ y:) = (:f\cdot x,\ g\cdot y:)$
by (*simp add*: *sprod-map-def*)

lemma *sprod-map-spair'*:
  $f\cdot\bot = \bot \Longrightarrow g\cdot\bot = \bot \Longrightarrow$ *sprod-map*$\cdot f\cdot g\cdot(:x,\ y:) = (:f\cdot x,\ g\cdot y:)$
by (*cases* $x = \bot \vee y = \bot$) *auto*

lemma *sprod-map-ID*: *sprod-map*$\cdot$*ID*$\cdot$*ID* = *ID*
unfolding *sprod-map-def* **by** (*simp add*: *cfun-eq-iff eta-cfun*)

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

lemma *ep-pair-sprod-map*:
  assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
  shows *ep-pair* (*sprod-map*$\cdot e1\cdot e2$) (*sprod-map*$\cdot p1\cdot p2$)
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*$\cdot p1\cdot p2\cdot($*sprod-map*$\cdot e1\cdot e2\cdot x) = x$
    by (*induct x*) *simp-all*
  fix *y* **show** *sprod-map*$\cdot e1\cdot e2\cdot($*sprod-map*$\cdot p1\cdot p2\cdot y) \sqsubseteq y$
    apply (*induct y*, *simp*)
    apply (*case-tac* $p1\cdot x = \bot$, *simp*, *case-tac* $p2\cdot y = \bot$, *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*$\cdot d1\cdot d2$)
proof
  interpret *d1*: *deflation d1* **by** *fact*
  interpret *d2*: *deflation d2* **by** *fact*
  fix *x*
  show *sprod-map*$\cdot d1\cdot d2\cdot($*sprod-map*$\cdot d1\cdot d2\cdot x) =$ *sprod-map*$\cdot d1\cdot d2\cdot x$
    apply (*induct x*, *simp*)
    apply (*case-tac* $d1\cdot x = \bot$, *simp*, *case-tac* $d2\cdot y = \bot$, *simp*)
    apply (*simp add*: *d1.idem d2.idem*)

done
show *sprod-map·d1·d2·x* $\sqsubseteq$ *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 (*rule finite-deflation-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

22.5 Map function for strict sums

definition
ssum-map :: ('*a* → '*b*) → ('*c* → '*d*) → '*a* ⊕ '*c* → '*b* ⊕ '*d*
where
ssum-map = (Λ *f g*. *sscase*·(*sinl oo f*)·(*sinr oo g*))

lemma *ssum-map-strict* [*simp*]: *ssum-map·f·g*·⊥ = ⊥
unfolding *ssum-map-def* **by** *simp*

lemma *ssum-map-sinl* [*simp*]: *x* ≠ ⊥ ⟹ *ssum-map·f·g*·(*sinl·x*) = *sinl*·(*f·x*)
unfolding *ssum-map-def* **by** *simp*

lemma *ssum-map-sinr* [*simp*]: *x* ≠ ⊥ ⟹ *ssum-map·f·g*·(*sinr·x*) = *sinr*·(*g·x*)
unfolding *ssum-map-def* **by** *simp*

lemma *ssum-map-sinl'*: *f*·⊥ = ⊥ ⟹ *ssum-map·f·g*·(*sinl·x*) = *sinl*·(*f·x*)
by (*cases x* = ⊥) *simp-all*

lemma *ssum-map-sinr'*: *g*·⊥ = ⊥ ⟹ *ssum-map·f·g*·(*sinr·x*) = *sinr*·(*g·x*)
by (*cases x* = ⊥) *simp-all*

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

lemma *ssum-map-map*:
⟦*f1*·⊥ = ⊥; *g1*·⊥ = ⊥⟧ ⟹

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 (*rule finite-deflation-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. ssum-map·d1·d2·x = x} ⊆
        (λx. sinl·x) ` {x. d1·x = x} ∪
        (λx. sinr·x) ` {x. d2·x = x} ∪ {⊥}
    by (rule subsetI, case-tac x, simp-all)
  thus finite {x. ssum-map·d1·d2·x = x}
    by (rule finite-subset, simp add: d1.finite-fixes d2.finite-fixes)
qed
```

22.6 Map operator for strict function space

```
definition
  sfun-map :: ('b → 'a) → ('c → 'd) → ('a →! 'c) → ('b →! 'd)
where
  sfun-map = (Λ a b. sfun-abs oo cfun-map·a·b oo sfun-rep)

lemma sfun-map-ID: sfun-map·ID·ID = ID
  unfolding sfun-map-def
  by (simp add: cfun-map-ID cfun-eq-iff)

lemma sfun-map-map:
  assumes f2·⊥ = ⊥ and g2·⊥ = ⊥ shows
  sfun-map·f1·g1·(sfun-map·f2·g2·p) =
    sfun-map·(Λ x. f2·(f1·x))·(Λ x. g1·(g2·x))·p
unfolding sfun-map-def
by (simp add: cfun-eq-iff strictify-cancel assms cfun-map-map)

lemma ep-pair-sfun-map:
  assumes 1: ep-pair e1 p1
  assumes 2: ep-pair e2 p2
  shows ep-pair (sfun-map·p1·e2) (sfun-map·e1·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 f show sfun-map·e1·p2·(sfun-map·p1·e2·f) = f
    unfolding sfun-map-def
    apply (simp add: sfun-eq-iff strictify-cancel)
    apply (rule ep-pair.e-inverse)
    apply (rule ep-pair-cfun-map [OF 1 2])
    done
  fix g show sfun-map·p1·e2·(sfun-map·e1·p2·g) ⊑ g
    unfolding sfun-map-def
    apply (simp add: sfun-below-iff strictify-cancel)
    apply (rule ep-pair.e-p-below)
    apply (rule ep-pair-cfun-map [OF 1 2])
    done
qed
```

```
lemma deflation-sfun-map:
  assumes 1: deflation d1
  assumes 2: deflation d2
  shows deflation (sfun-map·d1·d2)
apply (simp add: sfun-map-def)
apply (rule deflation.intro)
apply simp
apply (subst strictify-cancel)
apply (simp add: cfun-map-def deflation-strict 1 2)
apply (simp add: cfun-map-def deflation.idem 1 2)
apply (simp add: sfun-below-iff)
apply (subst strictify-cancel)
apply (simp add: cfun-map-def deflation-strict 1 2)
apply (rule deflation.below)
apply (rule deflation-cfun-map [OF 1 2])
done

lemma finite-deflation-sfun-map:
  assumes 1: finite-deflation d1
  assumes 2: finite-deflation d2
  shows finite-deflation (sfun-map·d1·d2)
proof (intro finite-deflation-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 (sfun-map·d1·d2) by (rule deflation-sfun-map)
  from 1 2 have finite-deflation (cfun-map·d1·d2)
    by (rule finite-deflation-cfun-map)
  then have finite {f. cfun-map·d1·d2·f = f}
    by (rule finite-deflation.finite-fixes)
  moreover have inj (λf. sfun-rep·f)
    by (rule inj-onI, simp add: sfun-eq-iff)
  ultimately have finite ((λf. sfun-rep·f) -` {f. cfun-map·d1·d2·f = f})
    by (rule finite-vimageI)
  then show finite {f. sfun-map·d1·d2·f = f}
    unfolding sfun-map-def sfun-eq-iff
    by (simp add: strictify-cancel
         deflation-strict ‹deflation d1› ‹deflation d2›)
qed

end
```

23 Profinite and bifinite cpos

```
theory Bifinite
imports Map-Functions ~~/src/HOL/Library/Countable
begin

default-sort cpo
```

23.1 Chains of finite deflations

locale *approx-chain* =
fixes *approx* :: *nat* ⇒ ′*a* → ′*a*
assumes *chain-approx* [*simp*]: *chain* (λ*i*. *approx i*)
assumes *lub-approx* [*simp*]: (⨆ *i*. *approx i*) = *ID*
assumes *finite-deflation-approx* [*simp*]: ⋀*i*. *finite-deflation* (*approx i*)
begin

lemma *deflation-approx*: *deflation* (*approx i*)
using *finite-deflation-approx* **by** (*rule finite-deflation-imp-deflation*)

lemma *approx-idem*: *approx i*·(*approx i*·*x*) = *approx i*·*x*
using *deflation-approx* **by** (*rule deflation.idem*)

lemma *approx-below*: *approx i*·*x* ⊑ *x*
using *deflation-approx* **by** (*rule deflation.below*)

lemma *finite-range-approx*: *finite* (*range* (λ*x*. *approx i*·*x*))
apply (*rule finite-deflation.finite-range*)
apply (*rule finite-deflation-approx*)
done

lemma *compact-approx* [*simp*]: *compact* (*approx n*·*x*)
apply (*rule finite-deflation.compact*)
apply (*rule finite-deflation-approx*)
done

lemma *compact-eq-approx*: *compact x* ⟹ ∃ *i*. *approx i*·*x* = *x*
by (*rule admD2*, *simp-all*)

end

23.2 Omega-profinite and bifinite domains

class *bifinite* = *pcpo* +
assumes *bifinite*: ∃ (*a*::*nat* ⇒ ′*a* → ′*a*). *approx-chain a*

class *profinite* = *cpo* +
assumes *profinite*: ∃ (*a*::*nat* ⇒ ′$a_\bot$ → ′$a_\bot$). *approx-chain a*

23.3 Building approx chains

lemma *approx-chain-iso*:
assumes *a*: *approx-chain a*
assumes [*simp*]: ⋀*x*. *f*·(*g*·*x*) = *x*
assumes [*simp*]: ⋀*y*. *g*·(*f*·*y*) = *y*
shows *approx-chain* (λ*i*. *f oo a i oo g*)
proof −
have *1*: *f oo g* = *ID* **by** (*simp add*: *cfun-eqI*)

have *2*: *ep-pair f g* **by** (*simp add*: *ep-pair-def*)
from *1 2* **show** *?thesis*
using *a* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP ep-pair.finite-deflation-e-d-p*)
qed

lemma *approx-chain-u-map*:
assumes *approx-chain a*
shows *approx-chain* ($\lambda i.$ *u-map*$\cdot$(*a i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP u-map-ID finite-deflation-u-map*)

lemma *approx-chain-sfun-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* ($\lambda i.$ *sfun-map*$\cdot$(*a i*)$\cdot$(*b i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP sfun-map-ID finite-deflation-sfun-map*)

lemma *approx-chain-sprod-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* ($\lambda i.$ *sprod-map*$\cdot$(*a i*)$\cdot$(*b i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP sprod-map-ID finite-deflation-sprod-map*)

lemma *approx-chain-ssum-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* ($\lambda i.$ *ssum-map*$\cdot$(*a i*)$\cdot$(*b i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP ssum-map-ID finite-deflation-ssum-map*)

lemma *approx-chain-cfun-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* ($\lambda i.$ *cfun-map*$\cdot$(*a i*)$\cdot$(*b i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP cfun-map-ID finite-deflation-cfun-map*)

lemma *approx-chain-prod-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* ($\lambda i.$ *prod-map*$\cdot$(*a i*)$\cdot$(*b i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP prod-map-ID finite-deflation-prod-map*)

Approx chains for countable discrete types.

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

lemma *chain-discr-approx* [*simp*]: *chain discr-approx*
unfolding *discr-approx-def*
by (*rule chainI*, *simp add*: *monofun-cfun monofun-LAM*)

lemma *lub-discr-approx* [*simp*]: $(\bigsqcup i.\ \mathit{discr\text{-}approx}\ i) = \mathit{ID}$
apply (*rule cfun-eqI*)
apply (*simp add*: *contlub-cfun-fun*)
apply (*simp add*: *discr-approx-def*)
apply (*case-tac x*, *simp*)
apply (*rule lub-eqI*)
apply (*rule is-lubI*)
apply (*rule ub-rangeI*, *simp*)
apply (*drule ub-rangeD*)
apply (*erule rev-below-trans*)
apply *simp*
apply (*rule lessI*)
done

lemma *inj-on-undiscr* [*simp*]: *inj-on undiscr A*
using *Discr-undiscr* **by** (*rule inj-on-inverseI*)

lemma *finite-deflation-discr-approx*: *finite-deflation* (*discr-approx i*)
proof
  fix $x :: 'a\ \mathit{discr}\ u$
  show $\mathit{discr\text{-}approx}\ i \cdot x \sqsubseteq x$
    unfolding *discr-approx-def*
    by (*cases x*, *simp*, *simp*)
  show $\mathit{discr\text{-}approx}\ i \cdot (\mathit{discr\text{-}approx}\ i \cdot x) = \mathit{discr\text{-}approx}\ i \cdot x$
    unfolding *discr-approx-def*
    by (*cases x*, *simp*, *simp*)
  show *finite* $\{x::'a\ \mathit{discr}\ u.\ \mathit{discr\text{-}approx}\ i \cdot x = x\}$
  proof (*rule finite-subset*)
    let $?S = \mathit{insert}\ (\bot::'a\ \mathit{discr}\ u)\ ((\lambda x.\ \mathit{up} \cdot x)\ `\ \mathit{undiscr}\ -`\ \mathit{to\text{-}nat}\ -`\ \{..<i\})$
    show $\{x::'a\ \mathit{discr}\ u.\ \mathit{discr\text{-}approx}\ i \cdot x = x\} \subseteq\ ?S$
      unfolding *discr-approx-def*
      by (*rule subsetI*, *case-tac x*, *simp*, *simp split*: *if-split-asm*)
    show *finite ?S*
      by (*simp add*: *finite-vimageI*)
  qed
qed

lemma *discr-approx*: *approx-chain discr-approx*
using *chain-discr-approx lub-discr-approx finite-deflation-discr-approx*
by (*rule approx-chain.intro*)

23.4 Class instance proofs

instance *bifinite* ⊆ *profinite*
proof
  show $\exists (a::\mathit{nat} \Rightarrow 'a_\bot \rightarrow 'a_\bot).\ \mathit{approx\text{-}chain}\ a$
    using *bifinite* [**where** $'a='a$]
    by (*fast intro*!: *approx-chain-u-map*)

qed

instance *u* :: (*profinite*) *bifinite*
  by *standard* (*rule profinite*)

Types $'a \to 'b$ and $'a_\bot \to! 'b$ are isomorphic.

definition *encode-cfun* = (Λ *f*. *sfun-abs*·(*fup*·*f*))

definition *decode-cfun* = (Λ *g x*. *sfun-rep*·*g*·(*up*·*x*))

lemma *decode-encode-cfun* [*simp*]: *decode-cfun*·(*encode-cfun*·*x*) = *x*
unfolding *encode-cfun-def decode-cfun-def*
by (*simp add*: *eta-cfun*)

lemma *encode-decode-cfun* [*simp*]: *encode-cfun*·(*decode-cfun*·*y*) = *y*
unfolding *encode-cfun-def decode-cfun-def*
apply (*simp add*: *sfun-eq-iff strictify-cancel*)
apply (*rule cfun-eqI*, *case-tac x*, *simp-all*)
done

instance *cfun* :: (*profinite*, *bifinite*) *bifinite*
proof
  obtain *a* :: *nat* $\Rightarrow$ $'a_\bot \to 'a_\bot$ **where** *a*: *approx-chain a*
    using *profinite* **..**
  obtain *b* :: *nat* $\Rightarrow$ $'b \to 'b$ **where** *b*: *approx-chain b*
    using *bifinite* **..**
  have *approx-chain* (λ*i*. *decode-cfun oo sfun-map*·(*a i*)·(*b i*) *oo encode-cfun*)
    using *a b* **by** (*simp add*: *approx-chain-iso approx-chain-sfun-map*)
  thus $\exists$(*a*::*nat* $\Rightarrow$ $('a \to 'b) \to ('a \to 'b)$). *approx-chain a*
    by − (*rule exI*)
qed

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

definition *encode-prod-u* = (Λ(*up*·(*x*, *y*)). (:*up*·*x*, *up*·*y*:))

definition *decode-prod-u* = (Λ(:*up*·*x*, *up*·*y*:). *up*·(*x*, *y*))

lemma *decode-encode-prod-u* [*simp*]: *decode-prod-u*·(*encode-prod-u*·*x*) = *x*
unfolding *encode-prod-u-def decode-prod-u-def*
by (*case-tac x*, *simp*, *rename-tac y*, *case-tac y*, *simp*)

lemma *encode-decode-prod-u* [*simp*]: *encode-prod-u*·(*decode-prod-u*·*y*) = *y*
unfolding *encode-prod-u-def decode-prod-u-def*
apply (*case-tac y*, *simp*, *rename-tac a b*)
apply (*case-tac a*, *simp*, *case-tac b*, *simp*, *simp*)
done

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

obtain *a* :: *nat* ⇒ $'a_\bot$ → $'a_\bot$ **where** *a*: *approx-chain a*
  using *profinite* **..**
obtain *b* :: *nat* ⇒ $'b_\bot$ → $'b_\bot$ **where** *b*: *approx-chain b*
  using *profinite* **..**
have *approx-chain* (λ*i*. *decode-prod-u oo sprod-map*·(*a i*)·(*b i*) *oo encode-prod-u*)
  using *a b* **by** (*simp add*: *approx-chain-iso approx-chain-sprod-map*)
thus ∃(*a*::*nat* ⇒ $('a \times 'b)_\bot$ → $('a \times 'b)_\bot$). *approx-chain a*
  by − (*rule exI*)
qed

instance *prod* :: (*bifinite*, *bifinite*) *bifinite*
proof
  show ∃(*a*::*nat* ⇒ ('*a* × '*b*) → ('*a* × '*b*)). *approx-chain a*
  using *bifinite* [**where** '*a*='*a*] **and** *bifinite* [**where** '*a*='*b*]
  by (*fast intro*!: *approx-chain-prod-map*)
qed

instance *sfun* :: (*bifinite*, *bifinite*) *bifinite*
proof
  show ∃(*a*::*nat* ⇒ ('*a* →! '*b*) → ('*a* →! '*b*)). *approx-chain a*
  using *bifinite* [**where** '*a*='*a*] **and** *bifinite* [**where** '*a*='*b*]
  by (*fast intro*!: *approx-chain-sfun-map*)
qed

instance *sprod* :: (*bifinite*, *bifinite*) *bifinite*
proof
  show ∃(*a*::*nat* ⇒ ('*a* ⊗ '*b*) → ('*a* ⊗ '*b*)). *approx-chain a*
  using *bifinite* [**where** '*a*='*a*] **and** *bifinite* [**where** '*a*='*b*]
  by (*fast intro*!: *approx-chain-sprod-map*)
qed

instance *ssum* :: (*bifinite*, *bifinite*) *bifinite*
proof
  show ∃(*a*::*nat* ⇒ ('*a* ⊕ '*b*) → ('*a* ⊕ '*b*)). *approx-chain a*
  using *bifinite* [**where** '*a*='*a*] **and** *bifinite* [**where** '*a*='*b*]
  by (*fast intro*!: *approx-chain-ssum-map*)
qed

lemma *approx-chain-unit*: *approx-chain* (⊥ :: *nat* ⇒ *unit* → *unit*)
by (*simp add*: *approx-chain-def cfun-eq-iff finite-deflation-bottom*)

instance *unit* :: *bifinite*
  by *standard* (*fast intro*!: *approx-chain-unit*)

instance *discr* :: (*countable*) *profinite*
  by *standard* (*fast intro*!: *discr-approx*)

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

note [*simp*] = *cont-Abs-lift cont-Rep-lift Rep-lift-inverse Abs-lift-inverse*
obtain *a* :: *nat* ⇒ (′*a discr*)$_\perp$ → (′*a discr*)$_\perp$ **where** *a*: *approx-chain a*
using *profinite* **..**
hence *approx-chain* (λ*i*. (Λ *y*. *Abs-lift y*) *oo a i oo* (Λ *x*. *Rep-lift x*))
by (*rule approx-chain-iso*) *simp-all*
thus ∃(*a*::*nat* ⇒ ′*a lift* → ′*a lift*). *approx-chain a*
by − (*rule exI*)
qed

end

24 Defining algebraic domains by ideal completion

theory *Completion*
imports *Plain-HOLCF*
begin

24.1 Ideals over a preorder

locale *preorder* =
fixes *r* :: ′*a*::*type* ⇒ ′*a* ⇒ *bool* (**infix** ⪯ *50*)
assumes *r-refl*: *x* ⪯ *x*
assumes *r-trans*: ⟦*x* ⪯ *y*; *y* ⪯ *z*⟧ ⟹ *x* ⪯ *z*
begin

definition
ideal :: ′*a set* ⇒ *bool* **where**
ideal A = ((∃*x*. *x* ∈ *A*) ∧ (∀*x*∈*A*. ∀*y*∈*A*. ∃*z*∈*A*. *x* ⪯ *z* ∧ *y* ⪯ *z*) ∧
(∀*x y*. *x* ⪯ *y* ⟶ *y* ∈ *A* ⟶ *x* ∈ *A*))

lemma *idealI*:
assumes ∃*x*. *x* ∈ *A*
assumes ⋀*x y*. ⟦*x* ∈ *A*; *y* ∈ *A*⟧ ⟹ ∃*z*∈*A*. *x* ⪯ *z* ∧ *y* ⪯ *z*
assumes ⋀*x y*. ⟦*x* ⪯ *y*; *y* ∈ *A*⟧ ⟹ *x* ∈ *A*
shows *ideal A*
unfolding *ideal-def* **using** *assms* **by** *fast*

lemma *idealD1*:
ideal A ⟹ ∃*x*. *x* ∈ *A*
unfolding *ideal-def* **by** *fast*

lemma *idealD2*:
⟦*ideal A*; *x* ∈ *A*; *y* ∈ *A*⟧ ⟹ ∃*z*∈*A*. *x* ⪯ *z* ∧ *y* ⪯ *z*
unfolding *ideal-def* **by** *fast*

lemma *idealD3*:
⟦*ideal A*; *x* ⪯ *y*; *y* ∈ *A*⟧ ⟹ *x* ∈ *A*
unfolding *ideal-def* **by** *fast*

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*: $\exists A.\ A \in \{A.\ ideal\ A\}$
by (*fast intro*: *ideal-principal*)

The set of ideals is a cpo

lemma *ideal-UN*:
  fixes $A :: nat \Rightarrow 'a\ set$
  assumes *ideal-A*: $\bigwedge i.\ ideal\ (A\ i)$
  assumes *chain-A*: $\bigwedge i\ j.\ i \le j \Longrightarrow A\ i \subseteq A\ j$
  shows *ideal* $(\bigcup i.\ A\ i)$
 apply (*rule idealI*)
   apply (*cut-tac idealD1* [*OF ideal-A*], *fast*)
  apply (*clarify*, *rename-tac i j*)
  apply (*drule subsetD* [*OF chain-A* [*OF max.cobounded1*]])
  apply (*drule subsetD* [*OF chain-A* [*OF max.cobounded2*]])
  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 $Abs :: 'a\ set \Rightarrow 'b::below$
  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*, *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 $Abs :: 'a\ set \Rightarrow 'b::po$
  assumes *type*: *type-definition Rep Abs* $\{S.\ ideal\ S\}$
  assumes *below*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow Rep\ x \subseteq Rep\ y$
  assumes *S*: *chain S*
  shows *typedef-ideal-lub*: $range\ S <<|\ Abs\ (\bigcup i.\ Rep\ (S\ i))$
   and *typedef-ideal-rep-lub*: $Rep\ (\bigsqcup i.\ S\ i) = (\bigcup i.\ Rep\ (S\ i))$

proof –
have *1*: *ideal* ($\bigcup$*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* ($\bigcup$*i*. *Rep* (*S i*))) = ($\bigcup$*i*. *Rep* (*S i*))
by (*simp add*: *type-definition.Abs-inverse* [*OF type*])
show *3*: *range S* <<| *Abs* ($\bigcup$*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*: ($\bigsqcup$*i*. *S i*) = *Abs* ($\bigcup$*i*. *Rep* (*S i*))
by (*rule lub-eqI*)
show *5*: *Rep* ($\bigsqcup$*i*. *S i*) = ($\bigcup$*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* ⊑ *y* ⟷ *Rep x* ⊆ *Rep y*
shows *OFCLASS*(*'b*, *cpo-class*)
by *standard* (*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

24.2 Lemmas about least upper bounds

lemma *is-ub-thelub-ex*: ⟦∃ *u*. *S* <<| *u*; *x* ∈ *S*⟧ ⟹ *x* ⊑ *lub S*
apply (*erule exE*, *drule is-lub-lub*)
apply (*drule is-lubD1*)
apply (*erule* (*1*) *is-ubD*)
done

lemma *is-lub-thelub-ex*: ⟦∃ *u*. *S* <<| *u*; *S* <| *x*⟧ ⟹ *lub S* ⊑ *x*
by (*erule exE*, *drule is-lub-lub*, *erule is-lubD2*)

24.3 Locale for ideal completion

locale *ideal-completion* = *preorder* +

fixes *principal* :: $'a$::*type* ⇒ $'b$::*cpo*
fixes *rep* :: $'b$::*cpo* ⇒ $'a$::*type set*
assumes *ideal-rep*: ⋀*x*. *ideal* (*rep x*)
assumes *rep-lub*: ⋀*Y*. *chain Y* ⟹ *rep* (⨆*i*. *Y i*) = (⋃*i*. *rep* (*Y i*))
assumes *rep-principal*: ⋀*a*. *rep* (*principal a*) = {*b*. *b* ≼ *a*}
assumes *belowI*: ⋀*x y*. *rep x* ⊆ *rep y* ⟹ *x* ⊑ *y*
assumes *countable*: ∃*f*::$'a$ ⇒ *nat*. *inj f*
begin

lemma *rep-mono*: *x* ⊑ *y* ⟹ *rep x* ⊆ *rep y*
apply (*frule bin-chain*)
apply (*drule rep-lub*)
apply (*simp only*: *lub-eqI* [*OF is-lub-bin-chain*])
apply (*rule subsetI*, *rule UN-I* [**where** *a=0*], *simp-all*)
done

lemma *below-def*: *x* ⊑ *y* ⟷ *rep x* ⊆ *rep y*
by (*rule iffI* [*OF rep-mono belowI*])

lemma *principal-below-iff-mem-rep*: *principal a* ⊑ *x* ⟷ *a* ∈ *rep x*
unfolding *below-def rep-principal*
by (*auto intro*: *r-refl elim*: *idealD3* [*OF ideal-rep*])

lemma *principal-below-iff* [*simp*]: *principal a* ⊑ *principal b* ⟷ *a* ≼ *b*
by (*simp add*: *principal-below-iff-mem-rep rep-principal*)

lemma *principal-eq-iff*: *principal a* = *principal b* ⟷ *a* ≼ *b* ∧ *b* ≼ *a*
unfolding *po-eq-conv* [**where** $'a$=$'b$] *principal-below-iff* **..**

lemma *eq-iff*: *x* = *y* ⟷ *rep x* = *rep y*
unfolding *po-eq-conv below-def* **by** *auto*

lemma *principal-mono*: *a* ≼ *b* ⟹ *principal a* ⊑ *principal b*
by (*simp only*: *principal-below-iff*)

lemma *ch2ch-principal* [*simp*]:
∀*i*. *Y i* ≼ *Y* (*Suc i*) ⟹ *chain* (λ*i*. *principal* (*Y i*))
by (*simp add*: *chainI principal-mono*)

24.3.1 Principal ideals approximate all elements

lemma *compact-principal* [*simp*]: *compact* (*principal a*)
by (*rule compactI2*, *simp add*: *principal-below-iff-mem-rep rep-lub*)

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

lemma *obtain-principal-chain*:
obtains *Y* **where** ∀*i*. *Y i* ≼ *Y* (*Suc i*) **and** *x* = (⨆*i*. *principal* (*Y i*))
proof −
obtain *count* :: $'a$ ⇒ *nat* **where** *inj*: *inj count*

```
    using countable ..
  def enum ≡ λi. THE a. count a = i
  have enum-count [simp]: ⋀x. enum (count x) = x
    unfolding enum-def by (simp add: inj-eq [OF inj])
  def a ≡ LEAST i. enum i ∈ rep x
  def b ≡ λi. LEAST j. enum j ∈ rep x ∧ ¬ enum j ⪯ enum i
  def c ≡ λi j. LEAST k. enum k ∈ rep x ∧ enum i ⪯ enum k ∧ enum j ⪯ enum
k
  def P ≡ λi. ∃j. enum j ∈ rep x ∧ ¬ enum j ⪯ enum i
  def X ≡ rec-nat a (λn i. if P i then c i (b i) else i)
  have X-0: X 0 = a unfolding X-def by simp
  have X-Suc: ⋀n. X (Suc n) = (if P (X n) then c (X n) (b (X n)) else X n)
    unfolding X-def by simp
  have a-mem: enum a ∈ rep x
    unfolding a-def
    apply (rule LeastI-ex)
    apply (cut-tac ideal-rep [of x])
    apply (drule idealD1)
    apply (clarify, rename-tac a)
    apply (rule-tac x=count a in exI, simp)
    done
  have b: ⋀i. P i ⟹ enum i ∈ rep x
    ⟹ enum (b i) ∈ rep x ∧ ¬ enum (b i) ⪯ enum i
    unfolding P-def b-def by (erule LeastI2-ex, simp)
  have c: ⋀i j. enum i ∈ rep x ⟹ enum j ∈ rep x
    ⟹ enum (c i j) ∈ rep x ∧ enum i ⪯ enum (c i j) ∧ enum j ⪯ enum (c i j)
    unfolding c-def
    apply (drule (1) idealD2 [OF ideal-rep], clarify)
    apply (rule-tac a=count z in LeastI2, simp, simp)
    done
  have X-mem: ⋀n. enum (X n) ∈ rep x
    apply (induct-tac n)
    apply (simp add: X-0 a-mem)
    apply (clarsimp simp add: X-Suc, rename-tac n)
    apply (simp add: b c)
    done
  have X-chain: ⋀n. enum (X n) ⪯ enum (X (Suc n))
    apply (clarsimp simp add: X-Suc r-refl)
    apply (simp add: b c X-mem)
    done
  have less-b: ⋀n i. n < b i ⟹ enum n ∈ rep x ⟹ enum n ⪯ enum i
    unfolding b-def by (drule not-less-Least, simp)
  have X-covers: ⋀n. ∀k≤n. enum k ∈ rep x ⟶ enum k ⪯ enum (X n)
    apply (induct-tac n)
    apply (clarsimp simp add: X-0 a-def)
    apply (drule-tac k=0 in Least-le, simp add: r-refl)
    apply (clarsimp, rename-tac n k)
    apply (erule le-SucE)
    apply (rule r-trans [OF - X-chain], simp)
```

apply (*case-tac P* (*X n*), *simp add*: *X-Suc*)
apply (*rule-tac x=b* (*X n*) **and** *y=Suc n* **in** *linorder-cases*)
apply (*simp only*: *less-Suc-eq-le*)
apply (*drule spec*, *drule* (*1*) *mp*, *simp add*: *b X-mem*)
apply (*simp add*: *c X-mem*)
apply (*drule* (*1*) *less-b*)
apply (*erule r-trans*)
apply (*simp add*: *b c X-mem*)
apply (*simp add*: *X-Suc*)
apply (*simp add*: *P-def*)
done
have *1*: $\forall i.$ *enum* (*X i*) $\preceq$ *enum* (*X* (*Suc i*))
by (*simp add*: *X-chain*)
have *2*: $x = (\bigsqcup n.$ *principal* (*enum* (*X n*)))
apply (*simp add*: *eq-iff rep-lub 1 rep-principal*)
apply (*auto*, *rename-tac a*)
apply (*subgoal-tac* $\exists i.\ a =$ *enum i*, *erule exE*)
apply (*rule-tac x=i* **in** *exI*, *simp add*: *X-covers*)
apply (*rule-tac x=count a* **in** *exI*, *simp*)
apply (*erule idealD3* [*OF ideal-rep*])
apply (*rule X-mem*)
done
from *1 2* **show** *?thesis* **..**
qed

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes *P*: $\bigwedge a.$ *P* (*principal a*)
shows *P x*
apply (*rule obtain-principal-chain* [*of x*])
apply (*simp add*: *admD* [*OF adm*] *P*)
done

lemma *compact-imp-principal*: *compact x* $\Longrightarrow \exists a.\ x =$ *principal a*
apply (*rule obtain-principal-chain* [*of x*])
apply (*drule adm-compact-neq* [*OF - cont-id*])
apply (*subgoal-tac chain* ($\lambda i.$ *principal* (*Y i*)))
apply (*drule* (*2*) *admD2*, *fast*, *simp*)
done

24.4 Defining functions in terms of basis elements

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

lemma *extension-lemma*:
fixes *f* :: *'a::type* $\Rightarrow$ *'c::cpo*
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$

shows $\exists\, u.\ f\ `\ rep\ x <<|\ u$
proof –
obtain Y **where** $Y: \forall\, i.\ Y\ i \preceq Y\ (Suc\ i)$
and $x: x = (\bigsqcup i.\ principal\ (Y\ i))$
by (*rule obtain-principal-chain* [*of x*])
have *chain*: $chain\ (\lambda i.\ f\ (Y\ i))$
by (*rule chainI*, *simp add*: *f-mono Y*)
have *rep-x*: $rep\ x = (\bigcup n.\ \{a.\ a \preceq Y\ n\})$
by (*simp add*: *x rep-lub Y rep-principal*)
have $f\ `\ rep\ x <<|\ (\bigsqcup n.\ f\ (Y\ n))$
apply (*rule is-lubI*)
apply (*rule ub-imageI*, *rename-tac a*)
apply (*clarsimp simp add*: *rep-x*)
apply (*drule f-mono*)
apply (*erule below-lub* [*OF chain*])
apply (*rule lub-below* [*OF chain*])
apply (*drule-tac x=Y n* **in** *ub-imageD*)
apply (*simp add*: *rep-x*, *fast intro*: *r-refl*)
apply *assumption*
done
thus *?thesis* **..**
qed

lemma *extension-beta*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $extension\ f{\cdot}x = lub\ (f\ `\ rep\ x)$
unfolding *extension-def*
proof (*rule beta-cfun*)
have *lub*: $\bigwedge x.\ \exists\, u.\ f\ `\ rep\ x <<|\ u$
using *f-mono* **by** (*rule extension-lemma*)
show *cont*: $cont\ (\lambda x.\ lub\ (f\ `\ rep\ x))$
apply (*rule contI2*)
apply (*rule monofunI*)
apply (*rule is-lub-thelub-ex* [*OF lub ub-imageI*])
apply (*rule is-ub-thelub-ex* [*OF lub imageI*])
apply (*erule* (*1*) *subsetD* [*OF rep-mono*])
apply (*rule is-lub-thelub-ex* [*OF lub ub-imageI*])
apply (*simp add*: *rep-lub*, *clarify*)
apply (*erule rev-below-trans* [*OF is-ub-thelub*])
apply (*erule is-ub-thelub-ex* [*OF lub imageI*])
done
qed

lemma *extension-principal*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $extension\ f{\cdot}(principal\ a) = f\ a$
apply (*subst extension-beta*, *erule f-mono*)

apply (*subst rep-principal*)
apply (*rule lub-eqI*)
apply (*rule is-lub-maximal*)
apply (*rule ub-imageI*)
apply (*simp add*: *f-mono*)
apply (*rule imageI*)
apply (*simp add*: *r-refl*)
done

lemma *extension-mono*:
  assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
  assumes *g-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow g\ a \sqsubseteq g\ b$
  assumes *below*: $\bigwedge a.\ f\ a \sqsubseteq g\ a$
  shows *extension f* $\sqsubseteq$ *extension g*
 apply (*rule cfun-belowI*)
 apply (*simp only*: *extension-beta f-mono g-mono*)
 apply (*rule is-lub-thelub-ex*)
  apply (*rule extension-lemma*, *erule f-mono*)
 apply (*rule ub-imageI*, *rename-tac a*)
 apply (*rule below-trans* [*OF below*])
 apply (*rule is-ub-thelub-ex*)
  apply (*rule extension-lemma*, *erule g-mono*)
 apply (*erule imageI*)
done

lemma *cont-extension*:
  assumes *f-mono*: $\bigwedge a\ b\ x.\ a \preceq b \Longrightarrow f\ x\ a \sqsubseteq f\ x\ b$
  assumes *f-cont*: $\bigwedge a.$ *cont* $(\lambda x.\ f\ x\ a)$
  shows *cont* $(\lambda x.$ *extension* $(\lambda a.\ f\ x\ a))$
 apply (*rule contI2*)
  apply (*rule monofunI*)
  apply (*rule extension-mono*, *erule f-mono*, *erule f-mono*)
  apply (*erule cont2monofunE* [*OF f-cont*])
 apply (*rule cfun-belowI*)
 apply (*rule principal-induct*, *simp*)
 apply (*simp only*: *contlub-cfun-fun*)
 apply (*simp only*: *extension-principal f-mono*)
 apply (*simp add*: *cont2contlubE* [*OF f-cont*])
done

end

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

proof
  interpret *type-definition Rep Abs* $\{S.\ ideal\ S\}$ **by** *fact*
  fix $a\ b :: 'a$ **and** $x\ y :: 'b$ **and** $Y :: nat \Rightarrow 'b$
  show *ideal* (*Rep x*)
    using *Rep* [*of x*] **by** *simp*
  show $chain\ Y \Longrightarrow Rep\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ Rep\ (Y\ i))$
    using *type below* **by** (*rule typedef-ideal-rep-lub*)
  show $Rep\ (principal\ a) = \{b.\ b \preceq a\}$
    by (*simp add*: *principal Abs-inverse ideal-principal*)
  show $Rep\ x \subseteq Rep\ y \Longrightarrow x \sqsubseteq y$
    by (*simp only*: *below*)
  show $\exists f::'a \Rightarrow nat.\ inj\ f$
    by (*rule countable*)
qed

end

25 A universal bifinite domain

theory *Universal*
imports *Bifinite Completion* ~~/src/HOL/Library/Nat-Bijection
begin

25.1 Basis for universal domain

25.1.1 Basis datatype

type-synonym *ubasis* = *nat*

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

lemma *node-not-0* [*simp*]: $node\ i\ a\ S \neq 0$
unfolding *node-def* **by** *simp*

lemma *node-gt-0* [*simp*]: $0 < node\ i\ a\ S$
unfolding *node-def* **by** *simp*

lemma *node-inject* [*simp*]:
  $[\![finite\ S;\ finite\ T]\!]$
    $\Longrightarrow node\ i\ a\ S = node\ j\ b\ T \longleftrightarrow i = j \wedge a = b \wedge S = T$
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*:
$[\![fst\ (prod\text{-}decode\ x) = a;\ snd\ (prod\text{-}decode\ x) = b]\!] \Longrightarrow x = prod\text{-}encode\ (a,\ b)$
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

25.1.2 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*⟧ $\Longrightarrow$ *ubasis-le a c*
| *ubasis-le-lower*:
  finite S $\Longrightarrow$ *ubasis-le a* (*node i a S*)
| *ubasis-le-upper*:
  ⟦*finite S*; *b* $\in$ *S*; *ubasis-le a b*⟧ $\Longrightarrow$ *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

interpretation *udom*: *preorder ubasis-le*
apply *standard*
apply (*rule ubasis-le-refl*)
apply (*erule* (*1*) *ubasis-le-trans*)
done

25.1.3 Generic take function

function
  ubasis-until :: (*ubasis* $\Rightarrow$ *bool*) $\Rightarrow$ *ubasis* $\Rightarrow$ *ubasis*
where
  ubasis-until P 0 = *0*
| *finite S* $\Longrightarrow$ *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*
done

termination *ubasis-until*
apply (*relation measure snd*)
apply (*rule wf-measure*)
apply (*simp add*: *node-gt1*)
done

lemma *ubasis-until*: *P 0* $\Longrightarrow$ *P* (*ubasis-until P x*)
by (*induct x rule*: *node-induct*) *simp-all*

lemma *ubasis-until'*: *0* < *ubasis-until P x* $\Longrightarrow$ *P* (*ubasis-until P x*)
by (*induct x rule*: *node-induct*) *auto*

lemma *ubasis-until-same*: $P\ x \Longrightarrow ubasis\text{-}until\ P\ x = x$
by (*induct x rule*: *node-induct*) *simp-all*

lemma *ubasis-until-idem*:
$P\ 0 \Longrightarrow ubasis\text{-}until\ P\ (ubasis\text{-}until\ P\ x) = ubasis\text{-}until\ P\ x$
by (*rule ubasis-until-same* [*OF ubasis-until*])

lemma *ubasis-until-0*:
$\forall x.\ x \neq 0 \longrightarrow \neg\ P\ x \Longrightarrow ubasis\text{-}until\ P\ x = 0$
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 $\bigwedge i\ a\ S\ b.\ [\![finite\ S;\ P\ (node\ i\ a\ S);\ b \in S;\ ubasis\text{-}le\ a\ b]\!] \Longrightarrow P\ b$
shows $ubasis\text{-}le\ a\ b \Longrightarrow ubasis\text{-}le\ (ubasis\text{-}until\ P\ a)\ (ubasis\text{-}until\ P\ b)$
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*) *assms*)
apply (*erule* (*2*) *ubasis-le.ubasis-le-upper*)
done
qed

lemma *finite-range-ubasis-until*:
finite $\{x.\ P\ x\} \Longrightarrow$ *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 Defining the universal domain by ideal completion

typedef *udom* = $\{S.$ *udom.ideal S*$\}$
by (*rule udom.ex-ideal*)

instantiation *udom* :: *below*
begin

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

instance ..
end

instance *udom* :: *po*
using *type-definition-udom below-udom-def*
by (*rule udom.typedef-ideal-po*)

instance *udom* :: *cpo*
using *type-definition-udom below-udom-def*
by (*rule udom.typedef-ideal-cpo*)

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

lemma *ubasis-countable*: $\exists f$::*ubasis* $\Rightarrow$ *nat*. *inj f*
by (*rule exI*, *rule inj-on-id*)

interpretation *udom*:
ideal-completion ubasis-le udom-principal Rep-udom
using *type-definition-udom below-udom-def*
using *udom-principal-def ubasis-countable*
by (*rule udom.typedef-ideal-completion*)

Universal domain is pointed

lemma *udom-minimal*: *udom-principal 0* $\sqsubseteq$ *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*: $\perp$ = *udom-principal 0*
by (*rule udom-minimal* [*THEN bottomI*, *symmetric*])

25.3 Compact bases of domains

typedef *'a compact-basis* = {*x*::*'a*::*pcpo*. *compact x*}
by *auto*

lemma *Rep-compact-basis'* [*simp*]: *compact* (*Rep-compact-basis a*)
by (*rule Rep-compact-basis* [*unfolded mem-Collect-eq*])

lemma *Abs-compact-basis-inverse'* [*simp*]:
compact x $\Longrightarrow$ *Rep-compact-basis* (*Abs-compact-basis x*) = *x*
by (*rule Abs-compact-basis-inverse* [*unfolded mem-Collect-eq*])

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

definition
compact-le-def:
(*op* $\sqsubseteq$) $\equiv$ (λ*x y*. *Rep-compact-basis x* $\sqsubseteq$ *Rep-compact-basis y*)

instance ..
end

instance *compact-basis* :: (*pcpo*) *po*
using *type-definition-compact-basis compact-le-def*
by (*rule typedef-po*)

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

definition
compact-bot :: *'a*::*pcpo compact-basis* **where**
compact-bot = *Abs-compact-basis* $\perp$

lemma *Rep-compact-bot* [*simp*]: *Rep-compact-basis compact-bot* = $\perp$
unfolding *compact-bot-def* **by** *simp*

lemma *compact-bot-minimal* [*simp*]: *compact-bot* $\sqsubseteq$ *a*

unfolding *compact-le-def Rep-compact-bot* **by** *simp*

25.4 Universality of *udom*

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

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

25.4.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
  fixes *A* :: *'a compact-basis set*
  shows ⟦*finite A*; *A* ≠ {}⟧ ⟹ ∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*
proof (*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*:
  $\llbracket$*finite* A; $y \in A$; *choose* $A \sqsubseteq y\rrbracket \Longrightarrow$ *choose* $A = y$
apply (*cases* $A = \{\}$, *simp*)
apply (*frule* (*1*) *choose-lemma*, *simp*)
done

lemma *choose-in*: $\llbracket$*finite* A; $A \neq \{\}\rrbracket \Longrightarrow$ *choose* $A \in A$
by (*frule* (*1*) *choose-lemma*, *simp*)

function
  choose-pos :: *'a compact-basis set* ⇒ *'a compact-basis* ⇒ *nat*
where
  choose-pos A x =
    (*if finite* $A \wedge x \in A \wedge x \neq$ *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* $A \Longrightarrow$ *choose-pos* A (*choose* A) = *0*
by (*simp add*: *choose-pos.simps*)

lemma *inj-on-choose-pos* [*OF refl*]:
  $\llbracket$*card* $A = n$; *finite* $A\rrbracket \Longrightarrow$ *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*: *if-split-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 Compact basis take function

primrec
 cb-take :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis* **where**
 cb-take 0 = (λ*x*. *compact-bot*)
| *cb-take* (*Suc n*) = (λ*a*. *Abs-compact-basis* (*approx n*·(*Rep-compact-basis a*)))

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

lemma *cb-take-zero* [*simp*]: *cb-take 0 a* = *compact-bot*
by (*simp only*: *cb-take.simps*)

lemma *Rep-cb-take*:
 Rep-compact-basis (*cb-take* (*Suc n*) *a*) = *approx n*·(*Rep-compact-basis a*)
by (*simp add*: *cb-take.simps*(*2*))

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

lemma *cb-take-covers*: ∃ *n*. *cb-take n x* = *x*
apply (*subgoal-tac* ∃ *n*. *cb-take* (*Suc n*) *x* = *x*, *fast*)
apply (*simp add*: *Rep-compact-basis-inject* [*symmetric*])
apply (*simp add*: *Rep-cb-take*)
apply (*rule compact-eq-approx*)

apply (*rule Rep-compact-basis'*)
done

lemma *cb-take-less*: *cb-take n x* ⊑ *x*
unfolding *compact-le-def*
by (*cases n*, *simp*, *simp add*: *Rep-cb-take approx-below*)

lemma *cb-take-idem*: *cb-take n* (*cb-take n x*) = *cb-take n x*
unfolding *Rep-compact-basis-inject* [*symmetric*]
by (*cases n*, *simp*, *simp add*: *Rep-cb-take approx-idem*)

lemma *cb-take-mono*: *x* ⊑ *y* ⟹ *cb-take n x* ⊑ *cb-take n y*
unfolding *compact-le-def*
by (*cases n*, *simp*, *simp add*: *Rep-cb-take monofun-cfun-arg*)

lemma *cb-take-chain-le*: *m* ≤ *n* ⟹ *cb-take m x* ⊑ *cb-take n x*
unfolding *compact-le-def*
apply (*cases m*, *simp*, *cases n*, *simp*)
apply (*simp add*: *Rep-cb-take*, *rule chain-mono*, *simp*, *simp*)
done

lemma *finite-range-cb-take*: *finite* (*range* (*cb-take n*))
apply (*cases n*)
apply (*subgoal-tac range* (*cb-take 0*) = {*compact-bot*}, *simp*, *force*)
apply (*rule finite-imageD* [**where** *f*=*Rep-compact-basis*])
apply (*rule finite-subset* [**where** *B*=*range* (λ*x*. *approx* (*n* − *1*)·*x*)])
apply (*clarsimp simp add*: *Rep-cb-take*)
apply (*rule finite-range-approx*)
apply (*rule inj-onI*, *simp add*: *Rep-compact-basis-inject*)
done

25.4.3 Rank of basis elements

definition
  rank :: *'a compact-basis* ⇒ *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* ≤ *n* ⟹ *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* ⟹ *rank x* ≤ *n*
unfolding *rank-def* **by** (*rule Least-le*)

lemma *rank-le-iff*: *rank x* ≤ *n* ⟷ *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* ⟷ *x = compact-bot*
using *rank-le-iff* [*of x 0*] **by** *auto*

definition
  rank-le :: *'a compact-basis* ⇒ *'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*

25.4.4 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*)

25.4.5 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*: $x \neq$ *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*: $x \neq$ *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* $\notin$ *range* (*basis-emb* :: *'a compact-basis* $\Rightarrow$ *nat*)$]\!]$
    $\Longrightarrow$ *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 $\Longrightarrow$ *node i a S* = *basis-emb x* $\longleftrightarrow$
    x $\neq$ *compact-bot* $\wedge$ *i* = *place x* $\wedge$ *a* = *basis-emb* (*sub x*) $\wedge$
        S = *basis-emb* ' {*y*. *place y* < *place x* $\wedge$ *x* $\sqsubseteq$ *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* $\Longrightarrow$ *basis-prj a* $\sqsubseteq$ *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* $\in$ *range* (*basis-emb* :: *'a compact-basis* $\Rightarrow$ *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* $\in$ *range* (*basis-emb* :: *'a compact-basis* $\Rightarrow$ *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

lemma ideal-completion:
  ideal-completion below Rep-compact-basis (approximants :: 'a ⇒ -)
proof
  fix w :: 'a
  show below.ideal (approximants w)
  proof (rule below.idealI)
    have Abs-compact-basis (approx 0·w) ∈ approximants w
      by (simp add: approximants-def approx-below)
    thus ∃ x. x ∈ approximants w ..
  next
    fix x y :: 'a compact-basis
    assume x: x ∈ approximants w and y: y ∈ approximants w
    obtain i where i: approx i·(Rep-compact-basis x) = Rep-compact-basis x
      using compact-eq-approx Rep-compact-basis' by fast
    obtain j where j: approx j·(Rep-compact-basis y) = Rep-compact-basis y
      using compact-eq-approx Rep-compact-basis' by fast
    let ?z = Abs-compact-basis (approx (max i j)·w)
    have ?z ∈ approximants w
      by (simp add: approximants-def approx-below)
    moreover from x y have x ⊑ ?z ∧ y ⊑ ?z
      by (simp add: approximants-def compact-le-def)
       (metis i j monofun-cfun chain-mono chain-approx max.cobounded1 max.cobounded2)
    ultimately show ∃ z ∈ approximants w. x ⊑ z ∧ y ⊑ z ..
  next
    fix x y :: 'a compact-basis
    assume x ⊑ y y ∈ approximants w thus x ∈ approximants w
      unfolding approximants-def compact-le-def
      by (auto elim: below-trans)
  qed
next
  fix Y :: nat ⇒ 'a
  assume chain Y
  thus approximants (⨆ i. Y i) = (⋃ i. approximants (Y i))
    unfolding approximants-def
    by (auto simp add: compact-below-lub-iff)
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
  hence ∀ z. compact z ⟶ z ⊑ x ⟶ z ⊑ y
```

by (*simp add*: *approximants-def subset-eq*)
(*metis Abs-compact-basis-inverse′*)
hence ($\bigsqcup i.\ approx\ i{\cdot}x) \sqsubseteq y$
by (*simp add*: *lub-below approx-below*)
thus $x \sqsubseteq y$
by (*simp add*: *lub-distribs*)
next
show $\exists f{::}'a\ compact\text{-}basis \Rightarrow nat.\ inj\ f$
by (*rule exI*, *rule inj-place*)
qed

end

interpretation *compact-basis*:
ideal-completion below Rep-compact-basis
approximants :: ′*a*::*bifinite* ⇒ ′*a compact-basis set*
proof −
obtain *a* :: *nat* ⇒ ′*a* → ′*a* **where** *approx-chain a*
using *bifinite* **..**
hence *bifinite-approx-chain a*
unfolding *bifinite-approx-chain-def* **.**
thus *ideal-completion below Rep-compact-basis* (*approximants* :: ′*a* ⇒ -)
by (*rule bifinite-approx-chain.ideal-completion*)
qed

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

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

definition
udom-emb :: ′*a* → *udom*
where
udom-emb = *compact-basis.extension* (λ*x*. *udom-principal* (*basis-emb x*))

definition
udom-prj :: *udom* → ′*a*
where
udom-prj = *udom.extension* (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.extension-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.extension-principal*)
apply (*rule compact-basis.principal-mono*)
apply (*erule basis-prj-mono*)
done

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
apply *standard*
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

abbreviation *udom-emb* ≡ *bifinite-approx-chain.udom-emb*
abbreviation *udom-prj* ≡ *bifinite-approx-chain.udom-prj*

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

25.5 Chain of approx functions for type *udom*

definition
udom-approx :: *nat* ⇒ *udom* → *udom*
where
udom-approx i =
udom.extension (λ*x*. *udom-principal* (*ubasis-until* (λ*y*. *y* ≤ *i*) *x*))

lemma *udom-approx-mono*:
ubasis-le a b ⟹
udom-principal (*ubasis-until* (λ*y*. *y* ≤ *i*) *a*) ⊑
udom-principal (*ubasis-until* (λ*y*. *y* ≤ *i*) *b*)
apply (*rule udom.principal-mono*)
apply (*rule ubasis-until-mono*)
apply (*frule* (*2*) *order-less-le-trans* [*OF node-gt2*])
apply (*erule order-less-imp-le*)
apply *assumption*
done

lemma *adm-mem-finite*: ⟦*cont f*; *finite S*⟧ ⟹ *adm* (λ*x*. *f x* ∈ *S*)
by (*erule adm-subst*, *induct set*: *finite*, *simp-all*)

lemma *udom-approx-principal*:
udom-approx i·(*udom-principal x*) =
udom-principal (*ubasis-until* (λ*y*. *y* ≤ *i*) *x*)

unfolding *udom-approx-def*
apply (*rule udom.extension-principal*)
apply (*erule udom-approx-mono*)
done

lemma *finite-deflation-udom-approx*: *finite-deflation* (*udom-approx i*)
proof
  fix *x* **show** *udom-approx i·*(*udom-approx i·x*) = *udom-approx i·x*
    by (*induct x rule*: *udom.principal-induct*, *simp*)
      (*simp add*: *udom-approx-principal ubasis-until-idem*)
next
  fix *x* **show** *udom-approx i·x* ⊑ *x*
    by (*induct x rule*: *udom.principal-induct*, *simp*)
      (*simp add*: *udom-approx-principal ubasis-until-less*)
next
  have ∗: *finite* (*range* (λ*x*. *udom-principal* (*ubasis-until* (λ*y*. *y* ≤ *i*) *x*)))
    apply (*subst range-composition* [**where** *f*=*udom-principal*])
    apply (*simp add*: *finite-range-ubasis-until*)
    done
  show *finite* {*x*. *udom-approx i·x* = *x*}
    apply (*rule finite-range-imp-finite-fixes*)
    apply (*rule rev-finite-subset* [*OF* ∗])
    apply (*clarsimp*, *rename-tac x*)
    apply (*induct-tac x rule*: *udom.principal-induct*)
    apply (*simp add*: *adm-mem-finite* ∗)
    apply (*simp add*: *udom-approx-principal*)
    done
qed

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

lemma *chain-udom-approx* [*simp*]: *chain* (λ*i*. *udom-approx i*)
unfolding *udom-approx-def*
apply (*rule chainI*)
apply (*rule udom.extension-mono*)
apply (*erule udom-approx-mono*)
apply (*erule udom-approx-mono*)
apply (*rule udom.principal-mono*)
apply (*rule ubasis-until-chain*, *simp*)
done

lemma *lub-udom-approx* [*simp*]: (⨆*i*. *udom-approx i*) = *ID*
apply (*rule cfun-eqI*, *simp add*: *contlub-cfun-fun*)
apply (*rule below-antisym*)
apply (*rule lub-below*)
apply (*simp*)
apply (*rule udom-approx.below*)
apply (*rule-tac x*=*x* **in** *udom.principal-induct*)

apply (*simp add*: *lub-distribs*)
apply (*rule-tac i=a* **in** *below-lub*)
apply *simp*
apply (*simp add*: *udom-approx-principal*)
apply (*simp add*: *ubasis-until-same ubasis-le-refl*)
done

lemma *udom-approx* [*simp*]: *approx-chain udom-approx*
proof
  show *chain* (λi. *udom-approx i*)
    by (*rule chain-udom-approx*)
  show ($\bigsqcup i$. *udom-approx i*) = *ID*
    by (*rule lub-udom-approx*)
qed

instance *udom* :: *bifinite*
  by *standard* (*fast intro*: *udom-approx*)

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

end

26 Algebraic deflations

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

default-sort *bifinite*

26.1 Type constructor for finite deflations

typedef *'a fin-defl* = {*d*::*'a* → *'a*. *finite-deflation d*}
by (*fast intro*: *finite-deflation-bottom*)

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

definition *below-fin-defl-def*:
  below ≡ $\lambda x\ y$. *Rep-fin-defl x* ⊑ *Rep-fin-defl y*

instance ..
end

instance *fin-defl* :: (*bifinite*) *po*
using *type-definition-fin-defl below-fin-defl-def*
by (*rule typedef-po*)

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*:
$(\bigwedge x.\ \mathit{Rep\text{-}fin\text{-}defl}\ a{\cdot}x = x \Longrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ b{\cdot}x = x) \Longrightarrow a \sqsubseteq b$
unfolding *below-fin-defl-def*
by (*rule Rep-fin-defl.belowI*)

lemma *fin-defl-belowD*:
$[\![a \sqsubseteq b;\ \mathit{Rep\text{-}fin\text{-}defl}\ a{\cdot}x = x]\!] \Longrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ b{\cdot}x = x$
unfolding *below-fin-defl-def*
by (*rule Rep-fin-defl.belowD*)

lemma *fin-defl-eqI*:
$(\bigwedge x.\ \mathit{Rep\text{-}fin\text{-}defl}\ a{\cdot}x = x \longleftrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ b{\cdot}x = x) \Longrightarrow a = b$
apply (*rule below-antisym*)
apply (*rule fin-defl-belowI*, *simp*)
apply (*rule fin-defl-belowI*, *simp*)
done

lemma *Rep-fin-defl-mono*: $a \sqsubseteq b \Longrightarrow \mathit{Rep\text{-}fin\text{-}defl}\ a \sqsubseteq \mathit{Rep\text{-}fin\text{-}defl}\ b$
unfolding *below-fin-defl-def* **.**

lemma *Abs-fin-defl-mono*:
$[\![\mathit{finite\text{-}deflation}\ a;\ \mathit{finite\text{-}deflation}\ b;\ a \sqsubseteq b]\!]$
$\Longrightarrow \mathit{Abs\text{-}fin\text{-}defl}\ a \sqsubseteq \mathit{Abs\text{-}fin\text{-}defl}\ b$
unfolding *below-fin-defl-def*
by (*simp add*: *Abs-fin-defl-inverse*)

lemma (**in** *finite-deflation*) *compact-belowI*:
assumes $\bigwedge x.\ \mathit{compact}\ x \Longrightarrow d{\cdot}x = x \Longrightarrow f{\cdot}x = x$ **shows** $d \sqsubseteq f$
by (*rule belowI*, *rule assms*, *erule subst*, *rule compact*)

lemma *compact-Rep-fin-defl* [*simp*]: *compact* (*Rep-fin-defl a*)
using *finite-deflation-Rep-fin-defl*
by (*rule finite-deflation-imp-compact*)

26.2 Defining algebraic deflations by ideal completion

typedef *'a defl* = {*S*::*'a fin-defl set*. *below.ideal S*}
by (*rule below.ex-ideal*)

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

```
begin

definition
  x ⊑ y ⟷ Rep-defl x ⊆ Rep-defl y

instance ..
end

instance defl :: (bifinite) po
using type-definition-defl below-defl-def
by (rule below.typedef-ideal-po)

instance defl :: (bifinite) cpo
using type-definition-defl below-defl-def
by (rule below.typedef-ideal-cpo)

definition
  defl-principal :: 'a fin-defl ⇒ 'a defl where
  defl-principal t = Abs-defl {u. u ⊑ t}

lemma fin-defl-countable: ∃f::'a fin-defl ⇒ nat. inj f
proof −
  obtain f :: 'a compact-basis ⇒ nat where inj-f: inj f
    using compact-basis.countable ..
  have ∗: ⋀d. finite (f ‘ Rep-compact-basis −‘ {x. Rep-fin-defl d·x = x})
    apply (rule finite-imageI)
    apply (rule finite-vimageI)
    apply (rule Rep-fin-defl.finite-fixes)
    apply (simp add: inj-on-def Rep-compact-basis-inject)
    done
  have range-eq: range Rep-compact-basis = {x. compact x}
    using type-definition-compact-basis by (rule type-definition.Rep-range)
  have inj (λd. set-encode
    (f ‘ Rep-compact-basis −‘ {x. Rep-fin-defl d·x = x}))
    apply (rule inj-onI)
    apply (simp only: set-encode-eq ∗)
    apply (simp only: inj-image-eq-iff inj-f)
    apply (drule-tac f=image Rep-compact-basis in arg-cong)
    apply (simp del: vimage-Collect-eq add: range-eq set-eq-iff)
    apply (rule Rep-fin-defl-inject [THEN iffD1])
    apply (rule below-antisym)
    apply (rule Rep-fin-defl.compact-belowI, rename-tac z)
    apply (drule-tac x=z in spec, simp)
    apply (rule Rep-fin-defl.compact-belowI, rename-tac z)
    apply (drule-tac x=z in spec, simp)
    done
  thus ?thesis by − (rule exI)
qed
```

interpretation *defl*: *ideal-completion below defl-principal Rep-defl*
using *type-definition-defl below-defl-def*
using *defl-principal-def fin-defl-countable*
by (*rule below.typedef-ideal-completion*)

Algebraic deflations are pointed

lemma *defl-minimal*: *defl-principal* (*Abs-fin-defl* ⊥) ⊑ *x*
apply (*induct x rule*: *defl.principal-induct*, *simp*)
apply (*rule defl.principal-mono*)
apply (*simp add*: *below-fin-defl-def*)
apply (*simp add*: *Abs-fin-defl-inverse finite-deflation-bottom*)
done

instance *defl* :: (*bifinite*) *pcpo*
by *intro-classes* (*fast intro*: *defl-minimal*)

lemma *inst-defl-pcpo*: ⊥ = *defl-principal* (*Abs-fin-defl* ⊥)
by (*rule defl-minimal* [*THEN bottomI*, *symmetric*])

26.3 Applying algebraic deflations

definition
  cast :: *'a defl* → *'a* → *'a*
where
  cast = *defl.extension Rep-fin-defl*

lemma *cast-defl-principal*:
  cast·(*defl-principal a*) = *Rep-fin-defl a*
unfolding *cast-def*
apply (*rule defl.extension-principal*)
apply (*simp only*: *below-fin-defl-def*)
done

lemma *deflation-cast*: *deflation* (*cast*·*d*)
apply (*induct d rule*: *defl.principal-induct*)
apply (*rule adm-subst* [*OF - adm-deflation*], *simp*)
apply (*simp add*: *cast-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 defl.compact-imp-principal*, *clarify*)
apply (*simp add*: *cast-defl-principal*)
apply (*rule finite-deflation-Rep-fin-defl*)
done

interpretation *cast*: *deflation cast*·*d*

by (*rule deflation-cast*)

declare *cast.idem* [*simp*]

lemma *compact-cast* [*simp*]: *compact d* $\Longrightarrow$ *compact* (*cast·d*)
apply (*rule finite-deflation-imp-compact*)
apply (*erule finite-deflation-cast*)
done

lemma *cast-below-cast*: *cast·A* $\sqsubseteq$ *cast·B* $\longleftrightarrow$ *A* $\sqsubseteq$ *B*
apply (*induct A rule*: *defl.principal-induct*, *simp*)
apply (*induct B rule*: *defl.principal-induct*, *simp*)
apply (*simp add*: *cast-defl-principal below-fin-defl-def*)
done

lemma *compact-cast-iff*: *compact* (*cast·d*) $\longleftrightarrow$ *compact d*
apply (*rule iffI*)
apply (*simp only*: *compact-def cast-below-cast* [*symmetric*])
apply (*erule adm-subst* [*OF cont-Rep-cfun2*])
apply (*erule compact-cast*)
done

lemma *cast-below-imp-below*: *cast·A* $\sqsubseteq$ *cast·B* $\Longrightarrow$ *A* $\sqsubseteq$ *B*
by (*simp only*: *cast-below-cast*)

lemma *cast-eq-imp-eq*: *cast·A* = *cast·B* $\Longrightarrow$ *A* = *B*
by (*simp add*: *below-antisym cast-below-imp-below*)

lemma *cast-strict1* [*simp*]: *cast·*$\bot$ = $\bot$
apply (*subst inst-defl-pcpo*)
apply (*subst cast-defl-principal*)
apply (*rule Abs-fin-defl-inverse*)
apply (*simp add*: *finite-deflation-bottom*)
done

lemma *cast-strict2* [*simp*]: *cast·A·*$\bot$ = $\bot$
by (*rule cast.below* [*THEN bottomI*])

26.4 Deflation combinators

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

definition
  defl-fun2 e p f =
    defl.extension (λ*a*.

```
    defl.extension (λb.
      defl-principal (Abs-fin-defl
        (e oo f·(Rep-fin-defl a)·(Rep-fin-defl b) oo p))))

lemma cast-defl-fun1:
  assumes ep: ep-pair e p
  assumes f: ⋀a. finite-deflation a ⟹ finite-deflation (f·a)
  shows cast·(defl-fun1 e p f·A) = e oo f·(cast·A) oo p
proof –
  have 1: ⋀a. finite-deflation (e oo f·(Rep-fin-defl a) oo p)
    apply (rule ep-pair.finite-deflation-e-d-p [OF ep])
    apply (rule f, rule finite-deflation-Rep-fin-defl)
    done
  show ?thesis
    by (induct A rule: defl.principal-induct, simp)
       (simp only: defl-fun1-def
                   defl.extension-principal
                   defl.extension-mono
                   defl.principal-mono
                   Abs-fin-defl-mono [OF 1 1]
                   monofun-cfun below-refl
                   Rep-fin-defl-mono
                   cast-defl-principal
                   Abs-fin-defl-inverse [unfolded mem-Collect-eq, OF 1])
qed

lemma cast-defl-fun2:
  assumes ep: ep-pair e p
  assumes f: ⋀a b. finite-deflation a ⟹ finite-deflation b ⟹
               finite-deflation (f·a·b)
  shows cast·(defl-fun2 e p f·A·B) = e oo f·(cast·A)·(cast·B) oo p
proof –
  have 1: ⋀a b. finite-deflation
      (e oo f·(Rep-fin-defl a)·(Rep-fin-defl b) oo p)
    apply (rule ep-pair.finite-deflation-e-d-p [OF ep])
    apply (rule f, (rule finite-deflation-Rep-fin-defl)+)
    done
  show ?thesis
    apply (induct A rule: defl.principal-induct, simp)
    apply (induct B rule: defl.principal-induct, simp)
    by (simp only: defl-fun2-def
                   defl.extension-principal
                   defl.extension-mono
                   defl.principal-mono
                   Abs-fin-defl-mono [OF 1 1]
                   monofun-cfun below-refl
                   Rep-fin-defl-mono
                   cast-defl-principal
                   Abs-fin-defl-inverse [unfolded mem-Collect-eq, OF 1])
```

qed

end

27 Representable domains

theory *Representable*
imports *Algebraic Map-Functions ~~/src/HOL/Library/Countable*
begin

default-sort *cpo*

27.1 Class of representable domains

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

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

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

class *predomain* = *predomain-syn* +
  assumes *predomain-ep*: *ep-pair liftemb liftprj*
  assumes *cast-liftdefl*: *cast·(liftdefl TYPE('a)) = liftemb oo liftprj*

syntax *-LIFTDEFL* :: *type* ⇒ *logic* ((1LIFTDEFL/(1'(-'))))
translations *LIFTDEFL('t)* ⇌ *CONST liftdefl TYPE('t)*

definition *liftdefl-of* :: *udom defl* → *udom u defl*
  where *liftdefl-of* = *defl-fun1 ID ID u-map*

lemma *cast-liftdefl-of*: *cast·(liftdefl-of·t) = u-map·(cast·t)*
by (*simp add*: *liftdefl-of-def cast-defl-fun1 ep-pair-def finite-deflation-u-map*)

class *domain* = *predomain-syn* + *pcpo* +
  fixes *emb* :: *'a* → *udom*
  fixes *prj* :: *udom* → *'a*
  fixes *defl* :: *'a itself* ⇒ *udom defl*
  assumes *ep-pair-emb-prj*: *ep-pair emb prj*
  assumes *cast-DEFL*: *cast·(defl TYPE('a)) = emb oo prj*
  assumes *liftemb-eq*: *liftemb = u-map·emb*
  assumes *liftprj-eq*: *liftprj = u-map·prj*
  assumes *liftdefl-eq*: *liftdefl TYPE('a) = liftdefl-of·(defl TYPE('a))*

syntax *-DEFL* :: *type* ⇒ *logic* ((1DEFL/(1'(-'))))
translations *DEFL('t)* ⇌ *CONST defl TYPE('t)*

instance *domain* ⊆ *predomain*
proof
  show *ep-pair liftemb* (*liftprj*::*udom*$_\bot$ → *'a*$_\bot$)
    unfolding *liftemb-eq liftprj-eq*
    by (*intro ep-pair-u-map ep-pair-emb-prj*)
  show *cast·LIFTDEFL*(*'a*) = *liftemb oo* (*liftprj*::*udom*$_\bot$ → *'a*$_\bot$)
    unfolding *liftemb-eq liftprj-eq liftdefl-eq*
    by (*simp add*: *cast-liftdefl-of cast-DEFL u-map-oo*)
qed

Constants *liftemb* and *liftprj* imply class predomain.

setup ‹
  fold Sign.add-const-constraint
  [(@{*const-name liftemb*}, *SOME* @{*typ 'a*::*predomain u* → *udom u*}),
   (@{*const-name liftprj*}, *SOME* @{*typ udom u* → *'a*::*predomain u*}),
   (@{*const-name liftdefl*}, *SOME* @{*typ 'a*::*predomain itself* ⇒ *udom u defl*})]
›

interpretation *predomain*: *pcpo-ep-pair liftemb liftprj*
  unfolding *pcpo-ep-pair-def* **by** (*rule predomain-ep*)

interpretation *domain*: *pcpo-ep-pair emb prj*
  unfolding *pcpo-ep-pair-def* **by** (*rule ep-pair-emb-prj*)

lemmas *emb-inverse* = *domain.e-inverse*
lemmas *emb-prj-below* = *domain.e-p-below*
lemmas *emb-eq-iff* = *domain.e-eq-iff*
lemmas *emb-strict* = *domain.e-strict*
lemmas *prj-strict* = *domain.p-strict*

27.2 Domains are bifinite

lemma *approx-chain-ep-cast*:
  assumes *ep*: *ep-pair* (*e*::*'a*::*pcpo* → *'b*::*bifinite*) (*p*::*'b* → *'a*)
  assumes *cast-t*: *cast·t* = *e oo p*
  shows ∃(*a*::*nat* ⇒ *'a*::*pcpo* → *'a*). *approx-chain a*
proof −
  interpret *ep-pair e p* **by** *fact*
  obtain *Y* **where** *Y*: ∀*i*. *Y i* ⊑ *Y* (*Suc i*)
  and *t*: *t* = (⨆*i*. *defl-principal* (*Y i*))
    by (*rule defl.obtain-principal-chain*)
  def *approx* ≡ λ*i*. (*p oo cast*·(*defl-principal* (*Y i*)) *oo e*) :: *'a* → *'a*
  have *approx-chain approx*
  proof (*rule approx-chain.intro*)
    show *chain* (λ*i*. *approx i*)
      unfolding *approx-def* **by** (*simp add*: *Y*)
    show (⨆*i*. *approx i*) = *ID*
      unfolding *approx-def*

by (*simp add*: *lub-distribs Y t* [*symmetric*] *cast-t cfun-eq-iff*)
show $\bigwedge i$. *finite-deflation* (*approx i*)
unfolding *approx-def*
apply (*rule finite-deflation-p-d-e*)
apply (*rule finite-deflation-cast*)
apply (*rule defl.compact-principal*)
apply (*rule below-trans* [*OF monofun-cfun-fun*])
apply (*rule is-ub-thelub*, *simp add*: *Y*)
apply (*simp add*: *lub-distribs Y t* [*symmetric*] *cast-t*)
done
qed
thus $\exists$(*a*::*nat* $\Rightarrow$ *'a* $\rightarrow$ *'a*). *approx-chain a* **by** − (*rule exI*)
qed

instance *domain* $\subseteq$ *bifinite*
by *standard* (*rule approx-chain-ep-cast* [*OF ep-pair-emb-prj cast-DEFL*])

instance *predomain* $\subseteq$ *profinite*
by *standard* (*rule approx-chain-ep-cast* [*OF predomain-ep cast-liftdefl*])

27.3 Universal domain ep-pairs

definition *u-emb* = *udom-emb* (λ*i*. *u-map*·(*udom-approx i*))
definition *u-prj* = *udom-prj* (λ*i*. *u-map*·(*udom-approx i*))

definition *prod-emb* = *udom-emb* (λ*i*. *prod-map*·(*udom-approx i*)·(*udom-approx i*))
definition *prod-prj* = *udom-prj* (λ*i*. *prod-map*·(*udom-approx i*)·(*udom-approx i*))

definition *sprod-emb* = *udom-emb* (λ*i*. *sprod-map*·(*udom-approx i*)·(*udom-approx i*))
definition *sprod-prj* = *udom-prj* (λ*i*. *sprod-map*·(*udom-approx i*)·(*udom-approx i*))

definition *ssum-emb* = *udom-emb* (λ*i*. *ssum-map*·(*udom-approx i*)·(*udom-approx i*))
definition *ssum-prj* = *udom-prj* (λ*i*. *ssum-map*·(*udom-approx i*)·(*udom-approx i*))

definition *sfun-emb* = *udom-emb* (λ*i*. *sfun-map*·(*udom-approx i*)·(*udom-approx i*))
definition *sfun-prj* = *udom-prj* (λ*i*. *sfun-map*·(*udom-approx i*)·(*udom-approx i*))

lemma *ep-pair-u*: *ep-pair u-emb u-prj*
unfolding *u-emb-def u-prj-def*
by (*simp add*: *ep-pair-udom approx-chain-u-map*)

lemma *ep-pair-prod*: *ep-pair prod-emb prod-prj*
unfolding *prod-emb-def prod-prj-def*

by (*simp add*: *ep-pair-udom approx-chain-prod-map*)

lemma *ep-pair-sprod*: *ep-pair sprod-emb sprod-prj*
unfolding *sprod-emb-def sprod-prj-def*
by (*simp add*: *ep-pair-udom approx-chain-sprod-map*)

lemma *ep-pair-ssum*: *ep-pair ssum-emb ssum-prj*
unfolding *ssum-emb-def ssum-prj-def*
by (*simp add*: *ep-pair-udom approx-chain-ssum-map*)

lemma *ep-pair-sfun*: *ep-pair sfun-emb sfun-prj*
unfolding *sfun-emb-def sfun-prj-def*
by (*simp add*: *ep-pair-udom approx-chain-sfun-map*)

27.4 Type combinators

definition *u-defl* :: *udom defl* → *udom defl*
where *u-defl* = *defl-fun1 u-emb u-prj u-map*

definition *prod-defl* :: *udom defl* → *udom defl* → *udom defl*
where *prod-defl* = *defl-fun2 prod-emb prod-prj prod-map*

definition *sprod-defl* :: *udom defl* → *udom defl* → *udom defl*
where *sprod-defl* = *defl-fun2 sprod-emb sprod-prj sprod-map*

definition *ssum-defl* :: *udom defl* → *udom defl* → *udom defl*
where *ssum-defl* = *defl-fun2 ssum-emb ssum-prj ssum-map*

definition *sfun-defl* :: *udom defl* → *udom defl* → *udom defl*
where *sfun-defl* = *defl-fun2 sfun-emb sfun-prj sfun-map*

lemma *cast-u-defl*:
cast·(*u-defl*·*A*) = *u-emb oo u-map*·(*cast*·*A*) *oo u-prj*
using *ep-pair-u finite-deflation-u-map*
unfolding *u-defl-def* **by** (*rule cast-defl-fun1*)

lemma *cast-prod-defl*:
cast·(*prod-defl*·*A*·*B*) =
prod-emb oo prod-map·(*cast*·*A*)·(*cast*·*B*) *oo prod-prj*
using *ep-pair-prod finite-deflation-prod-map*
unfolding *prod-defl-def* **by** (*rule cast-defl-fun2*)

lemma *cast-sprod-defl*:
cast·(*sprod-defl*·*A*·*B*) =
sprod-emb oo sprod-map·(*cast*·*A*)·(*cast*·*B*) *oo sprod-prj*
using *ep-pair-sprod finite-deflation-sprod-map*
unfolding *sprod-defl-def* **by** (*rule cast-defl-fun2*)

lemma *cast-ssum-defl*:

cast·(ssum-defl·A·B) =
ssum-emb oo ssum-map·(cast·A)·(cast·B) oo ssum-prj
using *ep-pair-ssum finite-deflation-ssum-map*
unfolding *ssum-defl-def* **by** (*rule cast-defl-fun2*)

lemma *cast-sfun-defl*:
cast·(sfun-defl·A·B) =
sfun-emb oo sfun-map·(cast·A)·(cast·B) oo sfun-prj
using *ep-pair-sfun finite-deflation-sfun-map*
unfolding *sfun-defl-def* **by** (*rule cast-defl-fun2*)

Special deflation combinator for unpointed types.

definition *u-liftdefl* :: *udom u defl* → *udom defl*
where *u-liftdefl* = *defl-fun1 u-emb u-prj ID*

lemma *cast-u-liftdefl*:
cast·(u-liftdefl·A) = u-emb oo cast·A oo u-prj
unfolding *u-liftdefl-def* **by** (*simp add*: *cast-defl-fun1 ep-pair-u*)

lemma *u-liftdefl-liftdefl-of*:
u-liftdefl·(liftdefl-of·A) = u-defl·A
by (*rule cast-eq-imp-eq*)
(*simp add*: *cast-u-liftdefl cast-liftdefl-of cast-u-defl*)

27.5 Class instance proofs

27.5.1 Universal domain

instantiation *udom* :: *domain*
begin

definition [*simp*]:
emb = (*ID* :: *udom* → *udom*)

definition [*simp*]:
prj = (*ID* :: *udom* → *udom*)

definition
defl (*t*::*udom itself*) = (⨆ *i. defl-principal* (*Abs-fin-defl* (*udom-approx i*)))

definition
(*liftemb* :: *udom u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → *udom u*) = *u-map·prj*

definition
liftdefl (*t*::*udom itself*) = *liftdefl-of·DEFL*(*udom*)

instance proof

show *ep-pair emb* (*prj* :: *udom* → *udom*)
by (*simp add*: *ep-pair.intro*)
show *cast·DEFL*(*udom*) = *emb oo* (*prj* :: *udom* → *udom*)
unfolding *defl-udom-def*
apply (*subst contlub-cfun-arg*)
apply (*rule chainI*)
apply (*rule defl.principal-mono*)
apply (*simp add*: *below-fin-defl-def*)
apply (*simp add*: *Abs-fin-defl-inverse finite-deflation-udom-approx*)
apply (*rule chainE*)
apply (*rule chain-udom-approx*)
apply (*subst cast-defl-principal*)
apply (*simp add*: *Abs-fin-defl-inverse finite-deflation-udom-approx*)
done
qed (*fact liftemb-udom-def liftprj-udom-def liftdefl-udom-def*)+

end

27.5.2 Lifted cpo

instantiation *u* :: (*predomain*) *domain*
begin

definition
emb = *u-emb oo liftemb*

definition
prj = *liftprj oo u-prj*

definition
defl (*t*::'*a u itself*) = *u-liftdefl·LIFTDEFL*('*a*)

definition
(*liftemb* :: '*a u u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → '*a u u*) = *u-map·prj*

definition
liftdefl (*t*::'*a u itself*) = *liftdefl-of·DEFL*('*a u*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → '*a u*)
unfolding *emb-u-def prj-u-def*
by (*intro ep-pair-comp ep-pair-u predomain-ep*)
show *cast·DEFL*('*a u*) = *emb oo* (*prj* :: *udom* → '*a u*)
unfolding *emb-u-def prj-u-def defl-u-def*
by (*simp add*: *cast-u-liftdefl cast-liftdefl assoc-oo*)
qed (*fact liftemb-u-def liftprj-u-def liftdefl-u-def*)+

end

lemma *DEFL-u*: *DEFL('a::predomain u) = u-liftdefl·LIFTDEFL('a)*
by (*rule defl-u-def*)

27.5.3 Strict function space

instantiation *sfun* :: (*domain*, *domain*) *domain*
begin

definition
emb = sfun-emb oo sfun-map·prj·emb

definition
prj = sfun-map·emb·prj oo sfun-prj

definition
defl (t::('a →! 'b) itself) = sfun-defl·DEFL('a)·DEFL('b)

definition
(*liftemb* :: (*'a →! 'b*) *u → udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u →* (*'a →! 'b*) *u*) = *u-map·prj*

definition
liftdefl (t::('a →! 'b) itself) = liftdefl-of·DEFL('a →! 'b)

instance proof
show *ep-pair emb* (*prj* :: *udom → 'a →! 'b*)
unfolding *emb-sfun-def prj-sfun-def*
by (*intro ep-pair-comp ep-pair-sfun ep-pair-sfun-map ep-pair-emb-prj*)
show *cast·DEFL('a →! 'b) = emb oo* (*prj* :: *udom → 'a →! 'b*)
unfolding *emb-sfun-def prj-sfun-def defl-sfun-def cast-sfun-defl*
by (*simp add*: *cast-DEFL oo-def sfun-eq-iff sfun-map-map*)
qed (*fact liftemb-sfun-def liftprj-sfun-def liftdefl-sfun-def*)+

end

lemma *DEFL-sfun*:
DEFL('a::domain →! 'b::domain) = sfun-defl·DEFL('a)·DEFL('b)
by (*rule defl-sfun-def*)

27.5.4 Continuous function space

instantiation *cfun* :: (*predomain*, *domain*) *domain*
begin

definition

emb = emb oo encode-cfun

definition
prj = decode-cfun oo prj

definition
defl (t::('a → 'b) itself) = DEFL('a u →! 'b)

definition
(liftemb :: ('a → 'b) u → udom u) = u-map·emb

definition
(liftprj :: udom u → ('a → 'b) u) = u-map·prj

definition
liftdefl (t::('a → 'b) itself) = liftdefl-of·DEFL('a → 'b)

instance proof
have *ep-pair encode-cfun decode-cfun*
by *(rule ep-pair.intro, simp-all)*
thus *ep-pair emb (prj :: udom → 'a → 'b)*
unfolding *emb-cfun-def prj-cfun-def*
using *ep-pair-emb-prj* **by** *(rule ep-pair-comp)*
show *cast·DEFL('a → 'b) = emb oo (prj :: udom → 'a → 'b)*
unfolding *emb-cfun-def prj-cfun-def defl-cfun-def*
by *(simp add: cast-DEFL cfcomp1)*
qed *(fact liftemb-cfun-def liftprj-cfun-def liftdefl-cfun-def)+*

end

lemma *DEFL-cfun*:
DEFL('a::predomain → 'b::domain) = DEFL('a u →! 'b)
by *(rule defl-cfun-def)*

27.5.5 Strict product

instantiation *sprod :: (domain, domain) domain*
begin

definition
emb = sprod-emb oo sprod-map·emb·emb

definition
prj = sprod-map·prj·prj oo sprod-prj

definition
defl (t::('a ⊗ 'b) itself) = sprod-defl·DEFL('a)·DEFL('b)

definition

(*liftemb* :: ('*a* ⊗ '*b*) *u* → *udom u*) = *u-map*·*emb*

definition
(*liftprj* :: *udom u* → ('*a* ⊗ '*b*) *u*) = *u-map*·*prj*

definition
liftdefl (*t*::('*a* ⊗ '*b*) *itself*) = *liftdefl-of*·*DEFL*('*a* ⊗ '*b*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → '*a* ⊗ '*b*)
unfolding *emb-sprod-def prj-sprod-def*
by (*intro ep-pair-comp ep-pair-sprod ep-pair-sprod-map ep-pair-emb-prj*)
show *cast*·*DEFL*('*a* ⊗ '*b*) = *emb oo* (*prj* :: *udom* → '*a* ⊗ '*b*)
unfolding *emb-sprod-def prj-sprod-def defl-sprod-def cast-sprod-defl*
by (*simp add*: *cast-DEFL oo-def cfun-eq-iff sprod-map-map*)
qed (*fact liftemb-sprod-def liftprj-sprod-def liftdefl-sprod-def*)+

end

lemma *DEFL-sprod*:
DEFL('*a*::*domain* ⊗ '*b*::*domain*) = *sprod-defl*·*DEFL*('*a*)·*DEFL*('*b*)
by (*rule defl-sprod-def*)

27.5.6 Cartesian product

definition *prod-liftdefl* :: *udom u defl* → *udom u defl* → *udom u defl*
where *prod-liftdefl* = *defl-fun2* (*u-map*·*prod-emb oo decode-prod-u*)
(*encode-prod-u oo u-map*·*prod-prj*) *sprod-map*

lemma *cast-prod-liftdefl*:
cast·(*prod-liftdefl*·*a*·*b*) =
(*u-map*·*prod-emb oo decode-prod-u*) *oo sprod-map*·(*cast*·*a*)·(*cast*·*b*) *oo*
(*encode-prod-u oo u-map*·*prod-prj*)
unfolding *prod-liftdefl-def*
apply (*rule cast-defl-fun2*)
apply (*intro ep-pair-comp ep-pair-u-map ep-pair-prod*)
apply (*simp add*: *ep-pair.intro*)
apply (*erule* (*1*) *finite-deflation-sprod-map*)
done

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

definition
liftemb = (*u-map*·*prod-emb oo decode-prod-u*) *oo*
(*sprod-map*·*liftemb*·*liftemb oo encode-prod-u*)

definition
liftprj = (*decode-prod-u oo sprod-map*·*liftprj*·*liftprj*) *oo*

(*encode-prod-u oo u-map·prod-prj*)

definition
liftdefl (*t*::(′*a* × ′*b*) *itself*) = *prod-liftdefl·LIFTDEFL*(′*a*)·*LIFTDEFL*(′*b*)

instance proof
show *ep-pair liftemb* (*liftprj* :: *udom u* → (′*a* × ′*b*) *u*)
unfolding *liftemb-prod-def liftprj-prod-def*
by (*intro ep-pair-comp ep-pair-sprod-map ep-pair-u-map ep-pair-prod predomain-ep*, *simp-all add*: *ep-pair.intro*)
show *cast·LIFTDEFL*(′*a* × ′*b*) = *liftemb oo* (*liftprj* :: *udom u* → (′*a* × ′*b*) *u*)
unfolding *liftemb-prod-def liftprj-prod-def liftdefl-prod-def*
by (*simp add*: *cast-prod-liftdefl cast-liftdefl cfcomp1 sprod-map-map*)
qed

end

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

definition
emb = *prod-emb oo prod-map·emb·emb*

definition
prj = *prod-map·prj·prj oo prod-prj*

definition
defl (*t*::(′*a* × ′*b*) *itself*) = *prod-defl·DEFL*(′*a*)·*DEFL*(′*b*)

instance proof
show *1*: *ep-pair emb* (*prj* :: *udom* → ′*a* × ′*b*)
unfolding *emb-prod-def prj-prod-def*
by (*intro ep-pair-comp ep-pair-prod ep-pair-prod-map ep-pair-emb-prj*)
show *2*: *cast·DEFL*(′*a* × ′*b*) = *emb oo* (*prj* :: *udom* → ′*a* × ′*b*)
unfolding *emb-prod-def prj-prod-def defl-prod-def cast-prod-defl*
by (*simp add*: *cast-DEFL oo-def cfun-eq-iff prod-map-map*)
show *3*: *liftemb* = *u-map*·(*emb* :: ′*a* × ′*b* → *udom*)
unfolding *emb-prod-def liftemb-prod-def liftemb-eq*
unfolding *encode-prod-u-def decode-prod-u-def*
by (*rule cfun-eqI*, *case-tac x*, *simp*, *clarsimp*)
show *4*: *liftprj* = *u-map*·(*prj* :: *udom* → ′*a* × ′*b*)
unfolding *prj-prod-def liftprj-prod-def liftprj-eq*
unfolding *encode-prod-u-def decode-prod-u-def*
apply (*rule cfun-eqI*, *case-tac x*, *simp*)
apply (*rename-tac y*, *case-tac prod-prj·y*, *simp*)
done
show *5*: *LIFTDEFL*(′*a* × ′*b*) = *liftdefl-of·DEFL*(′*a* × ′*b*)
by (*rule cast-eq-imp-eq*)
(*simp add*: *cast-liftdefl cast-liftdefl-of cast-DEFL 2 3 4 u-map-oo*)

qed

end

lemma *DEFL-prod*:
  DEFL('a::domain × 'b::domain) = prod-defl·DEFL('a)·DEFL('b)
by (*rule defl-prod-def*)

lemma *LIFTDEFL-prod*:
  LIFTDEFL('a::predomain × 'b::predomain) =
    prod-liftdefl·LIFTDEFL('a)·LIFTDEFL('b)
by (*rule liftdefl-prod-def*)

27.5.7 Unit type

instantiation *unit* :: *domain*
begin

definition
  emb = (⊥ :: *unit* → *udom*)

definition
  prj = (⊥ :: *udom* → *unit*)

definition
  defl (*t::unit itself*) = ⊥

definition
  (*liftemb* :: *unit u* → *udom u*) = *u-map·emb*

definition
  (*liftprj* :: *udom u* → *unit u*) = *u-map·prj*

definition
  liftdefl (*t::unit itself*) = *liftdefl-of·DEFL(unit)*

instance proof
  show *ep-pair emb* (*prj* :: *udom* → *unit*)
    unfolding *emb-unit-def prj-unit-def*
    by (*simp add*: *ep-pair.intro*)
  show *cast·DEFL(unit)* = *emb oo* (*prj* :: *udom* → *unit*)
    unfolding *emb-unit-def prj-unit-def defl-unit-def* **by** *simp*
qed (*fact liftemb-unit-def liftprj-unit-def liftdefl-unit-def*)+

end

27.5.8 Discrete cpo

instantiation *discr* :: (*countable*) *predomain*
begin

definition
(*liftemb* :: *'a discr u* → *udom u*) = *strictify*·*up oo udom-emb discr-approx*

definition
(*liftprj* :: *udom u* → *'a discr u*) = *udom-prj discr-approx oo fup*·*ID*

definition
liftdefl (*t*::*'a discr itself*) =
(⨆ *i*. *defl-principal* (*Abs-fin-defl* (*liftemb oo discr-approx i oo* (*liftprj*::*udom u* → *'a discr u*))))

instance proof
show *1*: *ep-pair liftemb* (*liftprj* :: *udom u* → *'a discr u*)
unfolding *liftemb-discr-def liftprj-discr-def*
apply (*intro ep-pair-comp ep-pair-udom* [*OF discr-approx*])
apply (*rule ep-pair.intro*)
apply (*simp add*: *strictify-conv-if*)
apply (*case-tac y*, *simp*, *simp add*: *strictify-conv-if*)
done
show *cast*·*LIFTDEFL*(*'a discr*) = *liftemb oo* (*liftprj* :: *udom u* → *'a discr u*)
unfolding *liftdefl-discr-def*
apply (*subst contlub-cfun-arg*)
apply (*rule chainI*)
apply (*rule defl.principal-mono*)
apply (*simp add*: *below-fin-defl-def*)
apply (*simp add*: *Abs-fin-defl-inverse*
ep-pair.finite-deflation-e-d-p [*OF 1*]
approx-chain.finite-deflation-approx [*OF discr-approx*])
apply (*intro monofun-cfun below-refl*)
apply (*rule chainE*)
apply (*rule chain-discr-approx*)
apply (*subst cast-defl-principal*)
apply (*simp add*: *Abs-fin-defl-inverse*
ep-pair.finite-deflation-e-d-p [*OF 1*]
approx-chain.finite-deflation-approx [*OF discr-approx*])
apply (*simp add*: *lub-distribs*)
done
qed

end

27.5.9 Strict sum

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

definition
emb = *ssum-emb oo ssum-map*·*emb*·*emb*

definition
prj = ssum-map·prj·prj oo ssum-prj

definition
defl (t::('a ⊕ 'b) itself) = ssum-defl·DEFL('a)·DEFL('b)

definition
(liftemb :: ('a ⊕ 'b) u → udom u) = u-map·emb

definition
(liftprj :: udom u → ('a ⊕ 'b) u) = u-map·prj

definition
liftdefl (t::('a ⊕ 'b) itself) = liftdefl-of·DEFL('a ⊕ 'b)

instance proof
  show *ep-pair emb (prj :: udom → 'a ⊕ 'b)*
    unfolding *emb-ssum-def prj-ssum-def*
    by *(intro ep-pair-comp ep-pair-ssum ep-pair-ssum-map ep-pair-emb-prj)*
  show *cast·DEFL('a ⊕ 'b) = emb oo (prj :: udom → 'a ⊕ 'b)*
    unfolding *emb-ssum-def prj-ssum-def defl-ssum-def cast-ssum-defl*
    by *(simp add: cast-DEFL oo-def cfun-eq-iff ssum-map-map)*
qed *(fact liftemb-ssum-def liftprj-ssum-def liftdefl-ssum-def)+*

end

lemma *DEFL-ssum*:
DEFL('a::domain ⊕ 'b::domain) = ssum-defl·DEFL('a)·DEFL('b)
by *(rule defl-ssum-def)*

27.5.10 Lifted HOL type

instantiation *lift :: (countable) domain*
begin

definition
emb = emb oo (Λ x. Rep-lift x)

definition
prj = (Λ y. Abs-lift y) oo prj

definition
defl (t::'a lift itself) = DEFL('a discr u)

definition
(liftemb :: 'a lift u → udom u) = u-map·emb

definition

(*liftprj* :: *udom u* → *'a lift u*) = *u-map*·*prj*

definition
liftdefl (*t*::*'a lift itself*) = *liftdefl-of*·*DEFL*(*'a lift*)

instance proof
note [*simp*] = *cont-Rep-lift cont-Abs-lift Rep-lift-inverse Abs-lift-inverse*
have *ep-pair* (Λ(*x*::*'a lift*). *Rep-lift x*) (Λ *y*. *Abs-lift y*)
by (*simp add*: *ep-pair-def*)
thus *ep-pair emb* (*prj* :: *udom* → *'a lift*)
unfolding *emb-lift-def prj-lift-def*
using *ep-pair-emb-prj* **by** (*rule ep-pair-comp*)
show *cast*·*DEFL*(*'a lift*) = *emb oo* (*prj* :: *udom* → *'a lift*)
unfolding *emb-lift-def prj-lift-def defl-lift-def cast-DEFL*
by (*simp add*: *cfcomp1*)
qed (*fact liftemb-lift-def liftprj-lift-def liftdefl-lift-def*)+

end

end

28 Domain package support

theory *Domain-Aux*
imports *Map-Functions Fixrec*
begin

28.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
fixes *abs* :: *'a* → *'b*
fixes *rep* :: *'b* → *'a*
assumes *abs-iso* [*simp*]: *rep*·(*abs*·*x*) = *x*
assumes *rep-iso* [*simp*]: *abs*·(*rep*·*y*) = *y*
begin

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

lemma *abs-below*: (*abs*·*x* ⊑ *abs*·*y*) = (*x* ⊑ *y*)
proof
assume *abs*·*x* ⊑ *abs*·*y*
then have *rep*·(*abs*·*x*) ⊑ *rep*·(*abs*·*y*) **by** (*rule monofun-cfun-arg*)
then show *x* ⊑ *y* **by** *simp*
next
assume *x* ⊑ *y*
then show *abs*·*x* ⊑ *abs*·*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 bottomI*)
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-bottom-iff*: $(abs{\cdot}x = \bot) = (x = \bot)$
by (*auto elim*: *abs-defin' intro*: *abs-strict*)

lemma *rep-bottom-iff*: $(rep{\cdot}x = \bot) = (x = \bot)$
by (*rule iso.abs-bottom-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-bottom-iff*)

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

proof (*unfold compact-def*)
  assume *adm* ($\lambda y.$ *abs*$\cdot x \not\sqsubseteq y$)
  with *cont-Rep-cfun2*
  have *adm* ($\lambda y.$ *abs*$\cdot x \not\sqsubseteq$ *abs*$\cdot y$) **by** (*rule adm-subst*)
  then show *adm* ($\lambda y.$ $x \not\sqsubseteq y$) **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

28.2 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

28.3 Finiteness

Let a "decisive" function be a deflation that maps every input to either itself or bottom. Then if a domain's take functions are all decisive, then all values in the domain are finite.

definition
decisive :: $('a::pcpo \rightarrow 'a) \Rightarrow bool$
where
decisive $d \longleftrightarrow (\forall x.\ d{\cdot}x = x \lor d{\cdot}x = \bot)$

lemma *decisiveI*: $(\bigwedge x.\ d{\cdot}x = x \lor d{\cdot}x = \bot) \Longrightarrow$ *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*

28.4 Proofs about constructor functions

Lemmas for proving nchotomy rule:

lemma *ex-one-bottom-iff*:
($\exists x.\ P\ x \wedge x \neq \bot$) = *P ONE*
by *simp*

lemma *ex-up-bottom-iff*:
($\exists x.\ P\ x \wedge x \neq \bot$) = ($\exists x.\ P\ (up\cdot x)$)
by (*safe*, *case-tac x*, *auto*)

lemma *ex-sprod-bottom-iff*:
($\exists y.\ P\ y \wedge y \neq \bot$) =
($\exists x\ y.\ (P\ (:x, y:) \wedge x \neq \bot) \wedge y \neq \bot$)
by (*safe*, *case-tac y*, *auto*)

lemma *ex-sprod-up-bottom-iff*:
($\exists y.\ P\ y \wedge y \neq \bot$) =
($\exists x\ y.\ P\ (:up\cdot x, y:) \wedge y \neq \bot$)
by (*safe*, *case-tac y*, *simp*, *case-tac x*, *auto*)

lemma *ex-ssum-bottom-iff*:
($\exists x.\ P\ x \wedge x \neq \bot$) =
(($\exists x.\ P\ (sinl\cdot x) \wedge x \neq \bot$) $\vee$
($\exists x.\ P\ (sinr\cdot x) \wedge x \neq \bot$))
by (*safe*, *case-tac x*, *auto*)

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

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

Rules for turning nchotomy into exhaust:

lemma *exh-casedist0*: $[\![R;\ R \Longrightarrow P]\!] \Longrightarrow P$
by *auto*

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

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

Rules for proving constructor properties

lemmas *con-strict-rules* =
sinl-strict sinr-strict spair-strict1 spair-strict2

lemmas *con-bottom-iff-rules* =
sinl-bottom-iff sinr-bottom-iff spair-bottom-iff up-defined ONE-defined

lemmas *con-below-iff-rules* =
sinl-below sinr-below sinl-below-sinr sinr-below-sinl con-bottom-iff-rules

lemmas *con-eq-iff-rules* =
sinl-eq sinr-eq sinl-eq-sinr sinr-eq-sinl con-bottom-iff-rules

lemmas *sel-strict-rules* =
cfcomp2 sscase1 sfst-strict ssnd-strict fup1

lemma *sel-app-extra-rules*:
$sscase{\cdot}ID{\cdot}\bot{\cdot}(sinr{\cdot}x) = \bot$
$sscase{\cdot}ID{\cdot}\bot{\cdot}(sinl{\cdot}x) = x$
$sscase{\cdot}\bot{\cdot}ID{\cdot}(sinl{\cdot}x) = \bot$
$sscase{\cdot}\bot{\cdot}ID{\cdot}(sinr{\cdot}x) = x$
$fup{\cdot}ID{\cdot}(up{\cdot}x) = x$
by (*cases* $x = \bot$, *simp*, *simp*)+

lemmas *sel-app-rules* =
sel-strict-rules sel-app-extra-rules

ssnd-spair sfst-spair up-defined spair-defined

lemmas *sel-bottom-iff-rules* =
cfcomp2 sfst-bottom-iff ssnd-bottom-iff

lemmas *take-con-rules* =
ssum-map-sinl′ ssum-map-sinr′ sprod-map-spair′ u-map-up
deflation-strict deflation-ID ID1 cfcomp2

28.5 ML setup

named-theorems *domain-deflation theorems like deflation a ==> deflation (foo-map$a)*
and *domain-map-ID theorems like foo-map$ID = ID*

ML-file *Tools/Domain/domain-take-proofs.ML*
ML-file *Tools/cont-consts.ML*
ML-file *Tools/cont-proc.ML*
ML-file *Tools/Domain/domain-constructors.ML*
ML-file *Tools/Domain/domain-induction.ML*

end

29 Domain package

theory *Domain*
imports *Representable Domain-Aux*
keywords
domaindef :: *thy-decl* **and** *lazy unsafe* **and**
domain-isomorphism domain :: *thy-decl*
begin

default-sort *domain*

29.1 Representations of types

lemma *emb-prj*: $emb \cdot ((prj \cdot x)::'a) = cast \cdot DEFL('a) \cdot x$
by (*simp add*: *cast-DEFL*)

lemma *emb-prj-emb*:
fixes $x :: 'a$
assumes $DEFL('a) \sqsubseteq DEFL('b)$
shows $emb \cdot (prj \cdot (emb \cdot x) :: 'b) = emb \cdot x$
unfolding *emb-prj*
apply (*rule cast.belowD*)
apply (*rule monofun-cfun-arg* [*OF assms*])
apply (*simp add*: *cast-DEFL*)
done

lemma *prj-emb-prj*:

assumes *DEFL('a)* ⊑ *DEFL('b)*
shows *prj·(emb·(prj·x :: 'b))* = *(prj·x :: 'a)*
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 (*rule monofun-cfun-arg* [*OF assms*])
done

Isomorphism lemmas used internally by the domain package:

lemma *domain-abs-iso*:
fixes *abs* **and** *rep*
assumes *DEFL*: *DEFL('b)* = *DEFL('a)*
assumes *abs-def*: (*abs* :: *'a* → *'b*) ≡ *prj oo emb*
assumes *rep-def*: (*rep* :: *'b* → *'a*) ≡ *prj oo emb*
shows *rep·(abs·x)* = *x*
unfolding *abs-def rep-def*
by (*simp add*: *emb-prj-emb DEFL*)

lemma *domain-rep-iso*:
fixes *abs* **and** *rep*
assumes *DEFL*: *DEFL('b)* = *DEFL('a)*
assumes *abs-def*: (*abs* :: *'a* → *'b*) ≡ *prj oo emb*
assumes *rep-def*: (*rep* :: *'b* → *'a*) ≡ *prj oo emb*
shows *abs·(rep·x)* = *x*
unfolding *abs-def rep-def*
by (*simp add*: *emb-prj-emb DEFL*)

29.2 Deflations as sets

definition *defl-set* :: *'a::bifinite defl* ⇒ *'a set*
where *defl-set A* = {*x. cast·A·x* = *x*}

lemma *adm-defl-set*: *adm* (λ*x. x* ∈ *defl-set A*)
unfolding *defl-set-def* **by** *simp*

lemma *defl-set-bottom*: ⊥ ∈ *defl-set A*
unfolding *defl-set-def* **by** *simp*

lemma *defl-set-cast* [*simp*]: *cast·A·x* ∈ *defl-set A*
unfolding *defl-set-def* **by** *simp*

lemma *defl-set-subset-iff*: *defl-set A* ⊆ *defl-set B* ⟷ *A* ⊑ *B*
apply (*simp add*: *defl-set-def subset-eq cast-below-cast* [*symmetric*])
apply (*auto simp add*: *cast.belowI cast.belowD*)
done

29.3 Proving a subtype is representable

Temporarily relax type constraints.

```
setup ‹
  fold Sign.add-const-constraint
  [ (@{const-name defl}, SOME @{typ 'a::pcpo itself ⇒ udom defl})
  , (@{const-name emb}, SOME @{typ 'a::pcpo → udom})
  , (@{const-name prj}, SOME @{typ udom → 'a::pcpo})
  , (@{const-name liftdefl}, SOME @{typ 'a::pcpo itself ⇒ udom u defl})
  , (@{const-name liftemb}, SOME @{typ 'a::pcpo u → udom u})
  , (@{const-name liftprj}, SOME @{typ udom u → 'a::pcpo u}) ]
›

lemma typedef-domain-class:
  fixes Rep :: 'a::pcpo ⇒ udom
  fixes Abs :: udom ⇒ 'a::pcpo
  fixes t :: udom defl
  assumes type: type-definition Rep Abs (defl-set 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))
  assumes defl: defl ≡ (λ a::'a itself. t)
  assumes liftemb: (liftemb :: 'a u → udom u) ≡ u-map·emb
  assumes liftprj: (liftprj :: udom u → 'a u) ≡ u-map·prj
  assumes liftdefl: (liftdefl :: 'a itself ⇒ -) ≡ (λt. liftdefl-of·DEFL('a))
  shows OFCLASS('a, domain-class)
proof
  have emb-beta: ⋀x. emb·x = Rep x
    unfolding emb
    apply (rule beta-cfun)
    apply (rule typedef-cont-Rep [OF type below adm-defl-set cont-id])
    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-defl-set])
    apply simp-all
    done
  have prj-emb: ⋀x::'a. prj·(emb·x) = x
    using type-definition.Rep [OF type]
    unfolding prj-beta emb-beta defl-set-def
    by (simp add: type-definition.Rep-inverse [OF type])
  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 ep-pair (emb :: 'a → udom) prj
    apply standard
    apply (simp add: prj-emb)
    apply (simp add: emb-prj cast.below)
```

```
    done
  show cast·DEFL('a) = emb oo (prj :: udom → 'a)
    by (rule cfun-eqI, simp add: defl emb-prj)
qed (simp-all only: liftemb liftprj liftdefl)

lemma typedef-DEFL:
  assumes defl ≡ (λa::'a::pcpo itself. t)
  shows DEFL('a::pcpo) = t
unfolding assms ..
```

Restore original typing constraints.

```
setup ‹
  fold Sign.add-const-constraint
   [(@{const-name defl}, SOME @{typ 'a::domain itself ⇒ udom defl}),
    (@{const-name emb}, SOME @{typ 'a::domain → udom}),
    (@{const-name prj}, SOME @{typ udom → 'a::domain}),
    (@{const-name liftdefl}, SOME @{typ 'a::predomain itself ⇒ udom u defl}),
    (@{const-name liftemb}, SOME @{typ 'a::predomain u → udom u}),
    (@{const-name liftprj}, SOME @{typ udom u → 'a::predomain u})]
›

ML-file Tools/domaindef.ML
```

29.4 Isomorphic deflations

```
definition isodefl :: ('a → 'a) ⇒ udom defl ⇒ bool
  where isodefl d t ⟷ cast·t = emb oo d oo prj

definition isodefl' :: ('a::predomain → 'a) ⇒ udom u defl ⇒ bool
  where isodefl' d t ⟷ cast·t = liftemb oo u-map·d oo liftprj

lemma isodeflI: (⋀x. cast·t·x = emb·(d·(prj·x))) ⟹ isodefl d t
unfolding isodefl-def by (simp add: cfun-eqI)

lemma cast-isodefl: isodefl d t ⟹ cast·t = (Λ x. emb·(d·(prj·x)))
unfolding isodefl-def by (simp add: cfun-eqI)

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 → 'a
  assumes isodefl d t shows deflation d
proof
  note assms [unfolded isodefl-def, simp]
  fix x :: 'a
  show d·(d·x) = d·x
    using cast.idem [of t emb·x] by simp
```

show $d \cdot x \sqsubseteq x$
using *cast.below* [*of t emb*$\cdot$*x*] **by** *simp*
qed

lemma *isodefl-ID-DEFL*: *isodefl* (*ID* :: $'a \rightarrow 'a$) *DEFL*($'a$)
unfolding *isodefl-def* **by** (*simp add*: *cast-DEFL*)

lemma *isodefl-LIFTDEFL*:
isodefl' (*ID* :: $'a \rightarrow 'a$) *LIFTDEFL*($'a$::*predomain*)
unfolding *isodefl'-def* **by** (*simp add*: *cast-liftdefl u-map-ID*)

lemma *isodefl-DEFL-imp-ID*: *isodefl* (d :: $'a \rightarrow 'a$) *DEFL*($'a$) $\Longrightarrow d = ID$
unfolding *isodefl-def*
apply (*simp add*: *cast-DEFL*)
apply (*simp add*: *cfun-eq-iff*)
apply (*rule allI*)
apply (*drule-tac x*=*emb*$\cdot$*x* **in** *spec*)
apply *simp*
done

lemma *isodefl-bottom*: *isodefl* $\bot$ $\bot$
unfolding *isodefl-def* **by** (*simp add*: *cfun-eq-iff*)

lemma *adm-isodefl*:
cont f $\Longrightarrow$ *cont g* $\Longrightarrow$ *adm* (λx. *isodefl* (*f x*) (*g x*))
unfolding *isodefl-def* **by** *simp*

lemma *isodefl-lub*:
assumes *chain d* **and** *chain t*
assumes $\bigwedge i$. *isodefl* (*d i*) (*t i*)
shows *isodefl* ($\bigsqcup i$. *d i*) ($\bigsqcup i$. *t i*)
using *assms* **unfolding** *isodefl-def*
by (*simp add*: *contlub-cfun-arg contlub-cfun-fun*)

lemma *isodefl-fix*:
assumes $\bigwedge d\ t$. *isodefl d t* $\Longrightarrow$ *isodefl* ($f \cdot d$) ($g \cdot t$)
shows *isodefl* (*fix*$\cdot f$) (*fix*$\cdot g$)
unfolding *fix-def2*
apply (*rule isodefl-lub*, *simp*, *simp*)
apply (*induct-tac i*)
apply (*simp add*: *isodefl-bottom*)
apply (*simp add*: *assms*)
done

lemma *isodefl-abs-rep*:
fixes *abs* **and** *rep* **and** *d*
assumes *DEFL*: *DEFL*($'b$) = *DEFL*($'a$)
assumes *abs-def*: (*abs* :: $'a \rightarrow 'b$) $\equiv$ *prj oo emb*
assumes *rep-def*: (*rep* :: $'b \rightarrow 'a$) $\equiv$ *prj oo emb*

shows *isodefl d t* ⟹ *isodefl* (*abs oo d oo rep*) *t*
unfolding *isodefl-def*
by (*simp add*: *cfun-eq-iff assms prj-emb-prj emb-prj-emb*)

lemma *isodefl'-liftdefl-of*: *isodefl d t* ⟹ *isodefl' d* (*liftdefl-of·t*)
unfolding *isodefl-def isodefl'-def*
by (*simp add*: *cast-liftdefl-of u-map-oo liftemb-eq liftprj-eq*)

lemma *isodefl-sfun*:
isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
isodefl (*sfun-map·d1·d2*) (*sfun-defl·t1·t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-sfun-defl cast-isodefl*)
apply (*simp add*: *emb-sfun-def prj-sfun-def*)
apply (*simp add*: *sfun-map-map isodefl-strict*)
done

lemma *isodefl-ssum*:
isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
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 ⟹ *isodefl d2 t2* ⟹
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-prod*:
isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
isodefl (*prod-map·d1·d2*) (*prod-defl·t1·t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-prod-defl cast-isodefl*)
apply (*simp add*: *emb-prod-def prj-prod-def*)
apply (*simp add*: *prod-map-map cfcomp1*)
done

lemma *isodefl-u*:
isodefl d t ⟹ *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 liftemb-eq liftprj-eq u-map-map*)

done

lemma *isodefl-u-liftdefl*:
  isodefl′ d t ⟹ *isodefl* (*u-map·d*) (*u-liftdefl·t*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-u-liftdefl isodefl′-def*)
apply (*simp add*: *emb-u-def prj-u-def liftemb-eq liftprj-eq*)
done

lemma *encode-prod-u-map*:
  encode-prod-u·(*u-map*·(*prod-map*·*f*·*g*)·(*decode-prod-u*·*x*))
    = *sprod-map*·(*u-map*·*f*)·(*u-map*·*g*)·*x*
unfolding *encode-prod-u-def decode-prod-u-def*
apply (*case-tac x*, *simp*, *rename-tac a b*)
apply (*case-tac a*, *simp*, *case-tac b*, *simp*, *simp*)
done

lemma *isodefl-prod-u*:
  assumes *isodefl′ d1 t1* **and** *isodefl′ d2 t2*
  shows *isodefl′* (*prod-map*·*d1*·*d2*) (*prod-liftdefl*·*t1*·*t2*)
using *assms* **unfolding** *isodefl′-def*
unfolding *liftemb-prod-def liftprj-prod-def*
by (*simp add*: *cast-prod-liftdefl cfcomp1 encode-prod-u-map sprod-map-map*)

lemma *encode-cfun-map*:
  encode-cfun·(*cfun-map*·*f*·*g*·(*decode-cfun*·*x*))
    = *sfun-map*·(*u-map*·*f*)·*g*·*x*
unfolding *encode-cfun-def decode-cfun-def*
apply (*simp add*: *sfun-eq-iff cfun-map-def sfun-map-def*)
apply (*rule cfun-eqI*, *rename-tac y*, *case-tac y*, *simp-all*)
done

lemma *isodefl-cfun*:
  assumes *isodefl* (*u-map*·*d1*) *t1* **and** *isodefl d2 t2*
  shows *isodefl* (*cfun-map*·*d1*·*d2*) (*sfun-defl*·*t1*·*t2*)
using *isodefl-sfun* [*OF assms*] **unfolding** *isodefl-def*
by (*simp add*: *emb-cfun-def prj-cfun-def cfcomp1 encode-cfun-map*)

29.5 Setting up the domain package

named-theorems *domain-defl-simps theorems like DEFL*(*′a t*) = *t-defl*$*DEFL*(*′a*)
 and *domain-isodefl theorems like isodefl d t* ==> *isodefl* (*foo-map*$*d*) (*foo-defl*$*t*)

ML-file *Tools/Domain/domain-isomorphism.ML*
ML-file *Tools/Domain/domain-axioms.ML*
ML-file *Tools/Domain/domain.ML*

lemmas [*domain-defl-simps*] =
  DEFL-cfun DEFL-sfun DEFL-ssum DEFL-sprod DEFL-prod DEFL-u

liftdefl-eq LIFTDEFL-prod u-liftdefl-liftdefl-of

lemmas [*domain-map-ID*] =
cfun-map-ID sfun-map-ID ssum-map-ID sprod-map-ID prod-map-ID u-map-ID

lemmas [*domain-isodefl*] =
isodefl-u isodefl-sfun isodefl-ssum isodefl-sprod
isodefl-cfun isodefl-prod isodefl-prod-u isodefl'-liftdefl-of
isodefl-u-liftdefl

lemmas [*domain-deflation*] =
deflation-cfun-map deflation-sfun-map deflation-ssum-map
deflation-sprod-map deflation-prod-map deflation-u-map

setup ‹
fold Domain-Take-Proofs.add-rec-type
[(@{*type-name cfun*}, [*true*, *true*]),
(@{*type-name sfun*}, [*true*, *true*]),
(@{*type-name ssum*}, [*true*, *true*]),
(@{*type-name sprod*}, [*true*, *true*]),
(@{*type-name prod*}, [*true*, *true*]),
(@{*type-name u*}, [*true*])]
›

end

30 A compact basis for powerdomains

theory *Compact-Basis*
imports *Universal*
begin

default-sort *bifinite*

30.1 A compact basis for powerdomains

definition *pd-basis* = $\{S::'a\ compact\text{-}basis\ set.\ finite\ S \wedge S \neq \{\}\}$

typedef *'a pd-basis* = *pd-basis* :: *'a compact-basis set set*
unfolding *pd-basis-def*
apply (*rule-tac x*={-} **in** *exI*)
apply *simp*
done

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*

The powerdomain basis type is countable.

lemma *pd-basis-countable*: $\exists f::'a\ pd\text{-}basis \Rightarrow nat.\ inj\ f$
proof −
  obtain g :: *'a compact-basis* ⇒ *nat* **where** *inj g*
    using *compact-basis.countable* **..**
  hence *image-g-eq*: $\bigwedge A\ B.\ g\ `\ A = g\ `\ B \longleftrightarrow A = B$
    by (*rule inj-image-eq-iff*)
  have *inj* (*λt. set-encode* (*g ‘ Rep-pd-basis t*))
    by (*simp add*: *inj-on-def set-encode-eq image-g-eq Rep-pd-basis-inject*)
  thus *?thesis* **by** − (*rule exI*)

qed

30.2 Unit and plus constructors

definition
  PDUnit :: *'a compact-basis* ⇒ *'a pd-basis* **where**
  PDUnit = (*λx. Abs-pd-basis* {*x*})

definition
  PDPlus :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
  PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* ∪ *Rep-pd-basis u*)

lemma *Rep-PDUnit*:
  Rep-pd-basis (*PDUnit x*) = {*x*}
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*: $\bigwedge a.\ P\ (PDUnit\ a)$
  assumes *PDPlus*: $\bigwedge a\ t.\ P\ t \Longrightarrow P\ (PDPlus\ (PDUnit\ a)\ t)$
  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*: $\bigwedge a.\ P\ (PDUnit\ a)$
assumes *PDPlus*: $\bigwedge t\ u.\ [\![P\ t;\ P\ u]\!] \Longrightarrow P\ (PDPlus\ t\ u)$
shows $P\ x$
apply (*induct x rule*: *pd-basis-induct1*)
apply (*rule PDUnit*, *erule PDPlus* [*OF PDUnit*])
done

30.3 Fold operator

definition
fold-pd ::
('*a compact-basis* ⇒ '*b*::*type*) ⇒ ('*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a pd-basis* ⇒ '*b*
where *fold-pd g f t* = *semilattice-set.F f* (*g* ' *Rep-pd-basis t*)

lemma *fold-pd-PDUnit*:
assumes *semilattice f*
shows *fold-pd g f* (*PDUnit x*) = *g x*
proof −
from *assms* **interpret** *semilattice-set f* **by** (*rule semilattice-set.intro*)
show *?thesis* **by** (*simp add*: *fold-pd-def Rep-PDUnit*)
qed

lemma *fold-pd-PDPlus*:
assumes *semilattice f*
shows *fold-pd g f* (*PDPlus t u*) = *f* (*fold-pd g f t*) (*fold-pd g f u*)
proof −
from *assms* **interpret** *semilattice-set f* **by** (*rule semilattice-set.intro*)
show *?thesis* **by** (*simp add*: *image-Un fold-pd-def Rep-PDPlus union*)
qed

end

31 Upper powerdomain

theory *UpperPD*
imports *Compact-Basis*
begin

31.1 Basis preorder

definition

upper-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** $\leq\sharp$ *50*) **where**
upper-le = ($\lambda u\ v.\ \forall y{\in}Rep\text{-}pd\text{-}basis\ v.\ \exists x{\in}Rep\text{-}pd\text{-}basis\ u.\ x \sqsubseteq y$)

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* $\lor$ *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* $\land$ *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

31.2 Type definition

typedef $'a$ *upper-pd* $(('(-')\sharp)) =$
$\{S::'a$ *pd-basis set*. *upper-le.ideal* $S\}$
by (*rule upper-le.ex-ideal*)

instantiation *upper-pd* :: (*bifinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-upper-pd* $x \subseteq$ *Rep-upper-pd* y

instance ..
end

instance *upper-pd* :: (*bifinite*) *po*
using *type-definition-upper-pd below-upper-pd-def*
by (*rule upper-le.typedef-ideal-po*)

instance *upper-pd* :: (*bifinite*) *cpo*
using *type-definition-upper-pd below-upper-pd-def*
by (*rule upper-le.typedef-ideal-cpo*)

definition
upper-principal :: $'a$ *pd-basis* $\Rightarrow$ $'a$ *upper-pd* **where**
upper-principal $t =$ *Abs-upper-pd* $\{u.\ u \leq\sharp\ t\}$

interpretation *upper-pd*:
ideal-completion upper-le upper-principal Rep-upper-pd
using *type-definition-upper-pd below-upper-pd-def*
using *upper-principal-def pd-basis-countable*
by (*rule upper-le.typedef-ideal-completion*)

Upper powerdomain is pointed

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *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*: $\bot$ = *upper-principal* (*PDUnit compact-bot*)
by (*rule upper-pd-minimal* [*THEN bottomI*, *symmetric*])

31.3 Monadic unit and plus

definition
upper-unit :: *'a* → *'a upper-pd* **where**
upper-unit = *compact-basis.extension* (λ*a*. *upper-principal* (*PDUnit a*))

definition
upper-plus :: *'a upper-pd* → *'a upper-pd* → *'a upper-pd* **where**
upper-plus = *upper-pd.extension* (λ*t*. *upper-pd.extension* (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* ⇒ *logic* ({-}♯)

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.extension-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.extension-principal*
upper-pd.extension-mono PDPlus-upper-mono)

interpretation *upper-add*: *semilattice upper-add* **proof**
fix *xs ys zs* :: *'a upper-pd*
show (*xs* ∪♯ *ys*) ∪♯ *zs* = *xs* ∪♯ (*ys* ∪♯ *zs*)
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct zs rule*: *upper-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done
show *xs* ∪♯ *ys* = *ys* ∪♯ *xs*
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *upper-pd.principal-induct*, *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 rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *upper-pd.principal-induct*, *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* ⊑ *ys*; *xs* ⊑ *zs*⟧ ⟹ *xs* ⊑ *ys* ∪♯ *zs*
apply (*subst upper-plus-absorb* [*of xs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *upper-below-plus-iff* [*simp*]:
xs ⊑ *ys* ∪♯ *zs* ⟷ *xs* ⊑ *ys* ∧ *xs* ⊑ *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* [*simp*]:
xs ∪♯ *ys* ⊑ {*z*}♯ ⟷ *xs* ⊑ {*z*}♯ ∨ *ys* ⊑ {*z*}♯
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct z rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)

done

lemma *upper-unit-below-iff* [*simp*]: $\{x\}\sharp \sqsubseteq \{y\}\sharp \longleftrightarrow x \sqsubseteq y$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct y rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
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*]: $\{\bot\}\sharp = \bot$
using *upper-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add*: *inst-upper-pd-pcpo*)

lemma *upper-plus-strict1* [*simp*]: $\bot \cup\sharp\ ys = \bot$
by (*rule bottomI*, *rule upper-plus-below1*)

lemma *upper-plus-strict2* [*simp*]: $xs \cup\sharp \bot = \bot$
by (*rule bottomI*, *rule upper-plus-below2*)

lemma *upper-unit-bottom-iff* [*simp*]: $\{x\}\sharp = \bot \longleftrightarrow x = \bot$
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-unit-eq-iff*)

lemma *upper-plus-bottom-iff* [*simp*]:
$xs \cup\sharp\ ys = \bot \longleftrightarrow xs = \bot \lor ys = \bot$
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *upper-pd.principal-induct*, *simp*)
apply (*simp add*: *inst-upper-pd-pcpo upper-pd.principal-eq-iff*
upper-le-PDPlus-PDUnit-iff)
done

lemma *compact-upper-unit*: *compact x* $\Longrightarrow$ *compact* $\{x\}\sharp$
by (*auto dest*!: *compact-basis.compact-imp-principal*)

lemma *compact-upper-unit-iff* [*simp*]: *compact* $\{x\}\sharp \longleftrightarrow$ *compact x*
apply (*safe elim*!: *compact-upper-unit*)
apply (*simp only*: *compact-def upper-unit-below-iff* [*symmetric*])
apply (*erule adm-subst* [*OF cont-Rep-cfun2*])
done

lemma *compact-upper-plus* [*simp*]:
$[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* ($xs \cup\sharp\ ys$)
by (*auto dest*!: *upper-pd.compact-imp-principal*)

31.4 Induction rules

lemma *upper-pd-induct1*:
  assumes *P*: *adm P*
  assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
  assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\sharp;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\sharp \cup\sharp\ ys)$
  shows *P* (*xs*::'*a upper-pd*)
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*
  [*case-names adm upper-unit upper-plus*, *induct type*: *upper-pd*]:
  assumes *P*: *adm P*
  assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
  assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs \cup\sharp\ ys)$
  shows *P* (*xs*::'*a upper-pd*)
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

31.5 Monadic bind

definition
  upper-bind-basis ::
  '*a pd-basis* ⇒ ('*a* → '*b upper-pd*) → '*b upper-pd* **where**
  upper-bind-basis = *fold-pd*
    ($\lambda a.\ \Lambda\ f.\ f \cdot$(*Rep-compact-basis a*))
    ($\lambda x\ y.\ \Lambda\ f.\ x \cdot f \cup\sharp\ y \cdot f$)

lemma *ACI-upper-bind*:
  semilattice ($\lambda x\ y.\ \Lambda\ f.\ x \cdot f \cup\sharp\ y \cdot f$)
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-compact-basis a*))
  upper-bind-basis (*PDPlus t u*) =
    ($\Lambda\ f.$ *upper-bind-basis* $t \cdot f \cup\sharp$ *upper-bind-basis* $u \cdot f$)

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*:
$t \le\sharp\ u \Longrightarrow$ *upper-bind-basis* $t \sqsubseteq$ *upper-bind-basis* u
unfolding *cfun-below-iff*
apply (*erule upper-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *below-trans* [*OF upper-plus-below1*])
apply *simp*
done

definition
upper-bind :: *'a upper-pd* → (*'a* → *'b upper-pd*) → *'b upper-pd* **where**
upper-bind = *upper-pd.extension upper-bind-basis*

syntax
-upper-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
((*3*⋃♯-∈-./ -) [*0*, *0*, *10*] *10*)

translations
⋃♯*x*∈*xs*. *e* == *CONST upper-bind*·*xs*·(Λ *x*. *e*)

lemma *upper-bind-principal* [*simp*]:
upper-bind·(*upper-principal t*) = *upper-bind-basis t*
unfolding *upper-bind-def*
apply (*rule upper-pd.extension-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 rule*: *upper-pd.principal-induct*, *simp*,
induct ys rule: *upper-pd.principal-induct*, *simp*, *simp*)

lemma *upper-bind-strict* [*simp*]: *upper-bind*·⊥·*f* = *f*·⊥
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-bind-unit*)

lemma *upper-bind-bind*:
upper-bind·(*upper-bind*·*xs*·*f*)·*g* = *upper-bind*·*xs*·(Λ *x*. *upper-bind*·(*f*·*x*)·*g*)
by (*induct xs*, *simp-all*)

31.6 Map

definition
upper-map :: $('a \to 'b) \to 'a$ *upper-pd* $\to 'b$ *upper-pd* **where**
upper-map = (Λ *f xs*. *upper-bind*·*xs*·(Λ *x*. {*f*·*x*}♯))

lemma *upper-map-unit* [*simp*]:
upper-map·*f*·{*x*}♯ = {*f*·*x*}♯
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-map-bottom* [*simp*]: *upper-map*·*f*·⊥ = {*f*·⊥}♯
unfolding *upper-map-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*: *cfun-eq-iff 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-bind-map*:
upper-bind·(*upper-map*·*f*·*xs*)·*g* = *upper-bind*·*xs*·(Λ *x*. *g*·(*f*·*x*))
by (*simp add*: *upper-map-def upper-bind-bind*)

lemma *upper-map-bind*:
upper-map·*f*·(*upper-bind*·*xs*·*g*) = *upper-bind*·*xs*·(Λ *x*. *upper-map*·*f*·(*g*·*x*))
by (*simp add*: *upper-map-def upper-bind-bind*)

lemma *ep-pair-upper-map*: *ep-pair e p* ⟹ *ep-pair* (*upper-map*·*e*) (*upper-map*·*p*)
apply *standard*
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 del*: *upper-below-plus-iff*)
done

lemma *deflation-upper-map*: *deflation d* ⟹ *deflation* (*upper-map*·*d*)
apply *standard*
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 del*: *upper-below-plus-iff*)
done

lemma *finite-deflation-upper-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*upper-map·d*)
proof (*rule finite-deflation-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* (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* (*upper-principal* ' *Rep-pd-basis* −' (*Pow* (*Rep-compact-basis* −' *range* (λ*x*. *d·x*)))) **by** *simp*
hence *finite* (*range* (λ*xs*. *upper-map·d·xs*))
apply (*rule rev-finite-subset*)
apply *clarsimp*
apply (*induct-tac xs rule*: *upper-pd.principal-induct*)
apply (*simp add*: *adm-mem-finite* ∗)
apply (*rename-tac t*, *induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*] *upper-map-unit*)
apply *simp*
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 range-eqI*)
apply (*erule sym*)
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
thus *finite* {*xs*. *upper-map·d·xs* = *xs*}
by (*rule finite-range-imp-finite-fixes*)
qed

31.7 Upper powerdomain is bifinite

lemma *approx-chain-upper-map*:
assumes *approx-chain a*
shows *approx-chain* (λ*i*. *upper-map*·(*a i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP upper-map-ID finite-deflation-upper-map*)

instance *upper-pd* :: (*bifinite*) *bifinite*
proof
  show $\exists$ (*a*::*nat* $\Rightarrow$ *'a upper-pd* $\rightarrow$ *'a upper-pd*). *approx-chain a*
    using *bifinite* [**where** *'a*=*'a*]
    by (*fast intro*!: *approx-chain-upper-map*)
qed

31.8 Join

definition
  upper-join :: *'a upper-pd upper-pd* $\rightarrow$ *'a upper-pd* **where**
  upper-join = (Λ *xss*. *upper-bind*·*xss*·(Λ *xs*. *xs*))

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-join-bottom* [*simp*]: *upper-join*·$\bot$ = $\bot$
unfolding *upper-join-def* **by** *simp*

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*)

end

32 Lower powerdomain

theory *LowerPD*
imports *Compact-Basis*
begin

32.1 Basis preorder

definition

lower-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** $\leq\flat$ *50*) **where**
lower-le = (λ*u v*. ∀ *x*∈*Rep-pd-basis u*. ∃ *y*∈*Rep-pd-basis v*. *x* ⊑ *y*)

lemma *lower-le-refl* [*simp*]: *t* $\leq\flat$ *t*
unfolding *lower-le-def* **by** *fast*

lemma *lower-le-trans*: ⟦*t* $\leq\flat$ *u*; *u* $\leq\flat$ *v*⟧ ⟹ *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* ⊑ *y* ⟹ *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*⟧ ⟹ *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* ⊑ *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* ∨ *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* ∧ *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*: ⋀*a b*. *a* ⊑ *b* ⟹ *P* (*PDUnit a*) (*PDUnit b*)
assumes *2*: ⋀*t u a*. *P* (*PDUnit a*) *t* ⟹ *P* (*PDUnit a*) (*PDPlus t u*)
assumes *3*: ⋀*t u v*. ⟦*P t v*; *P u v*⟧ ⟹ *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

32.2 Type definition

typedef *'a lower-pd* $(('(\text{-}')\flat)) =$
$\{S::'a\ pd\text{-}basis\ set.\ lower\text{-}le.ideal\ S\}$
by (*rule lower-le.ex-ideal*)

instantiation *lower-pd* :: (*bifinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-lower-pd x* $\subseteq$ *Rep-lower-pd y*

instance ..
end

instance *lower-pd* :: (*bifinite*) *po*
using *type-definition-lower-pd below-lower-pd-def*
by (*rule lower-le.typedef-ideal-po*)

instance *lower-pd* :: (*bifinite*) *cpo*
using *type-definition-lower-pd below-lower-pd-def*
by (*rule lower-le.typedef-ideal-cpo*)

definition
lower-principal :: *'a pd-basis* $\Rightarrow$ *'a lower-pd* **where**
lower-principal t = *Abs-lower-pd* $\{u.\ u \leq\flat\ t\}$

interpretation *lower-pd*:
ideal-completion lower-le lower-principal Rep-lower-pd
using *type-definition-lower-pd below-lower-pd-def*
using *lower-principal-def pd-basis-countable*
by (*rule lower-le.typedef-ideal-completion*)

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*: ⊥ = *lower-principal* (*PDUnit compact-bot*)
by (*rule lower-pd-minimal* [*THEN bottomI*, *symmetric*])

32.3 Monadic unit and plus

definition
lower-unit :: *'a* → *'a lower-pd* **where**
lower-unit = *compact-basis.extension* (λ*a*. *lower-principal* (*PDUnit a*))

definition
lower-plus :: *'a lower-pd* → *'a lower-pd* → *'a lower-pd* **where**
lower-plus = *lower-pd.extension* (λ*t*. *lower-pd.extension* (λ*u*.
lower-principal (*PDPlus t u*)))

abbreviation
lower-add :: *'a lower-pd* ⇒ *'a lower-pd* ⇒ *'a lower-pd*
(**infixl** ∪♭ *65*) **where**
xs ∪♭ *ys* == *lower-plus*·*xs*·*ys*

syntax
-lower-pd :: *args* ⇒ *logic* ({-}♭)

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.extension-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.extension-principal*
lower-pd.extension-mono PDPlus-lower-mono)

interpretation *lower-add*: *semilattice lower-add* **proof**
fix *xs ys zs* :: *'a lower-pd*
show (*xs* ∪♭ *ys*) ∪♭ *zs* = *xs* ∪♭ (*ys* ∪♭ *zs*)
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct zs rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done
show *xs* ∪♭ *ys* = *ys* ∪♭ *xs*
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)

apply (*induct ys rule*: *lower-pd.principal-induct*, *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* ⊑ *xs* ∪♭ *ys*
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-lower-le*)
done

lemma *lower-plus-below2*: *ys* ⊑ *xs* ∪♭ *ys*
by (*subst lower-plus-commute*, *rule lower-plus-below1*)

lemma *lower-plus-least*: ⟦*xs* ⊑ *zs*; *ys* ⊑ *zs*⟧ ⟹ *xs* ∪♭ *ys* ⊑ *zs*
apply (*subst lower-plus-absorb* [*of zs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *lower-plus-below-iff* [*simp*]:
xs ∪♭ *ys* ⊑ *zs* ⟷ *xs* ⊑ *zs* ∧ *ys* ⊑ *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* [*simp*]:
{*x*}♭ ⊑ *ys* ∪♭ *zs* ⟷ {*x*}♭ ⊑ *ys* ∨ {*x*}♭ ⊑ *zs*
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct ys rule*: *lower-pd.principal-induct*, *simp*)

apply (*induct zs rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
done

lemma *lower-unit-below-iff* [*simp*]: $\{x\}\flat \sqsubseteq \{y\}\flat \longleftrightarrow x \sqsubseteq y$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct y rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
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*]: $\{\bot\}\flat = \bot$
using *lower-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add*: *inst-lower-pd-pcpo*)

lemma *lower-unit-bottom-iff* [*simp*]: $\{x\}\flat = \bot \longleftrightarrow x = \bot$
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-unit-eq-iff*)

lemma *lower-plus-bottom-iff* [*simp*]:
  $xs \cup\flat\ ys = \bot \longleftrightarrow xs = \bot \wedge ys = \bot$
apply *safe*
apply (*rule bottomI*, *erule subst*, *rule lower-plus-below1*)
apply (*rule bottomI*, *erule subst*, *rule lower-plus-below2*)
apply (*rule lower-plus-absorb*)
done

lemma *lower-plus-strict1* [*simp*]: $\bot \cup\flat\ ys = ys$
apply (*rule below-antisym* [*OF - lower-plus-below2*])
apply (*simp add*: *lower-plus-least*)
done

lemma *lower-plus-strict2* [*simp*]: $xs \cup\flat \bot = xs$
apply (*rule below-antisym* [*OF - lower-plus-below1*])
apply (*simp add*: *lower-plus-least*)
done

lemma *compact-lower-unit*: *compact x* $\Longrightarrow$ *compact* $\{x\}\flat$
by (*auto dest*!: *compact-basis.compact-imp-principal*)

lemma *compact-lower-unit-iff* [*simp*]: *compact* $\{x\}\flat \longleftrightarrow$ *compact x*
apply (*safe elim*!: *compact-lower-unit*)
apply (*simp only*: *compact-def lower-unit-below-iff* [*symmetric*])

apply (*erule adm-subst* [*OF cont-Rep-cfun2*])
done

lemma *compact-lower-plus* [*simp*]:
⟦*compact xs*; *compact ys*⟧ ⟹ *compact* (*xs* ∪♭ *ys*)
by (*auto dest*!: *lower-pd.compact-imp-principal*)

32.4 Induction rules

lemma *lower-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: ⋀*x*. *P* {*x*}♭
assumes *insert*:
⋀*x ys*. ⟦*P* {*x*}♭; *P ys*⟧ ⟹ *P* ({*x*}♭ ∪♭ *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*
[*case-names adm lower-unit lower-plus*, *induct type*: *lower-pd*]:
assumes *P*: *adm P*
assumes *unit*: ⋀*x*. *P* {*x*}♭
assumes *plus*: ⋀*xs ys*. ⟦*P xs*; *P ys*⟧ ⟹ *P* (*xs* ∪♭ *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

32.5 Monadic bind

definition
lower-bind-basis ::
'*a pd-basis* ⇒ ('*a* → '*b lower-pd*) → '*b lower-pd* **where**
lower-bind-basis = *fold-pd*
(λ*a*. Λ *f*. *f*·(*Rep-compact-basis a*))
(λ*x y*. Λ *f*. *x*·*f* ∪♭ *y*·*f*)

lemma *ACI-lower-bind*:
semilattice (λ*x y*. Λ *f*. *x*·*f* ∪♭ *y*·*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*) =
(Λ *f*. *f*·(*Rep-compact-basis a*))
lower-bind-basis (*PDPlus t u*) =
(Λ *f*. *lower-bind-basis t*·*f* ∪♭ *lower-bind-basis u*·*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 ≤♭ *u* ⟹ *lower-bind-basis t* ⊑ *lower-bind-basis u*
unfolding *cfun-below-iff*
apply (*erule lower-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *rev-below-trans* [*OF lower-plus-below1*])
apply *simp*
done

definition
lower-bind :: *'a lower-pd* → (*'a* → *'b lower-pd*) → *'b lower-pd* **where**
lower-bind = *lower-pd.extension lower-bind-basis*

syntax
-lower-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
((*3*⋃♭-∈-./ -) [*0*, *0*, *10*] *10*)

translations
⋃♭*x*∈*xs*. *e* == *CONST lower-bind*·*xs*·(Λ *x*. *e*)

lemma *lower-bind-principal* [*simp*]:
lower-bind·(*lower-principal t*) = *lower-bind-basis t*
unfolding *lower-bind-def*
apply (*rule lower-pd.extension-principal*)
apply (*erule lower-bind-basis-mono*)
done

lemma *lower-bind-unit* [*simp*]:
lower-bind·{*x*}♭·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *lower-bind-plus* [*simp*]:
lower-bind·(*xs* ∪♭ *ys*)·*f* = *lower-bind*·*xs*·*f* ∪♭ *lower-bind*·*ys*·*f*
by (*induct xs rule*: *lower-pd.principal-induct*, *simp*,
induct ys rule: *lower-pd.principal-induct*, *simp*, *simp*)

lemma *lower-bind-strict* [*simp*]: *lower-bind*·⊥·*f* = *f*·⊥
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-bind-unit*)

lemma *lower-bind-bind*:
lower-bind·(*lower-bind*·*xs*·*f*)·*g* = *lower-bind*·*xs*·(Λ *x*. *lower-bind*·(*f*·*x*)·*g*)
by (*induct xs*, *simp-all*)

32.6 Map

definition
lower-map :: ('*a* → '*b*) → '*a lower-pd* → '*b lower-pd* **where**
lower-map = (Λ *f xs*. *lower-bind*·*xs*·(Λ *x*. {*f*·*x*}♭))

lemma *lower-map-unit* [*simp*]:
lower-map·*f*·{*x*}♭ = {*f*·*x*}♭
unfolding *lower-map-def* **by** *simp*

lemma *lower-map-plus* [*simp*]:
lower-map·*f*·(*xs* ∪♭ *ys*) = *lower-map*·*f*·*xs* ∪♭ *lower-map*·*f*·*ys*
unfolding *lower-map-def* **by** *simp*

lemma *lower-map-bottom* [*simp*]: *lower-map*·*f*·⊥ = {*f*·⊥}♭
unfolding *lower-map-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*: *cfun-eq-iff 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-bind-map*:
lower-bind·(*lower-map*·*f*·*xs*)·*g* = *lower-bind*·*xs*·(Λ *x*. *g*·(*f*·*x*))
by (*simp add*: *lower-map-def lower-bind-bind*)

lemma *lower-map-bind*:
lower-map·*f*·(*lower-bind*·*xs*·*g*) = *lower-bind*·*xs*·(Λ *x*. *lower-map*·*f*·(*g*·*x*))
by (*simp add*: *lower-map-def lower-bind-bind*)

lemma *ep-pair-lower-map*: *ep-pair e p* ⟹ *ep-pair* (*lower-map*·*e*) (*lower-map*·*p*)
apply *standard*
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 del*: *lower-plus-below-iff*)
done

lemma *deflation-lower-map*: *deflation d* $\Longrightarrow$ *deflation* (*lower-map*·*d*)
apply *standard*
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 del*: *lower-plus-below-iff*)
done

lemma *finite-deflation-lower-map*:
  assumes *finite-deflation d* **shows** *finite-deflation* (*lower-map*·*d*)
proof (*rule finite-deflation-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* (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* (*lower-principal* ' *Rep-pd-basis* −' (*Pow* (*Rep-compact-basis* −' *range* (λ*x*. *d*·*x*)))) **by** *simp*
  hence *finite* (*range* (λ*xs*. *lower-map*·*d*·*xs*))
    apply (*rule rev-finite-subset*)
    apply *clarsimp*
    apply (*induct-tac xs rule*: *lower-pd.principal-induct*)
    apply (*simp add*: *adm-mem-finite* ∗)
    apply (*rename-tac t*, *induct-tac t rule*: *pd-basis-induct*)
    apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *lower-map-unit*)
    apply *simp*
    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 range-eqI*)
    apply (*erule sym*)
    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
  thus *finite* {*xs*. *lower-map*·*d*·*xs* = *xs*}
    by (*rule finite-range-imp-finite-fixes*)

qed

32.7 Lower powerdomain is bifinite

lemma *approx-chain-lower-map*:
assumes *approx-chain a*
shows *approx-chain* (λ*i. lower-map*·(*a i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP lower-map-ID finite-deflation-lower-map*)

instance *lower-pd* :: (*bifinite*) *bifinite*
proof
show ∃(*a*::*nat* ⇒ ′*a lower-pd* → ′*a lower-pd*). *approx-chain a*
using *bifinite* [**where** ′*a*=′*a*]
by (*fast intro*!: *approx-chain-lower-map*)
qed

32.8 Join

definition
lower-join :: ′*a lower-pd lower-pd* → ′*a lower-pd* **where**
lower-join = (Λ *xss*. *lower-bind*·*xss*·(Λ *xs*. *xs*))

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-join-bottom* [*simp*]: *lower-join*·⊥ = ⊥
unfolding *lower-join-def* **by** *simp*

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*)

end

33 Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

33.1 Basis preorder

definition
convex-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** $\le\natural$ *50*) **where**
convex-le = ($\lambda u\ v.\ u \le\sharp\ v \wedge u \le\flat\ v$)

lemma *convex-le-refl* [*simp*]: $t \le\natural\ t$
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-refl lower-le-refl*)

lemma *convex-le-trans*: ⟦$t \le\natural\ u$; $u \le\natural\ v$⟧ ⟹ $t \le\natural\ v$
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* $\le\natural$ *t*
unfolding *convex-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-convex-mono*: $x \sqsubseteq y$ ⟹ *PDUnit x* $\le\natural$ *PDUnit y*
unfolding *convex-le-def* **by** (*fast intro*: *PDUnit-upper-mono PDUnit-lower-mono*)

lemma *PDPlus-convex-mono*: ⟦$s \le\natural\ t$; $u \le\natural\ v$⟧ ⟹ *PDPlus s u* $\le\natural$ *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* $\le\natural$ *PDUnit b*) = ($a \sqsubseteq b$)
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit* **by** *fast*

lemma *convex-le-PDUnit-lemma1*:
(*PDUnit a* $\le\natural$ *t*) = (∀ *b*∈*Rep-pd-basis t*. $a \sqsubseteq b$)
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* $\le\natural$ *PDPlus t u*) = (*PDUnit a* $\le\natural$ *t* ∧ *PDUnit a* $\le\natural$ *u*)
unfolding *convex-le-PDUnit-lemma1 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDUnit-lemma2*:
(*t* $\le\natural$ *PDUnit b*) = (∀ *a*∈*Rep-pd-basis t*. $a \sqsubseteq b$)
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* $\le\natural$ *PDUnit a*) = (*t* $\le\natural$ *PDUnit a* ∧ *u* $\le\natural$ *PDUnit a*)

```
unfolding convex-le-PDUnit-lemma2 Rep-PDPlus by fast

lemma convex-le-PDPlus-lemma:
  assumes z: PDPlus t u ≤♮ z
  shows ∃ v w. z = PDPlus v w ∧ t ≤♮ v ∧ u ≤♮ w
proof (intro exI conjI)
  let ?A = {b∈Rep-pd-basis z. ∃ a∈Rep-pd-basis t. a ⊑ b}
  let ?B = {b∈Rep-pd-basis z. ∃ a∈Rep-pd-basis u. a ⊑ 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 ≤♮ ?v u ≤♮ ?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 ≤♮ 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

33.2 Type definition

typedef *'a convex-pd* (('(-')♮)) =
{*S*::*'a pd-basis set*. *convex-le.ideal S*}
by (*rule convex-le.ex-ideal*)

instantiation *convex-pd* :: (*bifinite*) *below*
begin

definition
$x \sqsubseteq y \longleftrightarrow$ *Rep-convex-pd x* $\subseteq$ *Rep-convex-pd y*

instance ..
end

instance *convex-pd* :: (*bifinite*) *po*
using *type-definition-convex-pd below-convex-pd-def*
by (*rule convex-le.typedef-ideal-po*)

instance *convex-pd* :: (*bifinite*) *cpo*
using *type-definition-convex-pd below-convex-pd-def*
by (*rule convex-le.typedef-ideal-cpo*)

definition
convex-principal :: *'a pd-basis* ⇒ *'a convex-pd* **where**
convex-principal t = *Abs-convex-pd* {*u*. *u* ≤♮ *t*}

interpretation *convex-pd*:
ideal-completion convex-le convex-principal Rep-convex-pd
using *type-definition-convex-pd below-convex-pd-def*
using *convex-principal-def pd-basis-countable*
by (*rule convex-le.typedef-ideal-completion*)

Convex powerdomain is pointed

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) ⊑ *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*: ⊥ = *convex-principal* (*PDUnit compact-bot*)
by (*rule convex-pd-minimal* [*THEN bottomI*, *symmetric*])

33.3 Monadic unit and plus

definition
convex-unit :: *'a* → *'a convex-pd* **where**
convex-unit = *compact-basis.extension* (λ*a*. *convex-principal* (*PDUnit a*))

definition
convex-plus :: *'a convex-pd* → *'a convex-pd* → *'a convex-pd* **where**
convex-plus = *convex-pd.extension* (λ*t*. *convex-pd.extension* (λ*u*.
convex-principal (*PDPlus t u*)))

abbreviation
convex-add :: *'a convex-pd* ⇒ *'a convex-pd* ⇒ *'a convex-pd*
(**infixl** ∪♮ *65*) **where**
xs ∪♮ *ys* == *convex-plus·xs·ys*

syntax
-convex-pd :: *args* ⇒ *logic* ({-}♮)

translations
{*x*,*xs*}♮ == {*x*}♮ ∪♮ {*xs*}♮
{*x*}♮ == *CONST convex-unit·x*

lemma *convex-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♮ = *convex-principal* (*PDUnit a*)
unfolding *convex-unit-def*
by (*simp add*: *compact-basis.extension-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.extension-principal*
convex-pd.extension-mono PDPlus-convex-mono)

interpretation *convex-add*: *semilattice convex-add* **proof**
fix *xs ys zs* :: *'a convex-pd*
show (*xs* ∪♮ *ys*) ∪♮ *zs* = *xs* ∪♮ (*ys* ∪♮ *zs*)
apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct zs rule*: *convex-pd.principal-induct*, *simp*)

apply (*simp add*: *PDPlus-assoc*)
done
show $xs \cup\natural ys = ys \cup\natural xs$
apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done
show $xs \cup\natural xs = xs$
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*]:
$\{x\}\natural \sqsubseteq ys \cup\natural zs \longleftrightarrow \{x\}\natural \sqsubseteq ys \wedge \{x\}\natural \sqsubseteq zs$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct zs rule*: *convex-pd.principal-induct*, *simp*)
apply *simp*
done

lemma *convex-plus-below-unit-iff* [*simp*]:
$xs \cup\natural ys \sqsubseteq \{z\}\natural \longleftrightarrow xs \sqsubseteq \{z\}\natural \wedge ys \sqsubseteq \{z\}\natural$
apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct z rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
done

lemma *convex-unit-below-iff* [*simp*]: $\{x\}\natural \sqsubseteq \{y\}\natural \longleftrightarrow x \sqsubseteq y$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct y rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
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*]: $\{\bot\}\natural = \bot$
using *convex-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add*: *inst-convex-pd-pcpo*)

lemma *convex-unit-bottom-iff* [*simp*]: $\{x\}\natural = \bot \longleftrightarrow x = \bot$
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-unit-eq-iff*)

lemma *compact-convex-unit*: *compact x* $\Longrightarrow$ *compact* $\{x\}\natural$
by (*auto dest*!: *compact-basis.compact-imp-principal*)

lemma *compact-convex-unit-iff* [*simp*]: *compact* $\{x\}\natural \longleftrightarrow$ *compact x*
apply (*safe elim*!: *compact-convex-unit*)
apply (*simp only*: *compact-def convex-unit-below-iff* [*symmetric*])
apply (*erule adm-subst* [*OF cont-Rep-cfun2*])
done

lemma *compact-convex-plus* [*simp*]:
 $[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* (*xs* $\cup\natural$ *ys*)
by (*auto dest*!: *convex-pd.compact-imp-principal*)

33.4 Induction rules

lemma *convex-pd-induct1*:
 assumes *P*: *adm P*
 assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
 assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\natural;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\natural \cup\natural\ ys)$
 shows *P* (*xs*::'*a convex-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*
 [*case-names adm convex-unit convex-plus*, *induct type*: *convex-pd*]:
 assumes *P*: *adm P*
 assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
 assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs \cup\natural\ ys)$
 shows *P* (*xs*::'*a convex-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

33.5 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*:
semilattice (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 *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.extension convex-bind-basis*

syntax
-convex-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
((*3*⋃♮-∈-./ -) [*0*, *0*, *10*] *10*)

translations
⋃♮*x*∈*xs*. *e* == *CONST convex-bind*·*xs*·(Λ *x*. *e*)

lemma *convex-bind-principal* [*simp*]:
  convex-bind·(*convex-principal t*) = *convex-bind-basis t*
unfolding *convex-bind-def*
apply (*rule convex-pd.extension-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 rule*: *convex-pd.principal-induct*, *simp*,
    induct ys rule: *convex-pd.principal-induct*, *simp*, *simp*)

lemma *convex-bind-strict* [*simp*]: *convex-bind*·⊥·*f* = *f*·⊥
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-bind-unit*)

lemma *convex-bind-bind*:
  convex-bind·(*convex-bind*·*xs*·*f*)·*g* =
    convex-bind·*xs*·(Λ *x*. *convex-bind*·(*f*·*x*)·*g*)
by (*induct xs*, *simp-all*)

33.6 Map

definition
  convex-map :: ('*a* → '*b*) → '*a convex-pd* → '*b convex-pd* **where**
  convex-map = (Λ *f xs*. *convex-bind*·*xs*·(Λ *x*. {*f*·*x*}♮))

lemma *convex-map-unit* [*simp*]:
  convex-map·*f*·{*x*}♮ = {*f*·*x*}♮
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-map-bottom* [*simp*]: *convex-map*·*f*·⊥ = {*f*·⊥}♮
unfolding *convex-map-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*: *cfun-eq-iff 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-bind-map*:
convex-bind·(*convex-map·f·xs*)*·g* = *convex-bind·xs·*(Λ *x*. *g·*(*f·x*))
by (*simp add*: *convex-map-def convex-bind-bind*)

lemma *convex-map-bind*:
convex-map·f·(*convex-bind·xs·g*) = *convex-bind·xs·*(Λ *x*. *convex-map·f·*(*g·x*))
by (*simp add*: *convex-map-def convex-bind-bind*)

lemma *ep-pair-convex-map*: *ep-pair e p* ⟹ *ep-pair* (*convex-map·e*) (*convex-map·p*)
apply *standard*
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 *standard*
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

lemma *finite-deflation-convex-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*convex-map·d*)
proof (*rule finite-deflation-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* (*range* (λ*xs*. *convex-map·d·xs*))
apply (*rule rev-finite-subset*)
apply *clarsimp*
apply (*induct-tac xs rule*: *convex-pd.principal-induct*)
apply (*simp add*: *adm-mem-finite* ∗)
apply (*rename-tac t*, *induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*] *convex-map-unit*)
apply *simp*
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 range-eqI*)
apply (*erule sym*)
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
thus *finite* {*xs*. *convex-map·d·xs* = *xs*}
by (*rule finite-range-imp-finite-fixes*)
qed

33.7 Convex powerdomain is bifinite

lemma *approx-chain-convex-map*:
assumes *approx-chain a*
shows *approx-chain* (*λi*. *convex-map*·(*a i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP convex-map-ID finite-deflation-convex-map*)

instance *convex-pd* :: (*bifinite*) *bifinite*
proof
show ∃(*a*::*nat* ⇒ *'a convex-pd* → *'a convex-pd*). *approx-chain a*
using *bifinite* [**where** *'a*=*'a*]
by (*fast intro*!: *approx-chain-convex-map*)
qed

33.8 Join

definition
convex-join :: *'a convex-pd convex-pd* → *'a convex-pd* **where**
convex-join = (Λ *xss*. *convex-bind·xss*·(Λ *xs*. *xs*))

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-join-bottom* [*simp*]: *convex-join*·⊥ = ⊥
unfolding *convex-join-def* **by** *simp*

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*)

33.9 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: $t \leq\natural\ u \Longrightarrow t \leq\sharp\ u$
unfolding *convex-le-def* **by** *simp*

definition
  convex-to-upper :: *'a convex-pd → 'a upper-pd* **where**
  convex-to-upper = convex-pd.extension 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.extension-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 rule*: *convex-pd.principal-induct*, *simp*,
    induct ys rule: *convex-pd.principal-induct*, *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*: $t \leq\natural\ u \Longrightarrow t \leq\flat\ u$
unfolding *convex-le-def* **by** *simp*

definition
convex-to-lower :: *'a convex-pd* → *'a lower-pd* **where**
convex-to-lower = convex-pd.extension lower-principal

lemma *convex-to-lower-principal* [*simp*]:
convex-to-lower·(convex-principal t) = lower-principal t
unfolding *convex-to-lower-def*
apply (*rule convex-pd.extension-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 rule*: *convex-pd.principal-induct*, *simp*,
induct ys rule: *convex-pd.principal-induct*, *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 (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*simp add*: *convex-le-def*)
done

lemmas *convex-plus-below-plus-iff* =
convex-pd-below-iff [**where** *xs*=*xs* ∪♮ *ys* **and** *ys*=*zs* ∪♮ *ws*]
for *xs ys zs ws*

lemmas *convex-pd-below-simps* =
convex-unit-below-plus-iff
convex-plus-below-unit-iff
convex-plus-below-plus-iff
convex-unit-below-iff
convex-to-upper-unit
convex-to-upper-plus
convex-to-lower-unit
convex-to-lower-plus
upper-pd-below-simps
lower-pd-below-simps

end

34 Powerdomains

theory *Powerdomains*
imports *ConvexPD Domain*
begin

34.1 Universal domain embeddings

definition *upper-emb* = *udom-emb* (λ*i*. *upper-map*·(*udom-approx i*))
definition *upper-prj* = *udom-prj* (λ*i*. *upper-map*·(*udom-approx i*))

definition *lower-emb* = *udom-emb* (λ*i*. *lower-map*·(*udom-approx i*))
definition *lower-prj* = *udom-prj* (λ*i*. *lower-map*·(*udom-approx i*))

definition *convex-emb* = *udom-emb* (λ*i*. *convex-map*·(*udom-approx i*))
definition *convex-prj* = *udom-prj* (λ*i*. *convex-map*·(*udom-approx i*))

lemma *ep-pair-upper*: *ep-pair upper-emb upper-prj*
unfolding *upper-emb-def upper-prj-def*
by (*simp add*: *ep-pair-udom approx-chain-upper-map*)

lemma *ep-pair-lower*: *ep-pair lower-emb lower-prj*
unfolding *lower-emb-def lower-prj-def*
by (*simp add*: *ep-pair-udom approx-chain-lower-map*)

lemma *ep-pair-convex*: *ep-pair convex-emb convex-prj*

unfolding *convex-emb-def convex-prj-def*
by (*simp add*: *ep-pair-udom approx-chain-convex-map*)

34.2 Deflation combinators

definition *upper-defl* :: *udom defl* → *udom defl*
where *upper-defl* = *defl-fun1 upper-emb upper-prj upper-map*

definition *lower-defl* :: *udom defl* → *udom defl*
where *lower-defl* = *defl-fun1 lower-emb lower-prj lower-map*

definition *convex-defl* :: *udom defl* → *udom defl*
where *convex-defl* = *defl-fun1 convex-emb convex-prj convex-map*

lemma *cast-upper-defl*:
cast·(*upper-defl*·*A*) = *upper-emb oo upper-map*·(*cast*·*A*) *oo upper-prj*
using *ep-pair-upper finite-deflation-upper-map*
unfolding *upper-defl-def* **by** (*rule cast-defl-fun1*)

lemma *cast-lower-defl*:
cast·(*lower-defl*·*A*) = *lower-emb oo lower-map*·(*cast*·*A*) *oo lower-prj*
using *ep-pair-lower finite-deflation-lower-map*
unfolding *lower-defl-def* **by** (*rule cast-defl-fun1*)

lemma *cast-convex-defl*:
cast·(*convex-defl*·*A*) = *convex-emb oo convex-map*·(*cast*·*A*) *oo convex-prj*
using *ep-pair-convex finite-deflation-convex-map*
unfolding *convex-defl-def* **by** (*rule cast-defl-fun1*)

34.3 Domain class instances

instantiation *upper-pd* :: (*domain*) *domain*
begin

definition
emb = *upper-emb oo upper-map*·*emb*

definition
prj = *upper-map*·*prj oo upper-prj*

definition
defl (*t*::′*a upper-pd itself*) = *upper-defl*·*DEFL*(′*a*)

definition
(*liftemb* :: ′*a upper-pd u* → *udom u*) = *u-map*·*emb*

definition
(*liftprj* :: *udom u* → ′*a upper-pd u*) = *u-map*·*prj*

definition

liftdefl (*t*::′*a upper-pd itself*) = *liftdefl-of*·*DEFL*(′*a upper-pd*)

instance proof
  show *ep-pair emb* (*prj* :: *udom* → ′*a upper-pd*)
    unfolding *emb-upper-pd-def prj-upper-pd-def*
    by (*intro ep-pair-comp ep-pair-upper ep-pair-upper-map ep-pair-emb-prj*)
next
  show *cast*·*DEFL*(′*a upper-pd*) = *emb oo* (*prj* :: *udom* → ′*a upper-pd*)
    unfolding *emb-upper-pd-def prj-upper-pd-def defl-upper-pd-def cast-upper-defl*
    by (*simp add*: *cast-DEFL oo-def cfun-eq-iff upper-map-map*)
qed (*fact liftemb-upper-pd-def liftprj-upper-pd-def liftdefl-upper-pd-def*)+

end

instantiation *lower-pd* :: (*domain*) *domain*
begin

definition
  emb = *lower-emb oo lower-map*·*emb*

definition
  prj = *lower-map*·*prj oo lower-prj*

definition
  defl (*t*::′*a lower-pd itself*) = *lower-defl*·*DEFL*(′*a*)

definition
  (*liftemb* :: ′*a lower-pd u* → *udom u*) = *u-map*·*emb*

definition
  (*liftprj* :: *udom u* → ′*a lower-pd u*) = *u-map*·*prj*

definition
  liftdefl (*t*::′*a lower-pd itself*) = *liftdefl-of*·*DEFL*(′*a lower-pd*)

instance proof
  show *ep-pair emb* (*prj* :: *udom* → ′*a lower-pd*)
    unfolding *emb-lower-pd-def prj-lower-pd-def*
    by (*intro ep-pair-comp ep-pair-lower ep-pair-lower-map ep-pair-emb-prj*)
next
  show *cast*·*DEFL*(′*a lower-pd*) = *emb oo* (*prj* :: *udom* → ′*a lower-pd*)
    unfolding *emb-lower-pd-def prj-lower-pd-def defl-lower-pd-def cast-lower-defl*
    by (*simp add*: *cast-DEFL oo-def cfun-eq-iff lower-map-map*)
qed (*fact liftemb-lower-pd-def liftprj-lower-pd-def liftdefl-lower-pd-def*)+

end

instantiation *convex-pd* :: (*domain*) *domain*
begin

definition
emb = *convex-emb oo convex-map·emb*

definition
prj = *convex-map·prj oo convex-prj*

definition
defl (*t*::'*a convex-pd itself*) = *convex-defl·DEFL*('*a*)

definition
(*liftemb* :: '*a convex-pd u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → '*a convex-pd u*) = *u-map·prj*

definition
liftdefl (*t*::'*a convex-pd itself*) = *liftdefl-of·DEFL*('*a convex-pd*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → '*a convex-pd*)
unfolding *emb-convex-pd-def prj-convex-pd-def*
by (*intro ep-pair-comp ep-pair-convex ep-pair-convex-map ep-pair-emb-prj*)
next
show *cast·DEFL*('*a convex-pd*) = *emb oo* (*prj* :: *udom* → '*a convex-pd*)
unfolding *emb-convex-pd-def prj-convex-pd-def defl-convex-pd-def cast-convex-defl*
by (*simp add*: *cast-DEFL oo-def cfun-eq-iff convex-map-map*)
qed (*fact liftemb-convex-pd-def liftprj-convex-pd-def liftdefl-convex-pd-def*)+

end

lemma *DEFL-upper*: *DEFL*('*a*::*domain upper-pd*) = *upper-defl·DEFL*('*a*)
by (*rule defl-upper-pd-def*)

lemma *DEFL-lower*: *DEFL*('*a*::*domain lower-pd*) = *lower-defl·DEFL*('*a*)
by (*rule defl-lower-pd-def*)

lemma *DEFL-convex*: *DEFL*('*a*::*domain convex-pd*) = *convex-defl·DEFL*('*a*)
by (*rule defl-convex-pd-def*)

34.4 Isomorphic deflations

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 $\Longrightarrow$ *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 $\Longrightarrow$ *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

34.5 Domain package setup for powerdomains

lemmas [*domain-defl-simps*] = *DEFL-upper DEFL-lower DEFL-convex*
lemmas [*domain-map-ID*] = *upper-map-ID lower-map-ID convex-map-ID*
lemmas [*domain-isodefl*] = *isodefl-upper isodefl-lower isodefl-convex*

lemmas [*domain-deflation*] =
deflation-upper-map deflation-lower-map deflation-convex-map

setup ‹
fold Domain-Take-Proofs.add-rec-type
[(@{*type-name upper-pd*}, [*true*]),
(@{*type-name lower-pd*}, [*true*]),
(@{*type-name convex-pd*}, [*true*])]
›

end

theory *HOLCF*
imports
Main
Domain
Powerdomains
begin

default-sort *domain*

end