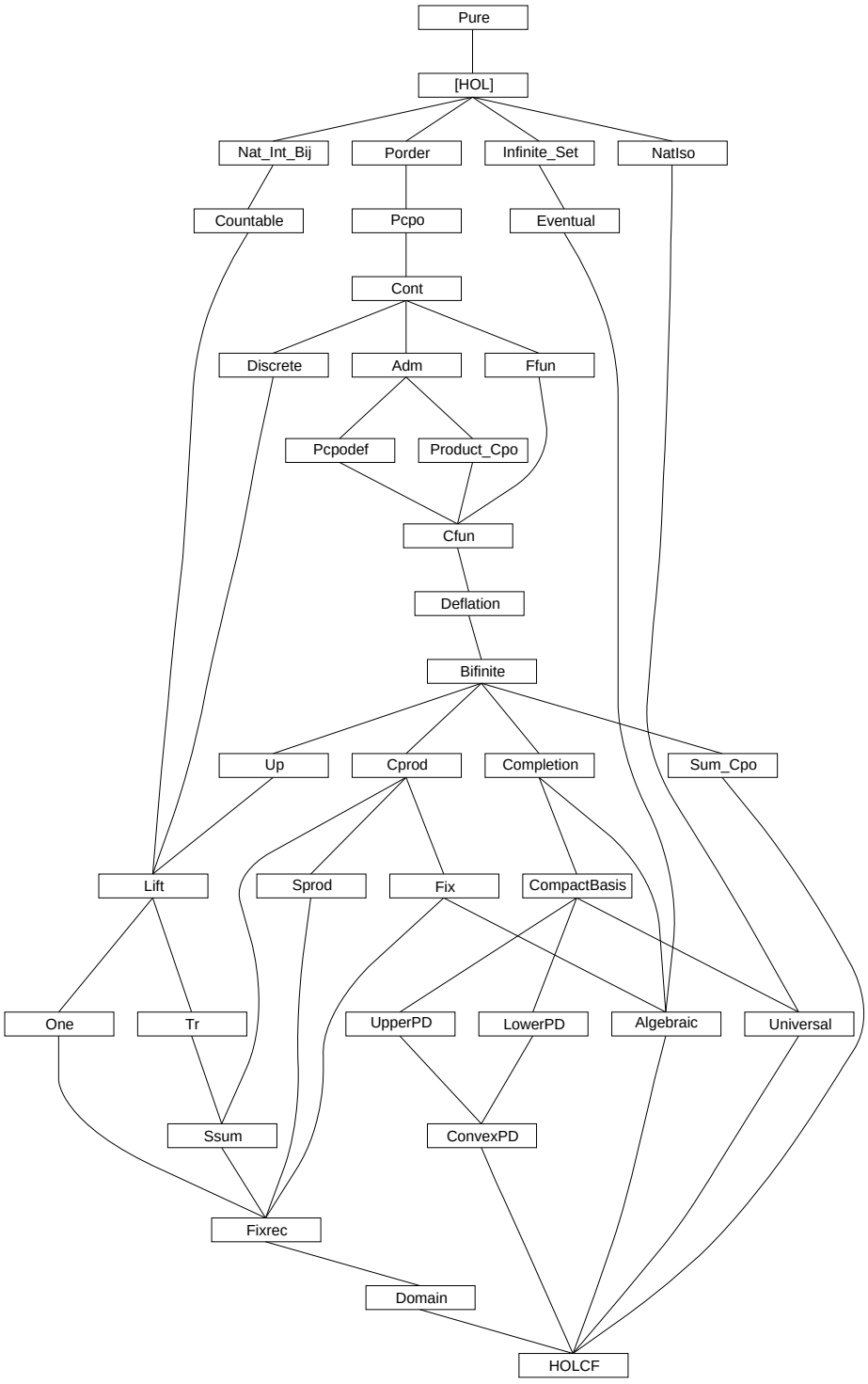
Isabelle/HOLCF — Higher-Order Logic of Computable Functions

April 19, 2009

Contents


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

1 Porder: Partial orders

theory *Porder*
imports *Main*
begin

1.1 Type class for partial orders

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

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

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

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

minimal fixes least element

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

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

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

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

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

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

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

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

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

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

1.2 Upper bounds

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

lemma *is-ubI*: (⋀*x*. *x* ∈ *S* ⟹ *x* ⊑ *u*) ⟹ *S* <| *u*
by (*simp add*: *is-ub-def*)

lemma *is-ubD*: ⟦*S* <| *u*; *x* ∈ *S*⟧ ⟹ *x* ⊑ *u*
by (*simp add*: *is-ub-def*)

lemma *ub-imageI*: (⋀*x*. *x* ∈ *S* ⟹ *f x* ⊑ *u*) ⟹ (λ*x*. *f x*) ‘ *S* <| *u*
unfolding *is-ub-def* **by** *fast*

lemma *ub-imageD*: ⟦*f* ‘ *S* <| *u*; *x* ∈ *S*⟧ ⟹ *f x* ⊑ *u*
unfolding *is-ub-def* **by** *fast*

lemma *ub-rangeI*: (⋀*i*. *S i* ⊑ *x*) ⟹ *range S* <| *x*
unfolding *is-ub-def* **by** *fast*

lemma *ub-rangeD*: *range S* <| *x* ⟹ *S i* ⊑ *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* ⊑ *y* ∧ *A* <| *y*)
unfolding *is-ub-def* **by** *fast*

lemma *is-ub-upward*: ⟦*S* <| *x*; *x* ⊑ *y*⟧ ⟹ *S* <| *y*
unfolding *is-ub-def* **by** (*fast intro*: *trans-less*)

1.3 Least upper bounds

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

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

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

syntax (*xsymbols*)

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

translations
LUB x:*A*. *t* == *CONST lub* ((%*x*. *t*) ‘ *A*)

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

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

access to some definition as inference rule

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

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

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

lubs are unique

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

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

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

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

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

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

lemma *is-lub-bin*: *x* ⊑ *y* ⟹ {*x*, *y*} <<| *y*
by (*simp add*: *is-lub-def*)

lemma *lub-bin*: *x* ⊑ *y* ⟹ *lub* {*x*, *y*} = *y*

by (*rule is-lub-bin* [*THEN thelubI*])

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

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

1.4 Countable chains

definition
— Here we use countable chains and I prefer to code them as functions!
chain :: $(nat \Rightarrow 'a::po) \Rightarrow bool$ **where**
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 trans-less*)

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

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

technical lemmas about (least) upper bounds of chains

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

lemma *is-ub-range-shift*:
$chain\ S \Longrightarrow range\ (\lambda i.\ S\ (i + j)) <| x = range\ S <| x$
apply (*rule iffI*)
apply (*rule ub-rangeI*)
apply (*rule-tac* $y{=}S\ (i + j)$ **in** *trans-less*)
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 *lub-const*: *range* ($\lambda x.\ c$) <<| *c*
by (*blast dest*: *ub-rangeD intro*: *is-lubI ub-rangeI*)

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

1.5 Finite chains

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

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

results about finite chains

lemma *max-in-chainI*: ($\bigwedge j.\ i \leq j \Longrightarrow Y\ i = Y\ j$) ⟹ *max-in-chain i Y*
unfolding *max-in-chain-def* **by** *fast*

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

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

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

lemma *lub-finch1*: ⟦*chain C*; *max-in-chain i C*⟧ ⟹ *range C* <<| *C i*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*, *rename-tac j*)
apply (*rule-tac x*=*i* **and** *y*=*j* **in** *linorder-le-cases*)
apply (*drule* (*1*) *max-in-chainD*, *simp*)
apply (*erule* (*1*) *chain-mono*)
apply (*erule ub-rangeD*)
done

lemma *lub-finch2*:
finite-chain C ⟹ *range C* <<| *C* (*LEAST i. max-in-chain i C*)

apply (*erule finite-chainE*)
apply (*erule LeastI2* [**where** $Q{=}\lambda i.\ range\ C <<|\ C\ i$])
apply (*erule* (*1*) *lub-finch1*)
done

lemma *finch-imp-finite-range*: *finite-chain* $Y \Longrightarrow$ *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 $f :: nat \Rightarrow 'a$ **and** $r :: 'a \Rightarrow 'a \Rightarrow bool$
assumes *mono*: $\bigwedge i\ j.\ i \leq j \Longrightarrow r\ (f\ i)\ (f\ j)$
assumes *finite-range*: *finite* (*range* f)
shows $\exists k.\ \forall i.\ r\ (f\ i)\ (f\ k)$
proof (*intro exI allI*)
fix $i :: nat$
let *?j* $= LEAST\ k.\ f\ k = f\ i$
let *?k* $= Max\ ((\lambda x.\ LEAST\ k.\ f\ k = x)$ ‘ $range\ f)$
have *?j* $\leq$ *?k*
proof (*rule Max-ge*)
show *finite* $((\lambda x.\ LEAST\ k.\ f\ k = x)$ ‘ $range\ f)$
using *finite-range* **by** (*rule finite-imageI*)
show *?j* $\in (\lambda x.\ LEAST\ k.\ f\ k = x)$ ‘ *range f*
by (*intro imageI rangeI*)
qed
hence $r\ (f$ *?j*$)\ (f$ *?k*$)$
by (*rule mono*)
also have f *?j* $= f\ i$
by (*rule LeastI*, *rule refl*)
finally show $r\ (f\ i)\ (f$ *?k*$)$ **.**
qed

lemma *finite-range-imp-finch*:
$[\![$*chain* Y; *finite* (*range* Y)$]\!] \Longrightarrow$ *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 antisym-less*)
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* (λi. *if i=0 then x else y*)
by (*rule chainI*, *simp*)

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

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

the maximal element in a chain is its lub

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

1.6 Directed sets

definition
directed :: '*a*::*po set* ⇒ *bool* **where**
directed S = $((\exists x.\ x \in S) \wedge (\forall x{\in}S.\ \forall y{\in}S.\ \exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z))$

lemma *directedI*:
assumes *1*: $\exists z.\ z \in S$
assumes *2*: $\bigwedge x\ y.\ [\![x \in S; y \in S]\!] \Longrightarrow \exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$
shows *directed S*
unfolding *directed-def* **using** *prems* **by** *fast*

lemma *directedD1*: *directed S* $\Longrightarrow \exists z.\ z \in S$
unfolding *directed-def* **by** *fast*

lemma *directedD2*: $[\![$*directed S*; $x \in S$; $y \in S]\!] \Longrightarrow \exists z{\in}S.\ x \sqsubseteq z \wedge y \sqsubseteq z$
unfolding *directed-def* **by** *fast*

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

lemma *directedE2*:
assumes *S*: *directed S*

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

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

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

lemma *not-directed-empty* [*simp*]: $\neg$ *directed* $\{\}$
by (*rule notI*, *drule directedD1*, *simp*)

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

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

lemma *directed-chain*: *chain S* $\Longrightarrow$ *directed* (*range S*)

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

end

2 Pcpo: Classes cpo and pcpo

theory *Pcpo*
imports *Porder*
begin

2.1 Complete partial orders

The class cpo of chain complete partial orders

class *cpo* = *po* +
— class axiom:
assumes *cpo*: *chain S* $\Longrightarrow \exists x :: 'a{::}po.$ *range S* <<| *x*

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

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

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

Properties of the lub

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

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

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

done

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

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

the ⊑ relation between two chains is preserved by their lubs

lemma *lub-mono*:
⟦*chain* (*X*::*nat* ⇒ *'a*::*cpo*); *chain Y*; ⋀*i*. *X i* ⊑ *Y i*⟧
⟹ (⨆ *i*. *X i*) ⊑ (⨆ *i*. *Y i*)
apply (*erule is-lub-thelub*)
apply (*rule ub-rangeI*)
apply (*rule trans-less*)
apply (*erule meta-spec*)
apply (*erule is-ub-thelub*)
done

the = relation between two chains is preserved by their lubs

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

more results about mono and = of lubs of chains

lemma *lub-mono2*:

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

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

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

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

lemma diag-lub:
  fixes Y :: nat ⇒ nat ⇒ 'a::cpo
  assumes 1: ⋀j. chain (λi. Y i j)
  assumes 2: ⋀i. chain (λj. Y i j)
  shows (⨆i. ⨆j. Y i j) = (⨆i. Y i i)
proof (rule antisym-less)
  have 3: chain (λi. Y i i)
    apply (rule chainI)
    apply (rule trans-less)
    apply (rule chainE [OF 1])
    apply (rule chainE [OF 2])
    done
  have 4: chain (λi. ⨆j. Y i j)
    by (rule ch2ch-lub [OF 1 2])
```

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

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

2.2 Pointed cpos

The class pcpo of pointed cpos

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

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

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

derive the old rule minimal

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

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

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

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

Addsimprocs [UU-reorient-simproc];
⟫

useful lemmas about ⊥

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

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

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

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

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

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

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

lemma *notUU-I*: $[\![x \sqsubseteq y; x \neq \bot]\!] \Longrightarrow y \neq \bot$

by (*blast intro*: *UU-I*)

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

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

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

class *chfin* = *po* +
assumes *chfin*: $chain\ Y \Longrightarrow \exists n.\ max\text{-}in\text{-}chain\ n\ (Y :: nat => 'a{::}po)$

class *flat* = *pcpo* +
assumes *ax-flat*: $(x :: 'a{::}pcpo) \sqsubseteq y \Longrightarrow x = \bot \vee x = y$

finite partial orders are chain-finite

instance *finite-po* < *chfin*
apply *intro-classes*
apply (*drule finite-range-imp-finch*)
apply (*rule finite*)
apply (*simp add*: *finite-chain-def*)
done

some properties for chfin and flat

chfin types are cpo

instance *chfin* < *cpo*
apply *intro-classes*
apply (*frule chfin*)
apply (*blast intro*: *lub-finch1*)
done

flat types are chfin

instance *flat* < *chfin*
apply *intro-classes*
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

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

lemma *flat-less-iff*:

fixes $x\ y :: 'a{::}flat$
shows $(x \sqsubseteq y) = (x = \bot \vee x = y)$
by (*safe dest*!: *ax-flat*)

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

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

Discrete cpos

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

subclass (**in** *discrete-cpo*) *po*
proof qed *simp-all*

In a discrete cpo, every chain is constant

lemma *discrete-chain-const*:
assumes S: *chain* $(S{::}nat \Rightarrow 'a{::}discrete\text{-}cpo)$
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

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

lemmata for improved admissibility introdution rule

lemma *infinite-chain-adm-lemma*:
$[\![$*chain* Y; $\forall i.\ P\ (Y\ i)$;
$\bigwedge Y.\ [\![$*chain* Y; $\forall i.\ P\ (Y\ i)$; $\neg$ *finite-chain* $Y]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)]\!]$
$\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
apply (*case-tac finite-chain* Y)
prefer *2* **apply** *fast*
apply (*unfold finite-chain-def*)
apply *safe*
apply (*erule lub-finch1* [*THEN thelubI*, *THEN ssubst*])
apply *assumption*

apply (*erule spec*)
done

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

end

3 Cont: Continuity and monotonicity

theory *Cont*
imports *Pcpo*
begin

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

defaultsort *po*

3.1 Definitions

definition
monofun :: ('*a* ⇒ '*b*) ⇒ *bool* — monotonicity **where**
monofun f = (∀*x y*. *x* ⊑ *y* ⟶ *f x* ⊑ *f y*)

definition
contlub :: ('*a*::*cpo* ⇒ '*b*::*cpo*) ⇒ *bool* — first cont. def **where**
contlub f = (∀*Y*. *chain Y* ⟶ *f* (⨆*i*. *Y i*) = (⨆*i*. *f* (*Y i*)))

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

lemma *contlubI*:
⟦⋀*Y*. *chain Y* ⟹ *f* (⨆*i*. *Y i*) = (⨆*i*. *f* (*Y i*))⟧ ⟹ *contlub f*
by (*simp add*: *contlub-def*)

lemma *contlubE*:
⟦*contlub f*; *chain Y*⟧ ⟹ *f* (⨆*i*. *Y i*) = (⨆*i*. *f* (*Y i*))
by (*simp add*: *contlub-def*)

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

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

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

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

3.2 *monofun f* ∧ *contlub f* ≡ *cont f*

monotone functions map chains to chains

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

monotone functions map upper bound to upper bounds

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

left to right: *monofun f* ∧ *contlub f* ⟹ *cont f*

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

first a lemma about binary chains

lemma *binchain-cont*:
$[\![cont\ f;\ x \sqsubseteq y]\!] \Longrightarrow range\ (\lambda i{::}nat.\ f\ (if\ i = 0\ then\ x\ else\ y)) <<|\ f\ y$
apply (*subgoal-tac f* $(\bigsqcup i{::}nat.\ if\ i = 0\ then\ x\ else\ y) = f\ y$)
apply (*erule subst*)
apply (*erule contE*)

apply (*erule bin-chain*)
apply (*rule-tac f=f* **in** *arg-cong*)
apply (*erule lub-bin-chain* [*THEN thelubI*])
done

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

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

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

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

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

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

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

lemma *cont2contlub*: *cont f* $\Longrightarrow$ *contlub f*
apply (*rule contlubI*)
apply (*rule thelubI* [*symmetric*])
apply (*erule* (*1*) *contE*)
done

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

lemma *contI2*:
  assumes *mono*: *monofun f*
  assumes *less*: $\bigwedge Y.\ [\![chain\ Y;\ chain\ (\lambda i.\ f\ (Y\ i))]\!]$
    $\Longrightarrow f\ (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ f\ (Y\ i))$
  shows *cont f*
apply (*rule monocontlub2cont*)
apply (*rule mono*)
apply (*rule contlubI*)
apply (*rule antisym-less*)
apply (*rule less*, *assumption*)
apply (*erule ch2ch-monofun* [*OF mono*])
apply (*rule is-lub-thelub*)
apply (*erule ch2ch-monofun* [*OF mono*])
apply (*rule ub2ub-monofun* [*OF mono*])
apply (*rule is-lubD1*)
apply (*erule cpo-lubI*)
done

3.3 Continuity simproc

ML ⟪
structure Cont2ContData = NamedThmsFun
(val name = cont2cont val description = continuity intro rule)
⟫

setup ⟪ *Cont2ContData.setup* ⟫

Given the term *cont f*, the procedure tries to construct the theorem *cont f* ≡ *True*. If this theorem cannot be completely solved by the introduction rules, then the procedure returns a conditional rewrite rule with the unsolved subgoals as premises.

setup ⟪
let
fun solve-cont thy ss t =
let
val tr = instantiate' [] [SOME (cterm-of thy t)] Eq-TrueI;
val rules = Cont2ContData.get (Simplifier.the-context ss);
val tac = REPEAT-ALL-NEW (match-tac rules);
in Option.map fst (Seq.pull (tac 1 tr)) end

val proc =
Simplifier.simproc @{theory} cont-proc [cont f] solve-cont;
in
Simplifier.map-simpset (fn ss => ss addsimprocs [proc])
end
⟫

3.4 Continuity of basic functions

The identity function is continuous

lemma *cont-id* [*cont2cont*]: *cont* (λ*x*. *x*)
apply (*rule contI*)
apply (*erule cpo-lubI*)
done

constant functions are continuous

lemma *cont-const* [*cont2cont*]: *cont* (λ*x*. *c*)
apply (*rule contI*)
apply (*rule lub-const*)
done

application of functions is continuous

lemma *cont2cont-apply*:
fixes *f* :: *'a*::*cpo* ⇒ *'b*::*cpo* ⇒ *'c*::*cpo* **and** *t* :: *'a* ⇒ *'b*
assumes *f1*: ⋀*y*. *cont* (λ*x*. *f x y*)
assumes *f2*: ⋀*x*. *cont* (λ*y*. *f x y*)

assumes t: *cont* $(\lambda x.\ t\ x)$
shows *cont* $(\lambda x.\ (f\ x)\ (t\ x))$
proof (*rule monocontlub2cont* [*OF monofunI contlubI*])
fix $x\ y :: 'a$ **assume** $x \sqsubseteq y$
then show $f\ x\ (t\ x) \sqsubseteq f\ y\ (t\ y)$
by (*auto intro*: *cont2monofunE* [*OF f1*]
cont2monofunE [*OF f2*]
cont2monofunE [*OF t*]
trans-less)
next
fix $Y :: nat \Rightarrow 'a$ **assume** *chain* Y
then show $f\ (\bigsqcup i.\ Y\ i)\ (t\ (\bigsqcup i.\ Y\ i)) = (\bigsqcup i.\ f\ (Y\ i)\ (t\ (Y\ i)))$
by (*simp only*: *cont2contlubE* [*OF t*] *ch2ch-cont* [*OF t*]
cont2contlubE [*OF f1*] *ch2ch-cont* [*OF f1*]
cont2contlubE [*OF f2*] *ch2ch-cont* [*OF f2*]
diag-lub)
qed

lemma *cont2cont-compose*:
$[\![cont\ c;\ cont\ (\lambda x.\ f\ x)]\!] \Longrightarrow cont\ (\lambda x.\ c\ (f\ x))$
by (*rule cont2cont-apply* [*OF cont-const*])

if-then-else is continuous

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

3.5 Finite chains and flat pcpos

monotone functions map finite chains to finite chains

lemma *monofun-finch2finch*:
$[\![monofun\ f;\ finite\text{-}chain\ Y]\!] \Longrightarrow finite\text{-}chain\ (\lambda n.\ f\ (Y\ n))$
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*:
$[\![cont\ f;\ finite\text{-}chain\ Y]\!] \Longrightarrow finite\text{-}chain\ (\lambda n.\ f\ (Y\ n))$
by (*rule cont2mono* [*THEN monofun-finch2finch*])

lemma *chfindom-monofun2cont*: $monofun\ f \Longrightarrow cont\ (f :: 'a :: chfin \Rightarrow 'b :: cpo)$
apply (*rule monocontlub2cont*)
apply *assumption*
apply (*rule contlubI*)
apply (*frule chfin2finch*)
apply (*clarsimp simp add*: *finite-chain-def*)

apply (*subgoal-tac max-in-chain i* ($\lambda i.$ *f* (*Y i*)))
apply (*simp add*: *maxinch-is-thelub ch2ch-monofun*)
apply (*force simp add*: *max-in-chain-def*)
done

some properties of flat

lemma *flatdom-strict2mono*: *f* $\bot$ = $\bot$ $\Longrightarrow$ *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*])

functions with discrete domain

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

end

4 Adm: Admissibility and compactness

theory *Adm*
imports *Cont*
begin

defaultsort *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*)

improved admissibility introduction

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

4.2 Admissibility on chain-finite types

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

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

4.3 Admissibility of special formulae and propagation

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

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

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

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

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

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

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

lemma *adm-disj-lemma3*:
$[\![chain\ (Y{::}nat \Rightarrow {'a}{::}cpo); \forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!] \Longrightarrow$

$(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ Y\ (LEAST\ j.\ i \leq j \wedge P\ (Y\ j)))$
apply (*frule* (*1*) *adm-disj-lemma1*)
apply (*rule antisym-less*)
apply (*rule lub-mono*, *assumption+*)
apply (*erule chain-mono*)
apply (*simp add*: *adm-disj-lemma2*)
apply (*rule lub-range-mono*, *fast*, *assumption+*)
done

lemma *adm-disj-lemma4*:
$[\![adm\ P;\ chain\ Y;\ \forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
apply (*subst adm-disj-lemma3*, *assumption+*)
apply (*erule admD*)
apply (*simp add*: *adm-disj-lemma1*)
apply (*simp add*: *adm-disj-lemma2*)
done

lemma *adm-disj-lemma5*:
$\forall n{::}nat.\ P\ n \vee Q\ n \Longrightarrow (\forall i.\ \exists j{\geq}i.\ P\ j) \vee (\forall i.\ \exists j{\geq}i.\ Q\ j)$
apply (*erule contrapos-pp*)
apply (*clarsimp*, *rename-tac a b*)
apply (*rule-tac x=max a b* **in** *exI*)
apply *simp*
done

lemma *adm-disj*: $[\![adm\ P;\ adm\ Q]\!] \Longrightarrow adm\ (\lambda x.\ P\ x \vee Q\ x)$
apply (*rule admI*)
apply (*erule adm-disj-lemma5* [*THEN disjE*])
apply (*erule* (*2*) *adm-disj-lemma4* [*THEN disjI1*])
apply (*erule* (*2*) *adm-disj-lemma4* [*THEN disjI2*])
done

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

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

lemma *adm-not-conj*:
$[\![adm\ (\lambda x.\ \neg\ P\ x);\ adm\ (\lambda x.\ \neg\ Q\ x)]\!] \Longrightarrow adm\ (\lambda x.\ \neg\ (P\ x \wedge Q\ x))$
by (*simp add*: *adm-imp*)

admissibility and continuity

lemma *adm-less*: $[\![cont\ u;\ cont\ v]\!] \Longrightarrow adm\ (\lambda x.\ u\ x \sqsubseteq v\ x)$
apply (*rule admI*)
apply (*simp add*: *cont2contlubE*)
apply (*rule lub-mono*)

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

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

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

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

4.4 Compactness

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

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

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

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

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

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

lemma *compact-imp-max-in-chain*:

$[\![chain\ Y;\ compact\ (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists\, i.\ max\text{-}in\text{-}chain\ i\ Y$
apply (*drule* (*1*) *compactD2*, *simp*)
apply (*erule exE*, *rule-tac x=i* **in** *exI*)
apply (*rule max-in-chainI*)
apply (*rule antisym-less*)
apply (*erule* (*1*) *chain-mono*)
apply (*erule* (*1*) *trans-less* [*OF is-ub-thelub*])
done

admissibility and compactness

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

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

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

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

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

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

end

5 Pcpodef: Subtypes of pcpos

theory *Pcpodef*
imports *Adm*
uses (*Tools/pcpodef-package.ML*)
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.sq-le*}, *NONE*) ⟫

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

setup ⟪ *Sign.add-const-constraint* (@{*const-name Porder.sq-le*},
  SOME @{*typ 'a*::*sq-ord* ⇒ *'a*::*sq-ord* ⇒ *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

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

5.3 Proving a subtype is chain-finite

lemma *monofun-Rep*:
  assumes *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
  shows *monofun Rep*
by (*rule monofunI*, *unfold less*)

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

theorem *typedef-chfin*:
fixes *Abs* :: *'a*::*chfin* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
shows *OFCLASS*(*'b*, *chfin-class*)
apply *intro-classes*
apply (*drule ch2ch-Rep* [*OF less*])
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 *Abs-inverse-lub-Rep*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*po*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *chain S* ⟹ *Rep* (*Abs* (⨆ *i*. *Rep* (*S i*))) = (⨆ *i*. *Rep* (*S i*))
apply (*rule type-definition.Abs-inverse* [*OF type*])
apply (*erule admD* [*OF adm ch2ch-Rep* [*OF less*]])
apply (*rule type-definition.Rep* [*OF type*])
done

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

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

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

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

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

theorem *typedef-is-lubI*:
assumes *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
shows *range* (λ*i*. *Rep* (*S i*)) <<| *Rep x* ⟹ *range S* <<| *x*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*subst less*)
apply (*erule is-ub-lub*)
apply (*subst less*)
apply (*erule is-lub-lub*)
apply (*erule ub2ub-Rep* [*OF less*])
done

theorem *typedef-cont-Abs*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
fixes *f* :: *'c*::*cpo* ⇒ *'a*::*cpo*

assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: *adm* $(\lambda x.\ x \in A)$
and *f-in-A*: $\bigwedge x.\ f\ x \in A$
and *cont-f*: *cont f*
shows *cont* $(\lambda x.\ Abs\ (f\ x))$
apply (*rule contI*)
apply (*rule typedef-is-lubI* [*OF less*])
apply (*simp only*: *type-definition.Abs-inverse* [*OF type f-in-A*])
apply (*erule cont-f* [*THEN contE*])
done

5.5 Proving subtype elements are compact

theorem *typedef-compact*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: *adm* $(\lambda x.\ x \in A)$
shows *compact* $(Rep\ k) \Longrightarrow$ *compact k*
proof (*unfold compact-def*)
have *cont-Rep*: *cont Rep*
by (*rule typedef-cont-Rep* [*OF type less adm*])
assume *adm* $(\lambda x.\ \neg\ Rep\ k \sqsubseteq x)$
with *cont-Rep* **have** *adm* $(\lambda x.\ \neg\ Rep\ k \sqsubseteq Rep\ x)$ **by** (*rule adm-subst*)
thus *adm* $(\lambda x.\ \neg\ k \sqsubseteq x)$ **by** (*unfold less*)
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 *less*: *op* $\sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *z-in-A*: $z \in A$
and *z-least*: $\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x$
shows *OFCLASS*(*'b*, *pcpo-class*)
apply (*intro-classes*)
apply (*rule-tac x=Abs z* **in** *exI*, *rule allI*)
apply (*unfold less*)
apply (*subst type-definition.Abs-inverse* [*OF type z-in-A*])
apply (*rule z-least* [*OF type-definition.Rep* [*OF type*]])
done

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

theorem *typedef-pcpo*:

fixes *Abs* :: *'a*::*pcpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *OFCLASS*(*'b*, *pcpo-class*)
by (*rule typedef-pcpo-generic* [*OF type less UU-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 *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *Abs* ⊥ = ⊥
apply (*rule UU-I*, *unfold less*)
apply (*simp add*: *type-definition.Abs-inverse* [*OF type UU-in-A*])
done

theorem *typedef-Rep-strict*:
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *Rep* ⊥ = ⊥
apply (*rule typedef-Abs-strict* [*OF type less UU-in-A*, *THEN subst*])
apply (*rule type-definition.Abs-inverse* [*OF type UU-in-A*])
done

theorem *typedef-Abs-strict-iff*:
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows *x* ∈ *A* ⟹ (*Abs x* = ⊥) = (*x* = ⊥)
apply (*rule typedef-Abs-strict* [*OF type less UU-in-A*, *THEN subst*])
apply (*simp add*: *type-definition.Abs-inject* [*OF type*] *UU-in-A*)
done

theorem *typedef-Rep-strict-iff*:
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *UU-in-A*: ⊥ ∈ *A*
shows (*Rep x* = ⊥) = (*x* = ⊥)
apply (*rule typedef-Rep-strict* [*OF type less UU-in-A*, *THEN subst*])
apply (*simp add*: *type-definition.Rep-inject* [*OF type*])
done

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

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

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

5.7 Proving a subtype is flat

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

5.8 HOLCF type definition package

use *Tools/pcpodef-package.ML*

end

6 Ffun: Class instances for the full function space

theory *Ffun*
imports *Cont*
begin

6.1 Full function space is a partial order

instantiation *fun* :: (*type*, *sq-ord*) *sq-ord*
begin

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

make the symbol << accessible for type fun

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

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

6.2 Full function space is chain complete

function application is monotone

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

chains of functions yield chains in the po range

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

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

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

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

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

lemma *is-lub-lambda*:
  assumes f: $\bigwedge x.$ *range* $(\lambda i.\ Y\ i\ x) <<|\ f\ x$
  shows *range* $Y <<|\ f$
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)

apply (*rule less-fun-ext*)
apply (*rule is-ub-lub* [*OF f*])
apply (*rule less-fun-ext*)
apply (*rule is-lub-lub* [*OF f*])
apply (*erule ub2ub-fun*)
done

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

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

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

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

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

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

chain-finite function spaces

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

lemma *maxinch-mono*:
⟦*max-in-chain i Y*; *i* ≤ *j*⟧ ⟹ *max-in-chain j Y*
unfolding *max-in-chain-def*
proof (*intro allI impI*)
fix *k*
assume *Y*: ∀ *n*≥*i*. *Y i* = *Y n*
assume *ij*: *i* ≤ *j*

assume *jk*: $j \leq k$
from *ij jk* **have** *ik*: $i \leq k$ **by** *simp*
from *Y ij* **have** *Yij*: $Y\ i = Y\ j$ **by** *simp*
from *Y ik* **have** *Yik*: $Y\ i = Y\ k$ **by** *simp*
from *Yij Yik* **show** $Y\ j = Y\ k$ **by** *auto*
qed

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

6.3 Full function space is pointed

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

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

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

for compatibility with old HOLCF-Version

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

function application is strict in the left argument

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

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

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

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

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

6.4 Propagation of monotonicity and continuity

the lub of a chain of monotone functions is monotone

lemma *monofun-lub-fun*:
$[\![chain\ (F{::}nat \Rightarrow 'a \Rightarrow 'b{::}cpo);\ \forall i.\ monofun\ (F\ i)]\!]$
$\Longrightarrow monofun\ (\bigsqcup i.\ F\ i)$
apply (*rule monofunI*)
apply (*simp add*: *thelub-fun*)
apply (*rule lub-mono*)
apply (*erule ch2ch-fun*)
apply (*erule ch2ch-fun*)
apply (*simp add*: *monofunE*)
done

the lub of a chain of continuous functions is continuous

lemma *contlub-lub-fun*:
$[\![chain\ F;\ \forall i.\ cont\ (F\ i)]\!] \Longrightarrow contlub\ (\bigsqcup i.\ F\ i)$
apply (*rule contlubI*)
apply (*simp add*: *thelub-fun*)
apply (*simp add*: *cont2contlubE*)
apply (*rule ex-lub*)
apply (*erule ch2ch-fun*)
apply (*simp add*: *ch2ch-cont*)
done

lemma *cont-lub-fun*:
$[\![chain\ F;\ \forall i.\ cont\ (F\ i)]\!] \Longrightarrow cont\ (\bigsqcup i.\ F\ i)$
apply (*rule monocontlub2cont*)
apply (*erule monofun-lub-fun*)
apply (*simp add*: *cont2mono*)
apply (*erule* (*1*) *contlub-lub-fun*)
done

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

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

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

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

lemma *mono2mono-lambda*:
assumes *f*: $\bigwedge y.$ *monofun* $(\lambda x.\ f\ x\ y)$ **shows** *monofun f*
apply (*rule monofunI*)
apply (*rule less-fun-ext*)
apply (*erule monofunE* [*OF f*])
done

lemma *cont2cont-lambda* [*simp*]:
assumes *f*: $\bigwedge y.$ *cont* $(\lambda x.\ f\ x\ y)$ **shows** *cont f*
apply (*subgoal-tac monofun f*)
apply (*rule monocontlub2cont*)
apply *assumption*
apply (*rule contlubI*)
apply (*rule ext*)
apply (*simp add*: *thelub-fun ch2ch-monofun*)
apply (*erule cont2contlubE* [*OF f*])
apply (*simp add*: *mono2mono-lambda cont2mono f*)
done

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

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

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

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

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

apply (*rule contlubI*)
apply (*subgoal-tac chain* (λ*i*. *f* (*Y i*)))
apply (*subgoal-tac chain* (λ*i*. *t* (*Y i*)))
apply (*simp add*: *cont2contlubE thelub-fun*)
apply (*rule diag-lub*)
apply (*erule ch2ch-fun*)
apply (*drule spec*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule* (*1*) *ch2ch-cont*)
done

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

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

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

end

7 Product-Cpo: The cpo of cartesian products

theory *Product-Cpo*
imports *Adm*
begin

defaultsort *cpo*

7.1 Type *unit* is a pcpo

instantiation *unit* :: *sq-ord*
begin

definition
less-unit-def [*simp*]: *x* ⊑ (*y*::*unit*) ≡ *True*

instance ..
end

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

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

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

7.2 Product type is a partial order

instantiation $*$ :: (*sq-ord*, *sq-ord*) *sq-ord*
begin

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

instance ..
end

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

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

lemma *prod-lessI*: $[\![fst\ p \sqsubseteq fst\ q;\ snd\ p \sqsubseteq snd\ q]\!] \Longrightarrow p \sqsubseteq q$
unfolding *less-cprod-def* **by** *simp*

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

Pair (-,-) is monotone in both arguments

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

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

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

fst and *snd* are monotone

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

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

7.4 Product type is a cpo

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

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

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

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

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

instance $*$ :: (*discrete-cpo*, *discrete-cpo*) *discrete-cpo*
proof
  fix x y :: $'a \times 'b$
  show $x \sqsubseteq y \longleftrightarrow x = y$
    unfolding *less-cprod-def Pair-fst-snd-eq*
    by *simp*
qed

7.5 Product type is pointed

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

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

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

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

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

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

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

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

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

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

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

done

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

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

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

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

lemma *cont2cont-Pair* [*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 cont2cont-apply* [*OF - cont-pair1 f*])
apply (*rule cont2cont-apply* [*OF - cont-pair2 g*])
apply (*rule cont-const*)
done

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

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

7.7 Compactness and chain-finiteness

lemma *fst-less-iff*: *fst* (*x*::′*a* × ′*b*) ⊑ *y* ⟷ *x* ⊑ (*y*, *snd x*)
unfolding *less-cprod-def* **by** *simp*

lemma *snd-less-iff*: *snd* (*x*::′*a* × ′*b*) ⊑ *y* = *x* ⊑ (*fst x*, *y*)
unfolding *less-cprod-def* **by** *simp*

lemma *compact-fst*: *compact x* ⟹ *compact* (*fst x*)
by (*rule compactI*, *simp add*: *fst-less-iff*)

lemma *compact-snd*: *compact x* ⟹ *compact* (*snd x*)

by (*rule compactI*, *simp add*: *snd-less-iff*)

lemma *compact-Pair*: ⟦*compact x*; *compact y*⟧ ⟹ *compact* (*x*, *y*)
by (*rule compactI*, *simp add*: *less-cprod-def*)

lemma *compact-Pair-iff* [*simp*]: *compact* (*x*, *y*) ⟷ *compact x* ∧ *compact y*
apply (*safe intro*!: *compact-Pair*)
apply (*drule compact-fst*, *simp*)
apply (*drule compact-snd*, *simp*)
done

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

end

8 Cfun: The type of continuous functions

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

defaultsort *cpo*

8.1 Definition of continuous function type

lemma *Ex-cont*: ∃*f*. *cont f*
by (*rule exI*, *rule cont-const*)

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

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

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

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

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

notation (*HTML* **output**)

```
Rep-CFun  ((-·/-) [999,1000] 999)
```

8.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: *'a*

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

To avoid eta-contraction of body:

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

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

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

Syntax for nested abstractions

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

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

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

print-ast-translation ⟪
```
(* rewrites (-cabs x (-cabs y (-cabs z t))) => (LAM x y z. t) *)
(* cf. Syntax.abs-ast-tr' from Syntax/syn-trans.ML *)
  let
    fun cabs-ast-tr' asts =
```

```
    (case Syntax.unfold-ast-p -cabs
        (Syntax.Appl (Syntax.Constant -cabs :: asts)) of
      ([], -) => raise Syntax.AST (cabs-ast-tr', asts)
    | (xs, body) => Syntax.Appl
        [Syntax.Constant -Lambda, Syntax.fold-ast -cargs xs, body]);
  in [(-cabs, cabs-ast-tr')] end;
⟫
```

Dummy patterns for continuous abstraction

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

8.3 Continuous function space is pointed

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

instance −> :: (*finite-po*, *finite-po*) *finite-po*
by (*rule typedef-finite-po* [*OF type-definition-CFun*])

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

instance −> :: (*cpo*, *discrete-cpo*) *discrete-cpo*
by *intro-classes* (*simp add*: *less-CFun-def Rep-CFun-inject*)

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

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

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

function application is strict in its first argument

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

for compatibility with old HOLCF-Version

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

8.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

Eta-equality for continuous functions

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

Extensionality for continuous functions

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

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

Extensionality wrt. ordering for continuous functions

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

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

Congruence for continuous function application

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

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

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

8.5 Continuity of application

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

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

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

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

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

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

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

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

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

monotonicity of application

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

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

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

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

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

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

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

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

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

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

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

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

apply (*rule thelubE*)
apply (*simp only*: *ch2ch-Rep-CFun*)
apply (*simp only*: *contlub-cfun*)
done

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

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

strictness

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

the lub of a chain of continous functions is monotone

lemma *lub-cfun-mono*: *chain F* $\Longrightarrow$ *monofun* $(\lambda x.\ \bigsqcup i.\ F\ i{\cdot}x)$
apply (*drule ch2ch-monofun* [*OF monofun-Rep-CFun*])
apply (*simp add*: *thelub-fun* [*symmetric*])
apply (*erule monofun-lub-fun*)
apply (*simp add*: *monofun-Rep-CFun2*)
done

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

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

the lub of a chain of cont. functions is continuous

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

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

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

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

8.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

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

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

lemma *cont2mono-LAM*:
  $[\![\bigwedge x.$ *cont* $(\lambda y.\ f\ x\ y);\ \bigwedge y.$ *monofun* $(\lambda x.\ f\ x\ y)]\!]$
    $\Longrightarrow$ *monofun* $(\lambda x.\ \Lambda\ y.\ f\ x\ y)$
  unfolding *monofun-def expand-cfun-less* **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′* [*cont2cont*]:
  fixes *f* :: $'a$::*cpo* $\Rightarrow$ $'b$::*cpo* $\Rightarrow$ $'c$::*cpo*
  assumes *f*: *cont* $(\lambda p.\ f\ (fst\ p)\ (snd\ p))$
  shows *cont* $(\lambda x.\ \Lambda\ y.\ f\ x\ y)$
proof (*rule cont2cont-LAM*)
  fix *x* :: $'a$
  have *cont* $(\lambda y.\ (x,\ y))$
    by (*rule cont-pair2*)
  with *f* **have** *cont* $(\lambda y.\ f\ (fst\ (x,\ y))\ (snd\ (x,\ y)))$
    by (*rule cont2cont-app3*)

thus *cont* ($\lambda y.\ f\ x\ y$)
by (*simp only*: *fst-conv snd-conv*)
next
fix $y :: 'b$
have *cont* ($\lambda x.\ (x,\ y)$)
by (*rule cont-pair1*)
with *f* **have** *cont* ($\lambda x.\ f\ (fst\ (x,\ y))\ (snd\ (x,\ y))$)
by (*rule cont2cont-app3*)
thus *cont* ($\lambda x.\ f\ x\ y$)
by (*simp only*: *fst-conv snd-conv*)
qed

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

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

8.7 Miscellaneous

Monotonicity of *Abs-CFun*

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

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

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

lemma *adm-chfindom*: *adm* ($\lambda(u::'a::cpo \to 'b::chfin).\ P(u{\cdot}s)$)
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 UU-I*)
apply (*drule-tac* $x{=}\bot$ **in** *spec*)
apply (*erule subst*)
apply (*rule monofun-cfun-arg*)
apply (*rule minimal*)

done

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

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

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

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

propagation of flatness and chain-finiteness by retractions

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

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

a result about functions with flat codomain

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

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

8.9 Identity and composition

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

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

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

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

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

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

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

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

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

The composition of f and g is interpretation of *oo*.

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

lemma *ID3* [*simp*]: $\mathit{ID} \mathit{oo} f = f$
by (*rule ext-cfun*, *simp*)

lemma *assoc-oo*: $f \mathit{oo} (g \mathit{oo} h) = (f \mathit{oo} g) \mathit{oo} h$
by (*rule ext-cfun*, *simp*)

8.10 Strictified functions

defaultsort *pcpo*

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

results about strictify

lemma *cont-strictify1*: $\mathit{cont}\ (\lambda f.\ \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
by (*simp add*: *cont-if*)

lemma *monofun-strictify2*: $\mathit{monofun}\ (\lambda x.\ \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
apply (*rule monofunI*)
apply (*auto simp add*: *monofun-cfun-arg*)
done

lemma *contlub-strictify2*: $\mathit{contlub}\ (\lambda x.\ \mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
apply (*rule contlubI*)
apply (*case-tac* $(\bigsqcup i.\ Y\ i) = \bot$)
apply (*drule* (*1*) *chain-UU-I*)
apply *simp*
apply (*simp del*: *if-image-distrib*)
apply (*simp only*: *contlub-cfun-arg*)
apply (*rule lub-equal2*)
apply (*rule chain-mono2* [*THEN exE*])
apply (*erule chain-UU-I-inverse2*)
apply (*assumption*)
apply (*rule-tac x=x* **in** *exI*, *clarsimp*)
apply (*erule chain-monofun*)
apply (*erule monofun-strictify2* [*THEN ch2ch-monofun*])
done

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

lemma *strictify-conv-if*: $\mathit{strictify}{\cdot}f{\cdot}x = (\mathit{if}\ x = \bot\ \mathit{then}\ \bot\ \mathit{else}\ f{\cdot}x)$
unfolding *strictify-def*

by (*simp add*: *cont-strictify1 cont-strictify2 cont2cont-LAM*)

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

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

8.11 Continuous let-bindings

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

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

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

end

9 Deflation: Continuous Deflations and Embedding-Projection Pairs

theory *Deflation*
imports *Cfun*
begin

defaultsort *cpo*

9.1 Continuous deflations

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

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

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

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

lemma *range-eq-fixes*: *range* $(\lambda x.\ d{\cdot}x) = \{x.\ d{\cdot}x = x\}$
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 *lessI*:
  assumes *f*: $\bigwedge x.\ d{\cdot}x = x \Longrightarrow f{\cdot}x = x$ **shows** $d \sqsubseteq f$
proof (*rule less-cfun-ext*)
  fix *x*
  from *less* **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 *lessD*: $[\![f \sqsubseteq d;\ f{\cdot}x = x]\!] \Longrightarrow d{\cdot}x = x$
proof (*rule antisym-less*)
  from *less* **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 *adm-deflation*: *adm* $(\lambda d.\ \mathit{deflation}\ d)$
by (*simp add*: *deflation-def*)

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

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

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

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

lemma *deflation-less-comp1*:
  assumes *deflation f*
  assumes *deflation g*
  shows $f \sqsubseteq g \Longrightarrow f{\cdot}(g{\cdot}x) = f{\cdot}x$
proof (*rule antisym-less*)

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

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

9.2 Deflations with finite range

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

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

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

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

lemma *compact*: *compact* $(d{\cdot}x)$
proof (*rule compactI2*)
fix $Y :: nat \Rightarrow 'a$
assume Y: *chain* Y
have *finite-chain* $(\lambda i.\ d{\cdot}(Y\ i))$
proof (*rule finite-range-imp-finch*)
show *chain* $(\lambda i.\ d{\cdot}(Y\ i))$
using Y **by** *simp*
have *range* $(\lambda i.\ d{\cdot}(Y\ i)) \subseteq$ *range* $(\lambda x.\ d{\cdot}x)$
by *clarsimp*
thus *finite* (*range* $(\lambda i.\ d{\cdot}(Y\ i))$)
using *finite-range* **by** (*rule finite-subset*)
qed

hence $\exists j.\ (\bigsqcup i.\ d{\cdot}(Y\ i)) = d{\cdot}(Y\ j)$
by (*simp add*: *finite-chain-def maxinch-is-thelub* Y)
then obtain j **where** j: $(\bigsqcup i.\ d{\cdot}(Y\ i)) = d{\cdot}(Y\ j)$ **..**

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

end

9.3 Continuous embedding-projection pairs

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

lemma *e-less-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-less-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-less-iff-less-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-less* **by** (*rule trans-less*)
qed

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

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

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-less-iff-less-p less*)
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-less-iff-less-p less)
  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.less)
    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-less = this
  show ?thesis
  proof
    fix x
    show (p oo d oo e)·x ⊑ x
      by (rule p-d-e-less)
  next
    fix x
    show (p oo d oo e)·((p oo d oo e)·x) = (p oo d oo e)·x
    proof (rule antisym-less)
      show (p oo d oo e)·((p oo d oo e)·x) ⊑ (p oo d oo e)·x
        by (rule p-d-e-less)
      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*: $\bigwedge x.\ d{\cdot}x \sqsubseteq e{\cdot}(p{\cdot}x)$
shows *finite-deflation* (*p oo d oo e*)
proof −
interpret *d*: *finite-deflation d* **by** *fact*
show *?thesis*
proof (*intro-locales*)
have *deflation d* **..**
thus *deflation* (*p oo d oo e*)
using *d* **by** (*rule deflation-p-d-e*)
next
show *finite-deflation-axioms* (*p oo d oo e*)
proof
have *finite* ((λ*x*. *d*·*x*) ‘ *range* (λ*x*. *e*·*x*))
by (*rule d.finite-image*)
hence *finite* ((λ*x*. *p*·*x*) ‘ (λ*x*. *d*·*x*) ‘ *range* (λ*x*. *e*·*x*))
by (*rule finite-imageI*)
hence *finite* (*range* (λ*x*. (*p oo d oo e*)·*x*))
by (*simp add*: *image-image*)
thus *finite* {*x*. (*p oo d oo e*)·*x* = *x*}
by (*rule finite-range-imp-finite-fixes*)
qed
qed
qed

end

9.4 Uniqueness of ep-pairs

lemma *ep-pair-unique-e-lemma*:
assumes *ep-pair e1 p* **and** *ep-pair e2 p*
shows *e1* ⊑ *e2*
proof (*rule less-cfun-ext*)
interpret *e1*: *ep-pair e1 p* **by** *fact*
interpret *e2*: *ep-pair e2 p* **by** *fact*
fix *x*
have *e1*·(*p*·(*e2*·*x*)) ⊑ *e2*·*x*
by (*rule e1.e-p-less*)
thus *e1*·*x* ⊑ *e2*·*x*
by (*simp only*: *e2.e-inverse*)
qed

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

lemma *ep-pair-unique-p-lemma*:
assumes *ep-pair e p1* **and** *ep-pair e p2*
shows $p1 \sqsubseteq p2$
proof (*rule less-cfun-ext*)
interpret *p1*: *ep-pair e p1* **by** *fact*
interpret *p2*: *ep-pair e p2* **by** *fact*
fix x
have $e{\cdot}(p1{\cdot}x) \sqsubseteq x$
by (*rule p1.e-p-less*)
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*: *p2.e-inverse*)
qed

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

9.5 Composing ep-pairs

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

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

locale *pcpo-ep-pair* = *ep-pair* +
constrains $e :: {'a}{::}pcpo \to {'b}{::}pcpo$
constrains $p :: {'b}{::}pcpo \to {'a}{::}pcpo$
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-less*)
  finally show $e{\cdot}\bot = \bot$ **by** *simp*
qed

lemma *e-defined-iff* [*simp*]: $e{\cdot}x = \bot \longleftrightarrow x = \bot$
by (*rule e-eq-iff* [**where** $y{=}\bot$, *unfolded e-strict*])

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

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

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

end

end

10 Bifinite: Bifinite domains and approximation

theory *Bifinite*
imports *Deflation*
begin

10.1 Omega-profinite and bifinite domains

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

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

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

lemma *finite-deflation-approx*: *finite-deflation* (*approx i*)
proof

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

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

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

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

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

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

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

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

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

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

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

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

by (*rule approx.compact*)

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

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

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

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

10.2 Instance for continuous function space

lemma *finite-range-cfun-lemma*:
assumes *a*: *finite* (*range* $(\lambda x.\ a{\cdot}x)$)
assumes *b*: *finite* (*range* $(\lambda y.\ b{\cdot}y)$)
shows *finite* (*range* $(\lambda f.\ \Lambda\ x.\ b{\cdot}(f{\cdot}(a{\cdot}x))))$ (**is** *finite* (*range ?h*))
proof (*rule finite-imageD*)
let *?f* = $\lambda g.$ *range* $(\lambda x.\ (a{\cdot}x,\ g{\cdot}x))$
show *finite* (*?f* ‘ *range ?h*)
proof (*rule finite-subset*)
let *?B* = *Pow* (*range* $(\lambda x.\ a{\cdot}x) \times$ *range* $(\lambda y.\ b{\cdot}y)$)
show *?f* ‘ *range ?h* $\subseteq$ *?B*
by *clarsimp*
show *finite ?B*
by (*simp add*: *a b*)
qed
show *inj-on ?f* (*range ?h*)
proof (*rule inj-onI*, *rule ext-cfun*, *clarsimp*)
fix *x f g*
assume *range* $(\lambda x.\ (a{\cdot}x,\ b{\cdot}(f{\cdot}(a{\cdot}x)))) =$ *range* $(\lambda x.\ (a{\cdot}x,\ b{\cdot}(g{\cdot}(a{\cdot}x))))$
hence *range* $(\lambda x.\ (a{\cdot}x,\ b{\cdot}(f{\cdot}(a{\cdot}x)))) \subseteq$ *range* $(\lambda x.\ (a{\cdot}x,\ b{\cdot}(g{\cdot}(a{\cdot}x))))$
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

instantiation -> :: (profinite, profinite) profinite
begin

definition
  approx-cfun-def:
    approx = (λn. Λ f x. approx n·(f·(approx n·x)))

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

end

instance -> :: (profinite, bifinite) bifinite ..

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

end
```

11 Cprod: The cpo of cartesian products

theory *Cprod*
imports *Bifinite*
begin

defaultsort *cpo*

11.1 Type *unit* is a pcpo

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

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

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

11.2 Continuous versions of constants

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

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

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

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

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

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

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

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

11.3 Convert all lemmas to the continuous versions

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

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

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

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

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

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

lemma *cpair-strict* [*simp*]: $\langle \bot,\ \bot \rangle = \bot$
by *simp*

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

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

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

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

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

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

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

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

lemma *cpair-cfst-csnd*: $\langle cfst{\cdot}p,\ csnd{\cdot}p \rangle = p$
by (*cases p rule*: *cprodE*, *simp*)

lemmas *surjective-pairing-Cprod2 = cpair-cfst-csnd*

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

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

lemma *cfst-less-iff*: $cfst{\cdot}x \sqsubseteq y = x \sqsubseteq <y,\ csnd{\cdot}x>$
by (*simp add*: *less-cprod*)

lemma *csnd-less-iff*: $csnd{\cdot}x \sqsubseteq y = x \sqsubseteq <cfst{\cdot}x,\ y>$
by (*simp add*: *less-cprod*)

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

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

lemma *compact-cpair*: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact\ <x,\ y>$
by (*simp add*: *cpair-eq-pair*)

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

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

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

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

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

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

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

11.4 Product type is a bifinite domain

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

definition
approx-cprod-def:
$approx = (\lambda n.\ \Lambda\langle x,\ y\rangle.\ \langle approx\ n{\cdot}x,\ approx\ n{\cdot}y\rangle)$

instance proof
fix i :: *nat* **and** x :: $'a \times 'b$
show *chain* ($approx$ :: $nat \Rightarrow 'a \times 'b \to 'a \times 'b$)
unfolding *approx-cprod-def* **by** *simp*
show $(\bigsqcup i.\ approx\ i{\cdot}x) = x$
unfolding *approx-cprod-def*
by (*simp add*: *lub-distribs eta-cfun*)
show $approx\ i{\cdot}(approx\ i{\cdot}x) = approx\ i{\cdot}x$
unfolding *approx-cprod-def csplit-def* **by** *simp*
have $\{x{::}'a \times 'b.\ approx\ i{\cdot}x = x\} \subseteq$
$\{x{::}'a.\ approx\ i{\cdot}x = x\} \times \{x{::}'b.\ approx\ i{\cdot}x = x\}$
unfolding *approx-cprod-def*
by (*clarsimp simp add*: *pair-eq-cpair*)
thus *finite* $\{x{::}'a \times 'b.\ approx\ i{\cdot}x = x\}$
by (*rule finite-subset*,
intro finite-cartesian-product finite-fixes-approx)
qed

end

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

lemma *approx-cpair* [*simp*]:
$approx\ i{\cdot}\langle x,\ y\rangle = \langle approx\ i{\cdot}x,\ approx\ i{\cdot}y\rangle$
unfolding *approx-cprod-def* **by** *simp*

lemma *cfst-approx*: $cfst{\cdot}(approx\ i{\cdot}p) = approx\ i{\cdot}(cfst{\cdot}p)$
by (*cases p rule*: *cprodE*, *simp*)

lemma *csnd-approx*: $csnd{\cdot}(approx\ i{\cdot}p) = approx\ i{\cdot}(csnd{\cdot}p)$
by (*cases p rule*: *cprodE*, *simp*)

end

12 Discrete: Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

12.1 Type *'a discr* is a discrete cpo

instantiation *discr :: (type) sq-ord*
begin

definition
less-discr-def:
(op ⊑ :: 'a discr ⇒ 'a discr ⇒ bool) = (op =)

instance ..
end

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

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

12.2 Type *'a discr* is a cpo

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

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

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

instance *discr :: (type) chfin*
apply *intro-classes*
apply *(rule-tac x=0 in exI)*
apply *(unfold max-in-chain-def)*

apply (*clarify*, *erule discr-chain0* [*symmetric*])
done

12.3 *undiscr*

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

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

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

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

lemma *cont-discr* [*iff*]: *cont* (%*x*::('*a*::*type*)*discr*. *f x*)
by (*rule cont-discrete-cpo*)

end

13 Up: The type of lifted values

theory *Up*
imports *Bifinite*
begin

defaultsort *cpo*

13.1 Definition of new type for lifting

datatype '*a u* = *Ibottom* | *Iup* '*a*

syntax (*xsymbols*)
u :: *type* ⇒ *type* ((-⊥) [*1000*] *999*)

consts
Ifup :: ('*a* → '*b*::*pcpo*) ⇒ '*a u* ⇒ '*b*

primrec
Ifup f Ibottom = ⊥
Ifup f (*Iup x*) = *f*·*x*

13.2 Ordering on lifted cpo

instantiation $u :: (cpo)\ sq\text{-}ord$
begin

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

instance ..
end

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

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

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

13.3 Lifted cpo is a partial order

instance $u :: (cpo)\ po$
proof
fix $x :: 'a\ u$
show $x \sqsubseteq x$
unfolding *less-up-def* **by** (*simp split*: *u.split*)
next
fix $x\ y :: 'a\ u$
assume $x \sqsubseteq y\ y \sqsubseteq x$ **thus** $x = y$
unfolding *less-up-def*
by (*auto split*: *u.split-asm intro*: *antisym-less*)
next
fix $x\ y\ z :: 'a\ u$
assume $x \sqsubseteq y\ y \sqsubseteq z$ **thus** $x \sqsubseteq z$
unfolding *less-up-def*
by (*auto split*: *u.split-asm intro*: *trans-less*)
qed

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

instance $u :: (finite\text{-}po)\ finite\text{-}po$
by (*intro-classes*, *simp add*: *u-UNIV*)

13.4 Lifted cpo is a cpo

lemma *is-lub-Iup*:

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

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

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

lemma *up-lemma2*:
⟦chain Y; Y j ≠ Ibottom⟧ ⟹ Y (i + j) ≠ Ibottom
apply (*erule contrapos-nn*)
apply (*drule-tac i=j* **and** *j=i + j* **in** *chain-mono*)
apply (*rule le-add2*)
apply (*case-tac Y j*)
apply *assumption*
apply *simp*
done

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

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

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

lemma *up-lemma6*:
⟦chain Y; Y j ≠ Ibottom⟧
⟹ range Y <<| Iup (⨆ i. THE a. Iup a = Y(i + j))

apply (*rule-tac j1 = j* **in** *is-lub-range-shift* [*THEN iffD1*])
apply *assumption*
apply (*subst up-lemma5*, *assumption+*)
apply (*rule is-lub-Iup*)
apply (*rule cpo-lubI*)
apply (*erule* (*1*) *up-lemma4*)
done

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

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

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

13.5 Lifted cpo is pointed

lemma *least-up*: $\exists x$::*'a u*. $\forall y$. *x* $\sqsubseteq$ *y*
apply (*rule-tac x* = *Ibottom* **in** *exI*)
apply (*rule minimal-up* [*THEN allI*])
done

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

for compatibility with old HOLCF-Version

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

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

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

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

13.7 Continuous versions of constants

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

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

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

continuous versions of lemmas for $'a_\perp$

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

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

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

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

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

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

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

lifting preserves chain-finiteness

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

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

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

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

instance $u :: (chfin)\ chfin$

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

properties of fup

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

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

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

13.8 Lifted cpo is a bifinite domain

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

definition
approx-up-def:
approx = ($\lambda n.$ *fup*·(Λ *x*. *up*·(*approx n*·*x*)))

instance proof
fix *i* :: *nat* **and** *x* :: *'a u*
show *chain* (*approx* :: *nat* ⇒ *'a u* → *'a u*)
unfolding *approx-up-def* **by** *simp*
show ($\bigsqcup i.$ *approx i*·*x*) = *x*
unfolding *approx-up-def*
by (*simp add*: *lub-distribs eta-cfun*)
show *approx i*·(*approx i*·*x*) = *approx i*·*x*
unfolding *approx-up-def*
by (*induct x*, *simp*, *simp*)
have {*x*::*'a u*. *approx i*·*x* = *x*} ⊆
insert $\bot$ (($\lambda x.$ *up*·*x*) ' {*x*::*'a*. *approx i*·*x* = *x*})
unfolding *approx-up-def*
by (*rule subsetI*, *case-tac x*, *simp-all*)
thus *finite* {*x*::*'a u*. *approx i*·*x* = *x*}
by (*rule finite-subset*, *simp add*: *finite-fixes-approx*)
qed

end

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

end

14 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports
~~/*src*/*HOL*/*List*
~~/*src*/*HOL*/*Hilbert-Choice*
~~/*src*/*HOL*/*Nat-Int-Bij*
~~/*src*/*HOL*/*Rational*
Main
begin

14.1 The class of countable types

class *countable* =
assumes *ex-inj*: $\exists$ *to-nat* :: $'a \Rightarrow$ *nat*. *inj to-nat*

lemma *countable-classI*:
fixes *f* :: $'a \Rightarrow$ *nat*
assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
shows *OFCLASS*($'a$, *countable-class*)
proof (*intro-classes*, *rule exI*)
show *inj f*
by (*rule injI* [*OF assms*]) *assumption*
qed

14.2 Conversion functions

definition *to-nat* :: $'a$::*countable* $\Rightarrow$ *nat* **where**
to-nat = (*SOME f*. *inj f*)

definition *from-nat* :: *nat* $\Rightarrow$ $'a$::*countable* **where**
from-nat = *inv* (*to-nat* :: $'a \Rightarrow$ *nat*)

lemma *inj-to-nat* [*simp*]: *inj to-nat*
by (*rule exE-some* [*OF ex-inj*]) (*simp add*: *to-nat-def*)

lemma *surj-from-nat* [*simp*]: *surj from-nat*
unfolding *from-nat-def* **by** (*simp add*: *inj-imp-surj-inv*)

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* $\longleftrightarrow$ *x* = *y*
using *injD* [*OF inj-to-nat*] **by** *auto*

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat x*) = *x*
by (*simp add*: *from-nat-def*)

14.3 Countable types

instance *nat* :: *countable*

by (*rule countable-classI* [*of id*]) *simp*

subclass (**in** *finite*) *countable*
proof
have *finite* (*UNIV*::*'a set*) **by** (*rule finite-UNIV*)
with *finite-conv-nat-seg-image* [*of UNIV*]
obtain *n* **and** *f* :: *nat* ⇒ *'a*
where *UNIV* = *f* ' {*i*. *i* < *n*} **by** *auto*
then have *surj f* **unfolding** *surj-def* **by** *auto*
then have *inj* (*inv f*) **by** (*rule surj-imp-inj-inv*)
then show ∃ *to-nat* :: *'a* ⇒ *nat*. *inj to-nat* **by** (*rule exI*[*of inj*])
qed

Pairs

primrec *sum* :: *nat* ⇒ *nat*
where
sum 0 = 0
| *sum* (*Suc n*) = *Suc n* + *sum n*

lemma *sum-arith*: *sum n* = *n* ∗ *Suc n div* 2
by (*induct n*) *auto*

lemma *sum-mono*: *n* ≥ *m* ⟹ *sum n* ≥ *sum m*
by (*induct n m rule*: *diff-induct*) *auto*

definition
pair-encode = (λ(*m*, *n*). *sum* (*m* + *n*) + *m*)

lemma *inj-pair-cencode*: *inj pair-encode*
unfolding *pair-encode-def*
proof (*rule injI*, *simp only*: *split-paired-all split-conv*)
fix *a b c d*
assume *eq*: *sum* (*a* + *b*) + *a* = *sum* (*c* + *d*) + *c*
have *a* + *b* = *c* + *d* ∨ *a* + *b* ≥ *Suc* (*c* + *d*) ∨ *c* + *d* ≥ *Suc* (*a* + *b*) **by** *arith*
then
show (*a*, *b*) = (*c*, *d*)
proof (*elim disjE*)
assume *sumeq*: *a* + *b* = *c* + *d*
then have *a* = *c* **using** *eq* **by** *auto*
moreover from *sumeq this* **have** *b* = *d* **by** *auto*
ultimately show *?thesis* **by** *simp*
next
assume *a* + *b* ≥ *Suc* (*c* + *d*)
from *sum-mono*[*OF this*] *eq*
show *?thesis* **by** *auto*
next
assume *c* + *d* ≥ *Suc* (*a* + *b*)
from *sum-mono*[*OF this*] *eq*
show *?thesis* **by** *auto*

```
  qed
qed

instance ∗ :: (countable, countable) countable
by (rule countable-classI [of λ(x, y). pair-encode (to-nat x, to-nat y)])
  (auto dest: injD [OF inj-pair-cencode] injD [OF inj-to-nat])
```

Sums

```
instance +:: (countable, countable) countable
  by (rule countable-classI [of (λx. case x of Inl a ⇒ to-nat (False, to-nat a)
                                  | Inr b ⇒ to-nat (True, to-nat b))])
    (auto split:sum.splits)
```

Integers

```
lemma int-cases: (i::int) = 0 ∨ i < 0 ∨ i > 0
by presburger

lemma int-pos-neg-zero:
  obtains (zero) (z::int) = 0 sgn z = 0 abs z = 0
  | (pos) n where z = of-nat n sgn z = 1 abs z = of-nat n
  | (neg) n where z = − (of-nat n) sgn z = −1 abs z = of-nat n
apply atomize-elim
apply (insert int-cases[of z])
apply (auto simp:zsgn-def)
apply (rule-tac x=nat (−z) in exI, simp)
apply (rule-tac x=nat z in exI, simp)
done

instance int :: countable
proof (rule countable-classI [of (λi. to-nat (nat (sgn i + 1), nat (abs i)))],
    auto dest: injD [OF inj-to-nat])
  fix x y
  assume a: nat (sgn x + 1) = nat (sgn y + 1) nat (abs x) = nat (abs y)
  show x = y
  proof (cases rule: int-pos-neg-zero[of x])
    case zero
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  next
    case (pos n)
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  next
    case (neg n)
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  qed
qed
```

Options

```
instance option :: (countable) countable
by (rule countable-classI[of λx. case x of None ⇒ 0
```

```
                                  | Some y ⇒ Suc (to-nat y)])
(auto split:option.splits)
```

Lists

```
lemma from-nat-to-nat-map [simp]: map from-nat (map to-nat xs) = xs
  by (simp add: comp-def map-compose [symmetric])

primrec
  list-encode :: 'a::countable list ⇒ nat
where
  list-encode [] = 0
| list-encode (x#xs) = Suc (to-nat (x, list-encode xs))

instance list :: (countable) countable
proof (rule countable-classI [of list-encode])
  fix xs ys :: 'a list
  assume cenc: list-encode xs = list-encode ys
  then show xs = ys
  proof (induct xs arbitrary: ys)
    case (Nil ys)
    with cenc show ?case by (cases ys, auto)
  next
    case (Cons x xs' ys)
    thus ?case by (cases ys) auto
  qed
qed
```

Functions

```
instance fun :: (finite, countable) countable
proof
  obtain xs :: 'a list where xs: set xs = UNIV
    using finite-list [OF finite-UNIV] ..
  show ∃ to-nat::('a ⇒ 'b) ⇒ nat. inj to-nat
  proof
    show inj (λf. to-nat (map f xs))
      by (rule injI, simp add: xs expand-fun-eq)
  qed
qed
```

14.4 The Rationals are Countably Infinite

```
definition nat-to-rat-surj :: nat ⇒ rat where
nat-to-rat-surj n = (let (a,b) = nat-to-nat2 n
                     in Fract (nat-to-int-bij a) (nat-to-int-bij b))

lemma surj-nat-to-rat-surj: surj nat-to-rat-surj
unfolding surj-def
proof
  fix r::rat
```

```
  show ∃ n. r = nat-to-rat-surj n
  proof(cases r)
    fix i j assume [simp]: r = Fract i j and j ≠ 0
    have r = (let m = inv nat-to-int-bij i; n = inv nat-to-int-bij j
              in nat-to-rat-surj(nat2-to-nat (m,n)))
      using nat2-to-nat-inj surj-f-inv-f[OF surj-nat-to-int-bij]
      by(simp add:Let-def nat-to-rat-surj-def nat-to-nat2-def)
    thus ∃ n. r = nat-to-rat-surj n by(auto simp:Let-def)
  qed
qed

lemma Rats-eq-range-nat-to-rat-surj: ℚ = range nat-to-rat-surj
by (simp add: Rats-def surj-nat-to-rat-surj surj-range)

context field-char-0
begin

lemma Rats-eq-range-of-rat-o-nat-to-rat-surj:
  ℚ = range (of-rat o nat-to-rat-surj)
using surj-nat-to-rat-surj
by (auto simp: Rats-def image-def surj-def)
   (blast intro: arg-cong[where f = of-rat])

lemma surj-of-rat-nat-to-rat-surj:
  r∈ℚ ⟹ ∃ n. r = of-rat(nat-to-rat-surj n)
by(simp add: Rats-eq-range-of-rat-o-nat-to-rat-surj image-def)

end

instance rat :: countable
proof
  show ∃ to-nat::rat ⇒ nat. inj to-nat
  proof
    have surj nat-to-rat-surj
      by (rule surj-nat-to-rat-surj)
    then show inj (inv nat-to-rat-surj)
      by (rule surj-imp-inj-inv)
  qed
qed

end
```

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

```
theory Lift
imports Discrete Up Countable
begin
```

defaultsort *type*

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

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

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

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

15.1 Lift as a datatype

lemma *lift-induct*: ⟦*P* ⊥; ⋀*x*. *P* (*Def x*)⟧ ⟹ *P y*
apply (*induct y*)
apply (*rule-tac p=y* **in** *upE*)
apply (*simp add*: *Abs-lift-strict*)
apply (*case-tac x*)
apply (*simp add*: *Def-def*)
done

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

lemmas *lift-distinct1 = lift.distinct*(*1*)
lemmas *lift-distinct2 = lift.distinct*(*2*)
lemmas *Def-not-UU = lift.distinct*(*2*)
lemmas *Def-inject = lift.inject*

⊥ and *Def*

lemma *Lift-exhaust*: *x* = ⊥ ∨ (∃ *y*. *x* = *Def y*)
by (*induct x*) *simp-all*

lemma *Lift-cases*: ⟦*x* = ⊥ ⟹ *P*; ∃ *a*. *x* = *Def a* ⟹ *P*⟧ ⟹ *P*
by (*insert Lift-exhaust*) *blast*

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* ≠ ⊥ in assumptions *defined* replaces *x* by *Def a* in conclusion.

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

⟫

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

lemma *DefE2*: ⟦*x* = *Def s*; *x* = ⊥⟧ ⟹ *R*
by *simp*

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

lemma *Def-less-is-eq* [*simp*]: *Def x* ⊑ *y* ⟷ *Def x* = *y*
by (*induct y*, *simp*, *simp add*: *Def-inject-less-eq*)

15.2 Lift is flat

instance *lift* :: (*type*) *flat*
proof
fix *x y* :: *'a lift*
assume *x* ⊑ *y* **thus** *x* = ⊥ ∨ *x* = *y*
by (*induct x*) *auto*
qed

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

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

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

15.3 Further operations

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

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

15.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

lemma *cont-lift-case1*: *cont* (λ*f*. *lift-case a f x*)
apply (*induct x*)
apply *simp*
apply *simp*

apply (*rule cont-id* [*THEN cont2cont-fun*])
done

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

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

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

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

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

rewrites for *flift1*, *flift2*

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

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

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

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

lemma *flift2-defined* [*simp*]: *x* ≠ ⊥ ⟹ (*flift2 f*)·*x* ≠ ⊥

by (*erule lift-definedE*, *simp*)

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

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

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

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

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

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

15.5 Lifted countable types are bifinite

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

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

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

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

end

end
```

16 Tr: The type of lifted booleans

```
theory Tr
imports Lift
begin
```

16.1 Type definition and constructors

```
types
  tr = bool lift

translations
  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*: $[\![p = \bot \Longrightarrow Q;\ p = TT \Longrightarrow Q;\ p = FF \Longrightarrow Q]\!] \Longrightarrow Q$
unfolding *FF-def TT-def* **by** (*induct p*) *auto*

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

distinctness for type *tr*

lemma *dist-less-tr* [*simp*]:
$\neg\ TT \sqsubseteq \bot\ \neg\ FF \sqsubseteq \bot\ \neg\ TT \sqsubseteq FF\ \neg\ FF \sqsubseteq TT$
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-less-iff* [*simp*]: $TT \sqsubseteq x \longleftrightarrow x = TT$
by (*induct x rule*: *tr-induct*) *simp-all*

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

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

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

16.2 Case analysis

defaultsort *pcpo*

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

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

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

by (*simp-all add*: *ifte-def TT-def FF-def*)

16.3 Boolean connectives

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

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

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

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

tactic for tr-thms with case split

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

lemmas about andalso, orelse, neg and if

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

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

apply (*cases y rule*: *trE*, *simp-all*)
done

lemma *neg-thms* [*simp*]:
  $neg{\cdot}TT = FF$
  $neg{\cdot}FF = TT$
  $neg{\cdot}\bot = \bot$
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 (*rule-tac p* = *Q* **in** *trE*)
apply (*simp-all*)
done

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

16.4 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*:
  $t \neq \bot \Longrightarrow ((t\ andalso\ s) = FF) = (t = FF \vee s = FF)$
apply (*rule-tac p* = *t* **in** *trE*)
apply *simp-all*
done

lemma *andalso-and*:
  $t \neq \bot \Longrightarrow ((t\ andalso\ s) \neq FF) = (t \neq FF \wedge s \neq FF)$
apply (*rule-tac p* = *t* **in** *trE*)
apply *simp-all*
done

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

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

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

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

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

16.5 Compactness

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

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

end

17 Ssum: The type of strict sums

theory *Ssum*
imports *Cprod Tr*
begin

defaultsort *pcpo*

17.1 Definition of strict sum type

pcpodef (*Ssum*) ('*a*, '*b*) ++ (**infixr** ++ *10*) =
{*p* :: *tr* × ('*a* × '*b*).
(*cfst*·*p* ⊑ *TT* ⟷ *csnd*·(*csnd*·*p*) = ⊥) ∧
(*cfst*·*p* ⊑ *FF* ⟷ *cfst*·(*csnd*·*p*) = ⊥)}
by *simp-all*

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

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

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

17.2 Definitions of constructors

definition
sinl :: '*a* → ('*a* ++ '*b*) **where**

sinl = (Λ *a*. *Abs-Ssum* <*strictify*·(Λ -. *TT*)·*a*, *a*, ⊥>)

definition
sinr :: *'b* → (*'a* ++ *'b*) **where**
sinr = (Λ *b*. *Abs-Ssum* <*strictify*·(Λ -. *FF*)·*b*, ⊥, *b*>)

lemma *sinl-Ssum*: <*strictify*·(Λ -. *TT*)·*a*, *a*, ⊥> ∈ *Ssum*
by (*simp add*: *Ssum-def strictify-conv-if*)

lemma *sinr-Ssum*: <*strictify*·(Λ -. *FF*)·*b*, ⊥, *b*> ∈ *Ssum*
by (*simp add*: *Ssum-def strictify-conv-if*)

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

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

lemma *Rep-Ssum-sinl*: *Rep-Ssum* (*sinl*·*a*) = <*strictify*·(Λ -. *TT*)·*a*, *a*, ⊥>
by (*simp add*: *sinl-Abs-Ssum Abs-Ssum-inverse sinl-Ssum*)

lemma *Rep-Ssum-sinr*: *Rep-Ssum* (*sinr*·*b*) = <*strictify*·(Λ -. *FF*)·*b*, ⊥, *b*>
by (*simp add*: *sinr-Abs-Ssum Abs-Ssum-inverse sinr-Ssum*)

17.3 Properties of *sinl* and *sinr*

Ordering

lemma *sinl-less* [*simp*]: (*sinl*·*x* ⊑ *sinl*·*y*) = (*x* ⊑ *y*)
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl strictify-conv-if*)

lemma *sinr-less* [*simp*]: (*sinr*·*x* ⊑ *sinr*·*y*) = (*x* ⊑ *y*)
by (*simp add*: *less-Ssum-def Rep-Ssum-sinr strictify-conv-if*)

lemma *sinl-less-sinr* [*simp*]: (*sinl*·*x* ⊑ *sinr*·*y*) = (*x* = ⊥)
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl Rep-Ssum-sinr strictify-conv-if*)

lemma *sinr-less-sinl* [*simp*]: (*sinr*·*x* ⊑ *sinl*·*y*) = (*x* = ⊥)
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl Rep-Ssum-sinr strictify-conv-if*)

Equality

lemma *sinl-eq* [*simp*]: (*sinl*·*x* = *sinl*·*y*) = (*x* = *y*)
by (*simp add*: *po-eq-conv*)

lemma *sinr-eq* [*simp*]: (*sinr*·*x* = *sinr*·*y*) = (*x* = *y*)
by (*simp add*: *po-eq-conv*)

lemma *sinl-eq-sinr* [*simp*]: (*sinl*·*x* = *sinr*·*y*) = (*x* = ⊥ ∧ *y* = ⊥)
by (*subst po-eq-conv*, *simp*)

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

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

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

Strictness

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

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

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

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

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

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

Compactness

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

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

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

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

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

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

17.4 Case analysis

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

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

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

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

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

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

17.5 Case analysis combinator

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

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

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

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

lemma *beta-sscase*:
$sscase{\cdot}f{\cdot}g{\cdot}s = (\Lambda{<}t,\ x,\ y{>}.\ If\ t\ then\ f{\cdot}x\ else\ g{\cdot}y\ fi){\cdot}(Rep\text{-}Ssum\ s)$
unfolding *sscase-def* **by** (*simp add*: *cont-Rep-Ssum cont2cont-LAM*)

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*)

17.6 Strict sum preserves flatness

instance ++ :: (*flat*, *flat*) *flat*
apply (*intro-classes*, *clarify*)
apply (*rule-tac p=x* **in** *ssumE*, *simp*)
apply (*rule-tac p=y* **in** *ssumE*, *simp-all add*: *flat-less-iff*)
apply (*rule-tac p=y* **in** *ssumE*, *simp-all add*: *flat-less-iff*)
done

17.7 Strict sum is a bifinite domain

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

definition
approx-ssum-def:
$approx = (\lambda n.\ sscase{\cdot}(\Lambda\ x.\ sinl{\cdot}(approx\ n{\cdot}x)){\cdot}(\Lambda\ y.\ sinr{\cdot}(approx\ n{\cdot}y)))$

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

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

instance proof
fix i :: *nat* **and** x :: $'a \oplus 'b$
show *chain* ($approx$ :: $nat \Rightarrow 'a \oplus 'b \to 'a \oplus 'b$)
unfolding *approx-ssum-def* **by** *simp*
show $(\bigsqcup i.\ approx\ i{\cdot}x) = x$
unfolding *approx-ssum-def*
by (*simp add*: *lub-distribs eta-cfun*)
show $approx\ i{\cdot}(approx\ i{\cdot}x) = approx\ i{\cdot}x$

by (*cases x*, *simp add*: *approx-ssum-def*, *simp*, *simp*)
have {*x*::'*a* ⊕ '*b*. *approx i*·*x* = *x*} ⊆
(λ*x*. *sinl*·*x*) ' {*x*. *approx i*·*x* = *x*} ∪
(λ*x*. *sinr*·*x*) ' {*x*. *approx i*·*x* = *x*}
by (*rule subsetI*, *case-tac x rule*: *ssumE2*, *simp*, *simp*)
thus *finite* {*x*::'*a* ⊕ '*b*. *approx i*·*x* = *x*}
by (*rule finite-subset*,
intro finite-UnI finite-imageI finite-fixes-approx)
qed

end

end

18 Sprod: The type of strict products

theory *Sprod*
imports *Cprod*
begin

defaultsort *pcpo*

18.1 Definition of strict product type

pcpodef (*Sprod*) ('*a*, '*b*) ** (**infixr** ** *20*) =
{*p*::'*a* × '*b*. *p* = ⊥ ∨ (*cfst*·*p* ≠ ⊥ ∧ *csnd*·*p* ≠ ⊥)}
by *simp-all*

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

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

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

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

18.2 Definitions of constants

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

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

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

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

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

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

18.3 Case analysis

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

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

lemma *Exh-Sprod*:
$z = \bot \vee (\exists\ a\ b.\ z = (:a,\ b:) \wedge a \neq \bot \wedge b \neq \bot)$
apply (*insert Rep-Sprod* [*of z*])
apply (*simp add*: *Rep-Sprod-simps eq-cprod*)
apply (*simp add*: *Sprod-def*)
apply (*erule disjE*, *simp*)
apply (*simp add*: *strictify-conv-if*)
apply *fast*
done

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

lemma *sprod-induct* [*induct type*: $**$]:

$[\![P\ \bot;\ \bigwedge x\ y.\ [\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow P\ (:x,\ y:)]\!] \Longrightarrow P\ x$
by (*cases* x, *simp-all*)

18.4 Properties of *spair*

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

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

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

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

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

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

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

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

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

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

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

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

18.5 Properties of *sfst* and *ssnd*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18.6 Compactness

lemma *compact-sfst*: *compact x* $\Longrightarrow$ *compact* (*sfst*·*x*)
by (*rule compactI*, *simp add*: *sfst-less-iff*)

lemma *compact-ssnd*: *compact x* $\Longrightarrow$ *compact* (*ssnd*·*x*)
by (*rule compactI*, *simp add*: *ssnd-less-iff*)

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

lemma *compact-spair-iff*:
compact (:*x*, *y*:) = ($x = \bot \vee y = \bot \vee$ (*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

18.7 Properties of *ssplit*

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

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

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

18.8 Strict product preserves flatness

instance $**$ :: (*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-less-iff flat-less-iff*)
done
qed

18.9 Strict product is a bifinite domain

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

definition
approx-sprod-def:

approx = (λ*n*. Λ(:*x*, *y*:). (:*approx n*·*x*, *approx n*·*y*:))

instance proof
fix *i* :: *nat* **and** *x* :: *'a* ⊗ *'b*
show *chain* (*approx* :: *nat* ⇒ *'a* ⊗ *'b* → *'a* ⊗ *'b*)
unfolding *approx-sprod-def* **by** *simp*
show (⨆ *i*. *approx i*·*x*) = *x*
unfolding *approx-sprod-def*
by (*simp add*: *lub-distribs eta-cfun*)
show *approx i*·(*approx i*·*x*) = *approx i*·*x*
unfolding *approx-sprod-def*
by (*simp add*: *ssplit-def strictify-conv-if*)
have *Rep-Sprod* ' {*x*::*'a* ⊗ *'b*. *approx i*·*x* = *x*} ⊆ {*x*. *approx i*·*x* = *x*}
unfolding *approx-sprod-def*
apply (*clarify*, *case-tac x*)
apply (*simp add*: *Rep-Sprod-strict*)
apply (*simp add*: *Rep-Sprod-spair spair-eq-iff*)
done
hence *finite* (*Rep-Sprod* ' {*x*::*'a* ⊗ *'b*. *approx i*·*x* = *x*})
using *finite-fixes-approx* **by** (*rule finite-subset*)
thus *finite* {*x*::*'a* ⊗ *'b*. *approx i*·*x* = *x*}
by (*rule finite-imageD*, *simp add*: *inj-on-def Rep-Sprod-inject*)
qed

end

lemma *approx-spair* [*simp*]:
approx i·(:*x*, *y*:) = (:*approx i*·*x*, *approx i*·*y*:)
unfolding *approx-sprod-def*
by (*simp add*: *ssplit-def strictify-conv-if*)

end

19 One: The unit domain

theory *One*
imports *Lift*
begin

types *one* = *unit lift*
translations
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*: $[\![p = \bot \Longrightarrow Q;\ p = ONE \Longrightarrow Q]\!] \Longrightarrow Q$
unfolding *ONE-def* **by** (*induct p*) *simp-all*

lemma *one-induct*: $[\![P\ \bot;\ P\ ONE]\!] \Longrightarrow P\ x$
by (*cases x rule*: *oneE*) *simp-all*

lemma *dist-less-one* [*simp*]: $\neg\ ONE \sqsubseteq \bot$
unfolding *ONE-def* **by** *simp*

lemma *less-ONE* [*simp*]: $x \sqsubseteq ONE$
by (*induct x rule*: *one-induct*) *simp-all*

lemma *ONE-less-iff* [*simp*]: $ONE \sqsubseteq x \longleftrightarrow x = ONE$
by (*induct x rule*: *one-induct*) *simp-all*

lemma *dist-eq-one* [*simp*]: $ONE \neq \bot\ \bot \neq ONE$
unfolding *ONE-def* **by** *simp-all*

lemma *one-neq-iffs* [*simp*]:
$x \neq ONE \longleftrightarrow x = \bot$
$ONE \neq x \longleftrightarrow x = \bot$
$x \neq \bot \longleftrightarrow x = ONE$
$\bot \neq x \longleftrightarrow x = ONE$
by (*induct x rule*: *one-induct*) *simp-all*

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

Case analysis function for type *one*

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

translations
case x of XCONST ONE ⇒ *t* == *CONST one-when*·*t*·*x*
Λ (*XCONST ONE*). *t* == *CONST one-when*·*t*

lemma *one-when1* [*simp*]: (*case* ⊥ *of ONE* ⇒ *t*) = ⊥
by (*simp add*: *one-when-def*)

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

lemma *one-when3* [*simp*]: (*case x of ONE* ⇒ *ONE*) = *x*
by (*induct x rule*: *one-induct*) *simp-all*

end

20 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun Cprod*
begin

defaultsort *pcpo*

20.1 Iteration

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

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

Derive inductive properties of iterate from primitive recursion

lemma *iterate-0* [*simp*]: $iterate\ 0{\cdot}F{\cdot}x = x$
by *simp*

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

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

lemma *iterate-Suc2*: $iterate\ (Suc\ n){\cdot}F{\cdot}x = iterate\ n{\cdot}F{\cdot}(F{\cdot}x)$
by (*induct 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. This property is essential since monotonicity of iterate makes no sense.

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

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

20.2 Least fixed point operator

definition

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

Binder syntax for *fix*

abbreviation
fix-syn :: $('a \Rightarrow 'a) \Rightarrow 'a$ (**binder** *FIX* *10*) **where**
fix-syn $(\lambda x.\ f\ x) \equiv fix{\cdot}(\Lambda\ x.\ f\ x)$

notation (*xsymbols*)
fix-syn (**binder** μ *10*)

Properties of *fix*

direct connection between *fix* and iteration

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

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

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

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

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

lemma *fix-eqI*:
  assumes *fixed*: $F{\cdot}x = x$ **and** *least*: $\bigwedge z.\ F{\cdot}z = z \Longrightarrow x \sqsubseteq z$
  shows $fix{\cdot}F = x$
apply (*rule antisym-less*)
apply (*rule fix-least* [*OF fixed*])
apply (*rule least* [*OF fix-eq* [*symmetric*]])
done

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

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

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

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

strictness of *fix*

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

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

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

fix applied to identity and constant functions

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

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

20.3 Fixed point induction

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

apply (*rule nat-induct*, *simp-all*)
done

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

lemma *fix-ind2*:
  assumes *adm*: *adm P*
  assumes *0*: $P\ \bot$ **and** *1*: $P\ (F{\cdot}\bot)$
  assumes *step*: $\bigwedge x.\ [\![P\ x;\ P\ (F{\cdot}x)]\!] \Longrightarrow P\ (F{\cdot}(F{\cdot}x))$
  shows $P\ (fix{\cdot}F)$
unfolding *fix-def2*
apply (*rule admD* [*OF adm chain-iterate*])
apply (*rule nat-less-induct*)
apply (*case-tac n*)
apply (*simp add*: *0*)
apply (*case-tac nat*)
apply (*simp add*: *1*)
apply (*frule-tac x=nat* **in** *spec*)
apply (*simp add*: *step*)
done

20.4 Recursive let bindings

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

nonterminals
  recbinds recbindt recbind

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

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

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

Bekic's Theorem: Simultaneous fixed points over pairs can be written in

terms of separate fixed points.

lemma *fix-cprod*:
  $fix{\cdot}(F{::}'a \times 'b \to 'a \times 'b) =$
   $\langle \mu\ x.\ cfst{\cdot}(F{\cdot}\langle x,\ \mu\ y.\ csnd{\cdot}(F{\cdot}\langle x,\ y\rangle)\rangle),$
    $\mu\ y.\ csnd{\cdot}(F{\cdot}\langle \mu\ x.\ cfst{\cdot}(F{\cdot}\langle x,\ \mu\ y.\ csnd{\cdot}(F{\cdot}\langle x,\ y\rangle)\rangle),\ y\rangle)\rangle$
  (**is** $fix{\cdot}F = \langle ?x,\ ?y\rangle$)
proof (*rule fix-eqI*)
  have *1*: $cfst{\cdot}(F{\cdot}\langle ?x,\ ?y\rangle) = ?x$
    by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
  have *2*: $csnd{\cdot}(F{\cdot}\langle ?x,\ ?y\rangle) = ?y$
    by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
  from *1 2* **show** $F{\cdot}\langle ?x,\ ?y\rangle = \langle ?x,\ ?y\rangle$ **by** (*simp add*: *eq-cprod*)
next
  fix z **assume** *F-z*: $F{\cdot}z = z$
  then obtain $x\ y$ **where** z: $z = \langle x,y\rangle$ **by** (*rule-tac p=z* **in** *cprodE*)
  from *F-z z* **have** *F-x*: $cfst{\cdot}(F{\cdot}\langle x,\ y\rangle) = x$ **by** *simp*
  from *F-z z* **have** *F-y*: $csnd{\cdot}(F{\cdot}\langle x,\ y\rangle) = y$ **by** *simp*
  let $?y1 = \mu\ y.\ csnd{\cdot}(F{\cdot}\langle x,\ y\rangle)$
  have $?y1 \sqsubseteq y$ **by** (*rule fix-least*, *simp add*: *F-y*)
  hence $cfst{\cdot}(F{\cdot}\langle x,\ ?y1\rangle) \sqsubseteq cfst{\cdot}(F{\cdot}\langle x,\ y\rangle)$ **by** (*simp add*: *monofun-cfun*)
  hence $cfst{\cdot}(F{\cdot}\langle x,\ ?y1\rangle) \sqsubseteq x$ **using** *F-x* **by** *simp*
  hence *1*: $?x \sqsubseteq x$ **by** (*simp add*: *fix-least-less*)
  hence $csnd{\cdot}(F{\cdot}\langle ?x,\ y\rangle) \sqsubseteq csnd{\cdot}(F{\cdot}\langle x,\ y\rangle)$ **by** (*simp add*: *monofun-cfun*)
  hence $csnd{\cdot}(F{\cdot}\langle ?x,\ y\rangle) \sqsubseteq y$ **using** *F-y* **by** *simp*
  hence *2*: $?y \sqsubseteq y$ **by** (*simp add*: *fix-least-less*)
  show $\langle ?x,\ ?y\rangle \sqsubseteq z$ **using** *z 1 2* **by** *simp*
qed

20.5 Weak admissibility

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

an admissible formula is also weak admissible

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

computational induction for weak admissible formulae

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

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

by (*simp*, *rule wfix-ind*)

end

21 Fixrec: Package for defining recursive functions in HOLCF

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

21.1 Maybe monad type

defaultsort *cpo*

pcpodef (**open**) *'a maybe* = *UNIV*::(*one* ++ *'a u*) *set*
by *simp-all*

definition
fail :: *'a maybe* **where**
fail = *Abs-maybe* (*sinl*·*ONE*)

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

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

lemma *maybeE*:
⟦*p* = ⊥ ⟹ *Q*; *p* = *fail* ⟹ *Q*; ⋀*x*. *p* = *return*·*x* ⟹ *Q*⟧ ⟹ *Q*
apply (*unfold fail-def return-def*)
apply (*cases p*, *rename-tac r*)
apply (*rule-tac p*=*r* **in** *ssumE*, *simp add*: *Abs-maybe-strict*)
apply (*rule-tac p*=*x* **in** *oneE*, *simp*, *simp*)
apply (*rule-tac p*=*y* **in** *upE*, *simp*, *simp add*: *cont-Abs-maybe*)
done

lemma *return-defined* [*simp*]: *return*·*x* ≠ ⊥
by (*simp add*: *return-def cont-Abs-maybe Abs-maybe-defined*)

lemma *fail-defined* [*simp*]: *fail* ≠ ⊥
by (*simp add*: *fail-def Abs-maybe-defined*)

lemma *return-eq* [*simp*]: (*return*·*x* = *return*·*y*) = (*x* = *y*)

by (*simp add*: *return-def cont-Abs-maybe Abs-maybe-inject*)

lemma *return-neq-fail* [*simp*]:
return·x ≠ fail fail ≠ return·x
by (*simp-all add*: *return-def fail-def cont-Abs-maybe Abs-maybe-inject*)

lemma *maybe-when-rews* [*simp*]:
maybe-when·f·r·⊥ = ⊥
maybe-when·f·r·fail = f
maybe-when·f·r·(return·x) = r·x
by (*simp-all add*: *return-def fail-def maybe-when-def cont-Rep-maybe*
cont2cont-LAM
cont-Abs-maybe Abs-maybe-inverse Rep-maybe-strict)

translations
case m of XCONST fail ⇒ t1 | XCONST return·x ⇒ t2
== CONST maybe-when·t1·(Λ x. t2)·m

21.1.1 Monadic bind operator

definition
bind :: *'a maybe → ('a → 'b maybe) → 'b maybe* **where**
bind = (Λ m f. case m of fail ⇒ fail | return·x ⇒ f·x)

monad laws

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

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

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

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

lemma *bind-assoc*:
bind·(bind·m·k)·h = bind·m·(Λ a. bind·(k·a)·h)
by (*rule-tac p=m* **in** *maybeE*, *simp-all*)

21.1.2 Run operator

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

rewrite rules for run

lemma *run-strict* [*simp*]: *run·⊥ = ⊥*

by (*simp add*: *run-def*)

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

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

21.1.3 Monad plus operator

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

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

rewrite rules for mplus

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

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

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

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

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

21.1.4 Fatbar combinator

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

abbreviation
fatbar-syn :: [*'a* → *'b maybe*, *'a* → *'b maybe*] ⇒ *'a* → *'b maybe* (**infixr** ‖ *60*)
where
m1 ‖ *m2* == *fatbar*·*m1*·*m2*

lemma *fatbar1*: *m*·*x* = ⊥ ⟹ (*m* ‖ *ms*)·*x* = ⊥
by (*simp add*: *fatbar-def*)

lemma *fatbar2*: *m*·*x* = *fail* ⟹ (*m* ‖ *ms*)·*x* = *ms*·*x*
by (*simp add*: *fatbar-def*)

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

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

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

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

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

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

21.2 Case branch combinator

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

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

lemma *branch-return* [*simp*]: $branch\ return{\cdot}r{\cdot}x = return{\cdot}(r{\cdot}x)$
by (*simp add*: *branch-def*)

21.2.1 Cases operator

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

rewrite rules for cases

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

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

lemma *cases-return* [*simp*]: $cases{\cdot}(return{\cdot}x) = x$
by (*simp add*: *cases-def*)

21.3 Case syntax

nonterminals
Case-syn Cases-syn

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

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

translations
-Case-syntax x ms == CONST Fixrec.cases·(ms·x)
-Case2 m ms == m ‖ ms

Parsing Case expressions

syntax
-pat :: 'a
-variable :: 'a
-noargs :: 'a

translations
-Case1 p r => CONST branch (-pat p)·(-variable p r)
-variable (-args x y) r => CONST csplit·(-variable x (-variable y r))
-variable -noargs r => CONST unit-when·r

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

Printing Case expressions

syntax
-match :: 'a

print-translation ⟪
let
fun dest-LAM (*Const* (@{*const-syntax Rep-CFun*},-) $ *Const* (@{*const-syntax unit-when*},-) $ *t*) =
(*Syntax.const -noargs*, *t*)
| *dest-LAM* (*Const* (@{*const-syntax Rep-CFun*},-) $ *Const* (@{*const-syntax csplit*},-) $ *t*) =
let
val (*v1*, *t1*) = *dest-LAM t*;
val (*v2*, *t2*) = *dest-LAM t1*;

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

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

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

translations

```
x <= -match Fixrec.return (-variable x)
```

21.4 Pattern combinators for data constructors

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

definition

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

definition

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

definition

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

definition

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

definition

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

definition

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

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

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

Parse translations (patterns)

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

CONST version is also needed for constructors with special syntax

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

Parse translations (variables)

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

translations
-variable (*CONST cpair*·*x*·*y*) *r* => *-variable* (*-args x y*) *r*
-variable (*CONST spair*·*x*·*y*) *r* => *-variable* (*-args x y*) *r*

Print translations

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

CONST TT <= -match (CONST TT-pat) -noargs
CONST FF <= -match (CONST FF-pat) -noargs
CONST ONE <= -match (CONST ONE-pat) -noargs

lemma *cpair-pat1*:
branch p·r·x = ⊥ ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = ⊥
apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p=p·x* **in** *maybeE*, *simp-all*)
done

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

lemma *cpair-pat3*:
branch p·r·x = return·s ⟹
branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = branch q·s·y
apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p=p·x* **in** *maybeE*, *simp-all*)
apply (*rule-tac p=q·y* **in** *maybeE*, *simp-all*)
done

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

lemma *spair-pat* [*simp*]:
branch (spair-pat p1 p2)·r·⊥ = ⊥
⟦x ≠ ⊥; y ≠ ⊥⟧
⟹ branch (spair-pat p1 p2)·r·(:x, y:) =
branch (cpair-pat p1 p2)·r·⟨x, y⟩
by (*simp-all add*: *branch-def spair-pat-def*)

lemma *sinl-pat* [*simp*]:
branch (sinl-pat p)·r·⊥ = ⊥
x ≠ ⊥ ⟹ branch (sinl-pat p)·r·(sinl·x) = branch p·r·x
y ≠ ⊥ ⟹ branch (sinl-pat p)·r·(sinr·y) = fail
by (*simp-all add*: *branch-def sinl-pat-def*)

lemma *sinr-pat* [*simp*]:
branch (sinr-pat p)·r·⊥ = ⊥
x ≠ ⊥ ⟹ branch (sinr-pat p)·r·(sinl·x) = fail
y ≠ ⊥ ⟹ branch (sinr-pat p)·r·(sinr·y) = branch p·r·y
by (*simp-all add*: *branch-def sinr-pat-def*)

lemma *up-pat* [*simp*]:
branch (up-pat p)·r·⊥ = ⊥
branch (up-pat p)·r·(up·x) = branch p·r·x

by (*simp-all add*: *branch-def up-pat-def*)

lemma *TT-pat* [*simp*]:
branch TT-pat·(*unit-when*·*r*)·⊥ = ⊥
branch TT-pat·(*unit-when*·*r*)·*TT* = *return*·*r*
branch TT-pat·(*unit-when*·*r*)·*FF* = *fail*
by (*simp-all add*: *branch-def TT-pat-def*)

lemma *FF-pat* [*simp*]:
branch FF-pat·(*unit-when*·*r*)·⊥ = ⊥
branch FF-pat·(*unit-when*·*r*)·*TT* = *fail*
branch FF-pat·(*unit-when*·*r*)·*FF* = *return*·*r*
by (*simp-all add*: *branch-def FF-pat-def*)

lemma *ONE-pat* [*simp*]:
branch ONE-pat·(*unit-when*·*r*)·⊥ = ⊥
branch ONE-pat·(*unit-when*·*r*)·*ONE* = *return*·*r*
by (*simp-all add*: *branch-def ONE-pat-def*)

21.5 Wildcards, as-patterns, and lazy patterns

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

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

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

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

Parse translations (patterns)

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

Parse translations (variables)

translations
-variable - *r* => *-variable -noargs r*
-variable (*-as-pat x y*) *r* => *-variable* (*-args x y*) *r*
-variable (*-lazy-pat x*) *r* => *-variable x r*

Print translations

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

Lazy patterns in lambda abstractions

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

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

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

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

21.6 Match functions for built-in types

defaultsort *pcpo*

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

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

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

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

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

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

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

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

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

lemma *match-UU-simps* [*simp*]:
match-UU·x = *fail*
by (*simp add*: *match-UU-def*)

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

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

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

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

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

lemma *match-ONE-simps* [*simp*]:
match-ONE·ONE = *return·*()

match-ONE·⊥ = ⊥
by (*simp-all add*: *match-ONE-def*)

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

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

21.7 Mutual recursion

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

lemma *cpair-equalI*: ⟦*x* ≡ *cfst*·*p*; *y* ≡ *csnd*·*p*⟧ ⟹ <*x*,*y*> ≡ *p*
by (*simp add*: *surjective-pairing-Cprod2*)

lemma *cpair-eqD1*: <*x*,*y*> = <*x*′,*y*′> ⟹ *x* = *x*′
by *simp*

lemma *cpair-eqD2*: <*x*,*y*> = <*x*′,*y*′> ⟹ *y* = *y*′
by *simp*

lemma for proving rewrite rules

lemma *ssubst-lhs*: ⟦*t* = *s*; *P s* = *Q*⟧ ⟹ *P t* = *Q*
by *simp*

21.8 Initializing the fixrec package

use *Tools/fixrec-package.ML*

setup ⟪ *FixrecPackage.setup* ⟫

setup ⟪
FixrecPackage.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 cpair*}, @{*const-name match-cpair*}),
(@{*const-name ONE*}, @{*const-name match-ONE*}),
(@{*const-name TT*}, @{*const-name match-TT*}),
(@{*const-name FF*}, @{*const-name match-FF*})]
⟫

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

end

22 Domain: Domain package

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

defaultsort *pcpo*

22.1 Continuous isomorphisms

A locale for continuous isomorphisms

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

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

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

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

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

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

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

```
  have ⊥ ⊑ rep·⊥ ..
  then have abs·⊥ ⊑ abs·(rep·⊥) by (rule monofun-cfun-arg)
  then have abs·⊥ ⊑ ⊥ by simp
  then show ?thesis by (rule UU-I)
qed

lemma rep-strict: rep·⊥ = ⊥
  by (rule iso.abs-strict [OF swap])

lemma abs-defin': abs·x = ⊥ ⟹ x = ⊥
proof −
  have x = rep·(abs·x) by simp
  also assume abs·x = ⊥
  also note rep-strict
  finally show x = ⊥ .
qed

lemma rep-defin': rep·z = ⊥ ⟹ z = ⊥
  by (rule iso.abs-defin' [OF swap])

lemma abs-defined: z ≠ ⊥ ⟹ abs·z ≠ ⊥
  by (erule contrapos-nn, erule abs-defin')

lemma rep-defined: z ≠ ⊥ ⟹ rep·z ≠ ⊥
  by (rule iso.abs-defined [OF iso.swap]) (rule iso-axioms)

lemma abs-defined-iff: (abs·x = ⊥) = (x = ⊥)
  by (auto elim: abs-defin' intro: abs-strict)

lemma rep-defined-iff: (rep·x = ⊥) = (x = ⊥)
  by (rule iso.abs-defined-iff [OF iso.swap]) (rule iso-axioms)

lemma (in iso) compact-abs-rev: compact (abs·x) ⟹ compact x
proof (unfold compact-def)
  assume adm (λy. ¬ abs·x ⊑ y)
  with cont-Rep-CFun2
  have adm (λy. ¬ abs·x ⊑ abs·y) by (rule adm-subst)
  then show adm (λy. ¬ x ⊑ y) using abs-less by simp
qed

lemma compact-rep-rev: compact (rep·x) ⟹ compact x
  by (rule iso.compact-abs-rev [OF iso.swap]) (rule iso-axioms)

lemma compact-abs: compact x ⟹ compact (abs·x)
  by (rule compact-rep-rev) simp

lemma compact-rep: compact x ⟹ compact (rep·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

22.2 Casedist

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

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

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

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

apply (*rule-tac p=x* **in** *upE*)
apply *simp*
apply *fast*
apply (*force intro*!: *spair-defined*)
done

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

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

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

Rules for turning exh into casedist

lemma *exh-casedist0*: $[\![R;\ R \Longrightarrow P]\!] \Longrightarrow P$
by *auto*

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

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

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

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

end

23 Completion: Defining bifinite domains by ideal completion

theory *Completion*
imports *Bifinite*
begin

23.1 Ideals over a preorder

locale *preorder* =
  fixes $r :: 'a::type \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *50*)
  assumes *r-refl*: $x \preceq x$
  assumes *r-trans*: $\llbracket x \preceq y; y \preceq z \rrbracket \Longrightarrow x \preceq z$
begin

definition
  ideal :: $'a\ set \Rightarrow bool$ **where**
  $ideal\ A = ((\exists x.\ x \in A) \wedge (\forall x{\in}A.\ \forall y{\in}A.\ \exists z{\in}A.\ x \preceq z \wedge y \preceq z) \wedge$
    $(\forall x\ y.\ x \preceq y \longrightarrow y \in A \longrightarrow x \in A))$

lemma *idealI*:
  assumes $\exists x.\ x \in A$
  assumes $\bigwedge x\ y.\ \llbracket x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
  assumes $\bigwedge x\ y.\ \llbracket x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
  shows *ideal A*
unfolding *ideal-def* **using** *prems* **by** *fast*

lemma *idealD1*:
  $ideal\ A \Longrightarrow \exists x.\ x \in A$
unfolding *ideal-def* **by** *fast*

lemma *idealD2*:
  $\llbracket ideal\ A; x \in A; y \in A \rrbracket \Longrightarrow \exists z{\in}A.\ x \preceq z \wedge y \preceq z$
unfolding *ideal-def* **by** *fast*

lemma *idealD3*:
  $\llbracket ideal\ A; x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$
unfolding *ideal-def* **by** *fast*

lemma *ideal-directed-finite*:
  assumes *A*: *ideal A*
  shows $\llbracket finite\ U;\ U \subseteq A \rrbracket \Longrightarrow \exists z{\in}A.\ \forall x{\in}U.\ x \preceq z$
apply (*induct U set*: *finite*)
apply (*simp add*: *idealD1* [*OF A*])
apply (*simp*, *clarify*, *rename-tac y*)
apply (*drule* (*1*) *idealD2* [*OF A*])
apply (*clarify*, *erule-tac x*=*z* **in** *rev-bexI*)
apply (*fast intro*: *r-trans*)
done

lemma *ideal-principal*: *ideal* $\{x.\ x \preceq z\}$
apply (*rule idealI*)
apply (*rule-tac x=z* **in** *exI*)
apply (*fast intro*: *r-refl*)
apply (*rule-tac x=z* **in** *bexI*, *fast*)
apply (*fast intro*: *r-refl*)
apply (*fast intro*: *r-trans*)
done

lemma *ex-ideal*: $\exists A.$ *ideal A*
by (*rule exI*, *rule ideal-principal*)

lemma *directed-image-ideal*:
assumes *A*: *ideal A*
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows *directed* (*f* ‘ *A*)
apply (*rule directedI*)
apply (*cut-tac idealD1* [*OF A*], *fast*)
apply (*clarify*, *rename-tac a b*)
apply (*drule* (*1*) *idealD2* [*OF A*])
apply (*clarify*, *rename-tac c*)
apply (*rule-tac x=f c* **in** *rev-bexI*)
apply (*erule imageI*)
apply (*simp add*: *f*)
done

lemma *lub-image-principal*:
assumes *f*: $\bigwedge x\ y.\ x \preceq y \Longrightarrow f\ x \sqsubseteq f\ y$
shows $(\bigsqcup x{\in}\{x.\ x \preceq y\}.\ f\ x) = f\ y$
apply (*rule thelubI*)
apply (*rule is-lub-maximal*)
apply (*rule ub-imageI*)
apply (*simp add*: *f*)
apply (*rule imageI*)
apply (*simp add*: *r-refl*)
done

The set of ideals is a cpo

lemma *ideal-UN*:
fixes *A* :: *nat* ⇒ *'a set*
assumes *ideal-A*: $\bigwedge i.$ *ideal* (*A i*)
assumes *chain-A*: $\bigwedge i\ j.\ i \leq j \Longrightarrow A\ i \subseteq A\ j$
shows *ideal* $(\bigcup i.\ A\ i)$
apply (*rule idealI*)
apply (*cut-tac idealD1* [*OF ideal-A*], *fast*)
apply (*clarify*, *rename-tac i j*)
apply (*drule subsetD* [*OF chain-A* [*OF le-maxI1*]])
apply (*drule subsetD* [*OF chain-A* [*OF le-maxI2*]])

```
  apply (drule (1) idealD2 [OF ideal-A])
  apply blast
 apply clarify
 apply (drule (1) idealD3 [OF ideal-A])
 apply fast
done

lemma typedef-ideal-po:
  fixes Abs :: 'a set ⇒ 'b::sq-ord
  assumes type: type-definition Rep Abs {S. ideal S}
  assumes less: ⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y
  shows OFCLASS('b, po-class)
 apply (intro-classes, unfold less)
  apply (rule subset-refl)
 apply (erule (1) subset-trans)
 apply (rule type-definition.Rep-inject [OF type, THEN iffD1])
 apply (erule (1) subset-antisym)
done

lemma
  fixes Abs :: 'a set ⇒ 'b::po
  assumes type: type-definition Rep Abs {S. ideal S}
  assumes less: ⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y
  assumes S: chain S
  shows typedef-ideal-lub: range S <<| Abs (⋃i. Rep (S i))
    and typedef-ideal-rep-contlub: Rep (⨆i. S i) = (⋃i. Rep (S i))
proof −
  have 1: ideal (⋃i. Rep (S i))
    apply (rule ideal-UN)
     apply (rule type-definition.Rep [OF type, unfolded mem-Collect-eq])
    apply (subst less [symmetric])
    apply (erule chain-mono [OF S])
    done
  hence 2: Rep (Abs (⋃i. Rep (S i))) = (⋃i. Rep (S i))
    by (simp add: type-definition.Abs-inverse [OF type])
  show 3: range S <<| Abs (⋃i. Rep (S i))
    apply (rule is-lubI)
     apply (rule is-ubI)
     apply (simp add: less 2, fast)
    apply (simp add: less 2 is-ub-def, fast)
    done
  hence 4: (⨆i. S i) = Abs (⋃i. Rep (S i))
    by (rule thelubI)
  show 5: Rep (⨆i. S i) = (⋃i. Rep (S i))
    by (simp add: 4 2)
qed

lemma typedef-ideal-cpo:
  fixes Abs :: 'a set ⇒ 'b::po
```

assumes *type*: *type-definition Rep Abs* {*S*. *ideal S*}
assumes *less*: $\bigwedge x\ y.\ x \sqsubseteq y \longleftrightarrow Rep\ x \subseteq Rep\ y$
shows *OFCLASS*('*b*, *cpo-class*)
by (*default*, *rule exI*, *erule typedef-ideal-lub* [*OF type less*])

end

interpretation *sq-le*: *preorder sq-le* :: '*a*::*po* ⇒ '*a* ⇒ *bool*
apply *unfold-locales*
apply (*rule refl-less*)
apply (*erule* (*1*) *trans-less*)
done

23.2 Lemmas about least upper bounds

lemma *finite-directed-contains-lub*:
$[\![finite\ S;\ directed\ S]\!] \Longrightarrow \exists\, u{\in}S.\ S <<|\ u$
apply (*drule* (*1*) *directed-finiteD*, *rule subset-refl*)
apply (*erule bexE*)
apply (*rule rev-bexI*, *assumption*)
apply (*erule* (*1*) *is-lub-maximal*)
done

lemma *lub-finite-directed-in-self*:
$[\![finite\ S;\ directed\ S]\!] \Longrightarrow lub\ S \in S$
apply (*drule* (*1*) *finite-directed-contains-lub*, *clarify*)
apply (*drule thelubI*, *simp*)
done

lemma *finite-directed-has-lub*: $[\![finite\ S;\ directed\ S]\!] \Longrightarrow \exists\, u.\ S <<|\ u$
by (*drule* (*1*) *finite-directed-contains-lub*, *fast*)

lemma *is-ub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ x \in S]\!] \Longrightarrow x \sqsubseteq lub\ S$
apply (*erule exE*, *drule lubI*)
apply (*drule is-lubD1*)
apply (*erule* (*1*) *is-ubD*)
done

lemma *is-lub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ S <|\ x]\!] \Longrightarrow lub\ S \sqsubseteq x$
by (*erule exE*, *drule lubI*, *erule is-lub-lub*)

23.3 Locale for ideal completion

locale *basis-take* = *preorder* +
fixes *take* :: *nat* ⇒ '*a*::*type* ⇒ '*a*
assumes *take-less*: *take n a* ≼ *a*
assumes *take-take*: *take n* (*take n a*) = *take n a*
assumes *take-mono*: *a* ≼ *b* ⟹ *take n a* ≼ *take n b*
assumes *take-chain*: *take n a* ≼ *take* (*Suc n*) *a*
assumes *finite-range-take*: *finite* (*range* (*take n*))

assumes *take-covers*: $\exists\, n.\ take\ n\ a = a$
begin

lemma *take-chain-less*: $m < n \Longrightarrow take\ m\ a \preceq take\ n\ a$
by (*erule less-Suc-induct*, *rule take-chain*, *erule* (*1*) *r-trans*)

lemma *take-chain-le*: $m \leq n \Longrightarrow take\ m\ a \preceq take\ n\ a$
by (*cases* $m = n$, *simp add*: *r-refl*, *simp add*: *take-chain-less*)

end

locale *ideal-completion* = *basis-take* +
fixes *principal* :: $'a$::*type* $\Rightarrow$ $'b$::*cpo*
fixes *rep* :: $'b$::*cpo* $\Rightarrow$ $'a$::*type set*
assumes *ideal-rep*: $\bigwedge x.\ preorder.ideal\ r\ (rep\ x)$
assumes *rep-contlub*: $\bigwedge Y.\ chain\ Y \Longrightarrow rep\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ rep\ (Y\ i))$
assumes *rep-principal*: $\bigwedge a.\ rep\ (principal\ a) = \{b.\ b \preceq a\}$
assumes *subset-repD*: $\bigwedge x\ y.\ rep\ x \subseteq rep\ y \Longrightarrow x \sqsubseteq y$
begin

lemma *finite-take-rep*: *finite* (*take n ' rep x*)
by (*rule finite-subset* [*OF image-mono* [*OF subset-UNIV*] *finite-range-take*])

lemma *rep-mono*: $x \sqsubseteq y \Longrightarrow rep\ x \subseteq rep\ y$
apply (*frule bin-chain*)
apply (*drule rep-contlub*)
apply (*simp only*: *thelubI* [*OF lub-bin-chain*])
apply (*rule subsetI*, *rule UN-I* [**where** *a=0*], *simp-all*)
done

lemma *less-def*: $x \sqsubseteq y \longleftrightarrow rep\ x \subseteq rep\ y$
by (*rule iffI* [*OF rep-mono subset-repD*])

lemma *rep-eq*: $rep\ x = \{a.\ principal\ a \sqsubseteq x\}$
unfolding *less-def rep-principal*
apply *safe*
apply (*erule* (*1*) *idealD3* [*OF ideal-rep*])
apply (*erule subsetD*, *simp add*: *r-refl*)
done

lemma *mem-rep-iff-principal-less*: $a \in rep\ x \longleftrightarrow principal\ a \sqsubseteq x$
by (*simp add*: *rep-eq*)

lemma *principal-less-iff-mem-rep*: $principal\ a \sqsubseteq x \longleftrightarrow a \in rep\ x$
by (*simp add*: *rep-eq*)

lemma *principal-less-iff* [*simp*]: $principal\ a \sqsubseteq principal\ b \longleftrightarrow a \preceq b$
by (*simp add*: *principal-less-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-less-iff* **..**

lemma *repD*: *a* ∈ *rep x* ⟹ *principal a* ⊑ *x*
by (*simp add*: *rep-eq*)

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

lemma *lessI*: (⋀*a*. *principal a* ⊑ *x* ⟹ *principal a* ⊑ *u*) ⟹ *x* ⊑ *u*
unfolding *principal-less-iff-mem-rep*
by (*simp add*: *less-def subset-eq*)

lemma *lub-principal-rep*: *principal* ‘ *rep x* <<| *x*
apply (*rule is-lubI*)
apply (*rule ub-imageI*)
apply (*erule repD*)
apply (*subst less-def*)
apply (*rule subsetI*)
apply (*drule* (*1*) *ub-imageD*)
apply (*simp add*: *rep-eq*)
done

23.4 Defining functions in terms of basis elements

definition
basis-fun :: (*'a*::*type* ⇒ *'c*::*cpo*) ⇒ *'b* → *'c* **where**
basis-fun = (λ*f*. (Λ *x*. *lub* (*f* ‘ *rep x*)))

lemma *basis-fun-lemma0*:
fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows ∃ *u*. *f* ‘ *take i* ‘ *rep x* <<| *u*
apply (*rule finite-directed-has-lub*)
apply (*rule finite-imageI*)
apply (*rule finite-take-rep*)
apply (*subst image-image*)
apply (*rule directed-image-ideal*)
apply (*rule ideal-rep*)
apply (*rule f-mono*)
apply (*erule take-mono*)
done

lemma *basis-fun-lemma1*:
fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
assumes *f-mono*: ⋀*a b*. *a* ≼ *b* ⟹ *f a* ⊑ *f b*
shows *chain* (λ*i*. *lub* (*f* ‘ *take i* ‘ *rep x*))
apply (*rule chainI*)
apply (*rule is-lub-thelub0*)

apply (*rule basis-fun-lemma0*, *erule f-mono*)
apply (*rule is-ubI*, *clarsimp*, *rename-tac a*)
apply (*rule trans-less* [*OF f-mono* [*OF take-chain*]])
apply (*rule is-ub-thelub0*)
apply (*rule basis-fun-lemma0*, *erule f-mono*)
apply *simp*
done

lemma *basis-fun-lemma2*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $f\ `\ rep\ x <<|\ (\bigsqcup i.\ lub\ (f\ `\ take\ i\ `\ rep\ x))$
apply (*rule is-lubI*)
apply (*rule ub-imageI*, *rename-tac a*)
apply (*cut-tac a=a* **in** *take-covers*, *erule exE*, *rename-tac i*)
apply (*erule subst*)
apply (*rule rev-trans-less*)
apply (*rule-tac x=i* **in** *is-ub-thelub*)
apply (*rule basis-fun-lemma1*, *erule f-mono*)
apply (*rule is-ub-thelub0*)
apply (*rule basis-fun-lemma0*, *erule f-mono*)
apply *simp*
apply (*rule is-lub-thelub* [*OF - ub-rangeI*])
apply (*rule basis-fun-lemma1*, *erule f-mono*)
apply (*rule is-lub-thelub0*)
apply (*rule basis-fun-lemma0*, *erule f-mono*)
apply (*rule is-ubI*, *clarsimp*, *rename-tac a*)
apply (*rule trans-less* [*OF f-mono* [*OF take-less*]])
apply (*erule* (*1*) *ub-imageD*)
done

lemma *basis-fun-lemma*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows $\exists u.\ f\ `\ rep\ x <<|\ u$
by (*rule exI*, *rule basis-fun-lemma2*, *erule f-mono*)

lemma *basis-fun-beta*:
fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes *f-mono*: $\bigwedge a\ b.\ a \preceq b \Longrightarrow f\ a \sqsubseteq f\ b$
shows *basis-fun* $f{\cdot}x = lub\ (f\ `\ rep\ x)$
unfolding *basis-fun-def*
proof (*rule beta-cfun*)
have *lub*: $\bigwedge x.\ \exists u.\ f\ `\ rep\ x <<|\ u$
using *f-mono* **by** (*rule basis-fun-lemma*)
show *cont*: *cont* $(\lambda x.\ lub\ (f\ `\ rep\ x))$
apply (*rule contI2*)
apply (*rule monofunI*)
apply (*rule is-lub-thelub0* [*OF lub ub-imageI*])

```
    apply (rule is-ub-thelub0 [OF lub imageI])
    apply (erule (1) subsetD [OF rep-mono])
   apply (rule is-lub-thelub0 [OF lub ub-imageI])
   apply (simp add: rep-contlub, clarify)
   apply (erule rev-trans-less [OF is-ub-thelub])
   apply (erule is-ub-thelub0 [OF lub imageI])
   done
qed

lemma basis-fun-principal:
  fixes f :: 'a::type ⇒ 'c::cpo
  assumes f-mono: ⋀a b. a ≺ b ⟹ f a ⊑ f b
  shows basis-fun f·(principal a) = f a
apply (subst basis-fun-beta, erule f-mono)
apply (subst rep-principal)
apply (rule lub-image-principal, erule f-mono)
done

lemma basis-fun-mono:
  assumes f-mono: ⋀a b. a ≺ b ⟹ f a ⊑ f b
  assumes g-mono: ⋀a b. a ≺ b ⟹ g a ⊑ g b
  assumes less: ⋀a. f a ⊑ g a
  shows basis-fun f ⊑ basis-fun g
 apply (rule less-cfun-ext)
 apply (simp only: basis-fun-beta f-mono g-mono)
 apply (rule is-lub-thelub0)
  apply (rule basis-fun-lemma, erule f-mono)
 apply (rule ub-imageI, rename-tac a)
 apply (rule trans-less [OF less])
 apply (rule is-ub-thelub0)
  apply (rule basis-fun-lemma, erule g-mono)
 apply (erule imageI)
done

lemma compact-principal [simp]: compact (principal a)
by (rule compactI2, simp add: principal-less-iff-mem-rep rep-contlub)
```

23.5 Bifiniteness of ideal completions

```
definition
  completion-approx :: nat ⇒ 'b → 'b where
  completion-approx = (λi. basis-fun (λa. principal (take i a)))

lemma completion-approx-beta:
  completion-approx i·x = (⨆ a∈rep x. principal (take i a))
unfolding completion-approx-def
by (simp add: basis-fun-beta principal-mono take-mono)

lemma completion-approx-principal:
```

completion-approx i·(principal a) = principal (take i a)
unfolding *completion-approx-def*
by (*simp add*: *basis-fun-principal principal-mono take-mono*)

lemma *chain-completion-approx*: *chain completion-approx*
unfolding *completion-approx-def*
apply (*rule chainI*)
apply (*rule basis-fun-mono*)
apply (*erule principal-mono* [*OF take-mono*])
apply (*erule principal-mono* [*OF take-mono*])
apply (*rule principal-mono* [*OF take-chain*])
done

lemma *lub-completion-approx*: $(\bigsqcup i.\ \textit{completion-approx}\ i{\cdot}x) = x$
unfolding *completion-approx-beta*
apply (*subst image-image* [**where** *f*=*principal*, *symmetric*])
apply (*rule unique-lub* [*OF - lub-principal-rep*])
apply (*rule basis-fun-lemma2*, *erule principal-mono*)
done

lemma *completion-approx-eq-principal*:
$\exists\, a{\in}\textit{rep}\ x.\ \textit{completion-approx}\ i{\cdot}x = \textit{principal}\ (\textit{take}\ i\ a)$
unfolding *completion-approx-beta*
apply (*subst image-image* [**where** *f*=*principal*, *symmetric*])
apply (*subgoal-tac finite* (*principal ‘ take i ‘ rep x*))
apply (*subgoal-tac directed* (*principal ‘ take i ‘ rep x*))
apply (*drule* (*1*) *lub-finite-directed-in-self*, *fast*)
apply (*subst image-image*)
apply (*rule directed-image-ideal*)
apply (*rule ideal-rep*)
apply (*erule principal-mono* [*OF take-mono*])
apply (*rule finite-imageI*)
apply (*rule finite-take-rep*)
done

lemma *completion-approx-idem*:
completion-approx i·(completion-approx i·x) = completion-approx i·x
using *completion-approx-eq-principal* [**where** *i*=*i* **and** *x*=*x*]
by (*auto simp add*: *completion-approx-principal take-take*)

lemma *finite-fixes-completion-approx*:
finite $\{x.\ \textit{completion-approx}\ i{\cdot}x = x\}$ (**is** *finite ?S*)
apply (*subgoal-tac ?S* $\subseteq$ *principal ‘ range* (*take i*))
apply (*erule finite-subset*)
apply (*rule finite-imageI*)
apply (*rule finite-range-take*)
apply (*clarify*, *erule subst*)
apply (*cut-tac x*=*x* **and** *i*=*i* **in** *completion-approx-eq-principal*)
apply *fast*

done

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes *P*: $\bigwedge a.\ P\ (\mathit{principal}\ a)$
shows *P x*
apply (*subgoal-tac P* ($\bigsqcup i.$ *completion-approx i·x*))
apply (*simp add*: *lub-completion-approx*)
apply (*rule admD* [*OF adm*])
apply (*simp add*: *chain-completion-approx*)
apply (*cut-tac x=x* **and** *i=i* **in** *completion-approx-eq-principal*)
apply (*clarify*, *simp add*: *P*)
done

lemma *principal-induct2*:
$[\![\bigwedge y.\ \mathit{adm}\ (\lambda x.\ P\ x\ y);\ \bigwedge x.\ \mathit{adm}\ (\lambda y.\ P\ x\ y);$
$\bigwedge a\ b.\ P\ (\mathit{principal}\ a)\ (\mathit{principal}\ b)]\!] \Longrightarrow P\ x\ y$
apply (*rule-tac x=y* **in** *spec*)
apply (*rule-tac x=x* **in** *principal-induct*, *simp*)
apply (*rule allI*, *rename-tac y*)
apply (*rule-tac x=y* **in** *principal-induct*, *simp*)
apply *simp*
done

lemma *compact-imp-principal*: *compact x* $\Longrightarrow \exists\, a.\ x = \mathit{principal}\ a$
apply (*drule adm-compact-neq* [*OF - cont-id*])
apply (*drule admD2* [**where** *Y*=$\lambda n.$ *completion-approx n·x*])
apply (*simp add*: *chain-completion-approx*)
apply (*simp add*: *lub-completion-approx*)
apply (*erule exE*, *erule ssubst*)
apply (*cut-tac i=i* **and** *x=x* **in** *completion-approx-eq-principal*)
apply (*clarify*, *erule exI*)
done

end

end

24 CompactBasis: Compact bases of domains

theory *CompactBasis*
imports *Completion*
begin

24.1 Compact bases of bifinite domains

defaultsort *profinite*

typedef (**open**) *'a compact-basis* = {*x*::*'a*::*profinite*. *compact x*}
by (*fast intro*: *compact-approx*)

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

instantiation *compact-basis* :: (*profinite*) *sq-ord*
begin

definition
compact-le-def:
(*op* ⊑) ≡ (λ*x y*. *Rep-compact-basis x* ⊑ *Rep-compact-basis y*)

instance ..

end

instance *compact-basis* :: (*profinite*) *po*
by (*rule typedef-po*
[*OF type-definition-compact-basis compact-le-def*])

Take function for compact basis

definition
compact-take :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis* **where**
compact-take = (λ*n a*. *Abs-compact-basis* (*approx n*·(*Rep-compact-basis a*)))

lemma *Rep-compact-take*:
Rep-compact-basis (*compact-take n a*) = *approx n*·(*Rep-compact-basis a*)
unfolding *compact-take-def*
by (*simp add*: *Abs-compact-basis-inverse*)

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

interpretation *compact-basis*:
basis-take sq-le compact-take
proof
fix *n* :: *nat* **and** *a* :: *'a compact-basis*
show *compact-take n a* ⊑ *a*
unfolding *compact-le-def*
by (*simp add*: *Rep-compact-take approx-less*)
next
fix *n* :: *nat* **and** *a* :: *'a compact-basis*
show *compact-take n* (*compact-take n a*) = *compact-take n a*
by (*simp add*: *Rep-compact-basis-inject* [*symmetric*] *Rep-compact-take*)
next
fix *n* :: *nat* **and** *a b* :: *'a compact-basis*
assume *a* ⊑ *b* **thus** *compact-take n a* ⊑ *compact-take n b*
unfolding *compact-le-def Rep-compact-take*
by (*rule monofun-cfun-arg*)

next
fix *n* :: *nat* **and** *a* :: *'a compact-basis*
show $\bigwedge n\ a.$ *compact-take n a* $\sqsubseteq$ *compact-take* (*Suc n*) *a*
unfolding *compact-le-def Rep-compact-take*
by (*rule chainE*, *simp*)
next
fix *n* :: *nat*
show *finite* (*range* (*compact-take n*))
apply (*rule finite-imageD* [**where** *f*=*Rep-compact-basis*])
apply (*rule finite-subset* [**where** *B*=*range* ($\lambda x.$ *approx n*·*x*)])
apply (*clarsimp simp add*: *Rep-compact-take*)
apply (*rule finite-range-approx*)
apply (*rule inj-onI*, *simp add*: *Rep-compact-basis-inject*)
done
next
fix *a* :: *'a compact-basis*
show $\exists\, n.$ *compact-take n a* = *a*
apply (*simp add*: *Rep-compact-basis-inject* [*symmetric*])
apply (*simp add*: *Rep-compact-take*)
apply (*rule profinite-compact-eq-approx*)
apply (*rule compact-Rep-compact-basis*)
done
qed

Ideal completion

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

interpretation *compact-basis*:
ideal-completion sq-le compact-take Rep-compact-basis approximants
proof
fix *w* :: *'a*
show *preorder.ideal sq-le* (*approximants w*)
proof (*rule sq-le.idealI*)
show $\exists\, x.$ *x* ∈ *approximants w*
unfolding *approximants-def*
apply (*rule-tac x*=*Abs-compact-basis* (*approx 0*·*w*) **in** *exI*)
apply (*simp add*: *Abs-compact-basis-inverse approx-less*)
done
next
fix *x y* :: *'a compact-basis*
assume *x* ∈ *approximants w y* ∈ *approximants w*
thus $\exists\, z \in$ *approximants w*. *x* $\sqsubseteq$ *z* ∧ *y* $\sqsubseteq$ *z*
unfolding *approximants-def*
apply *simp*
apply (*cut-tac a*=*x* **in** *compact-Rep-compact-basis*)
apply (*cut-tac a*=*y* **in** *compact-Rep-compact-basis*)
apply (*drule profinite-compact-eq-approx*)

```
    apply (drule profinite-compact-eq-approx)
    apply (clarify, rename-tac i j)
    apply (rule-tac x=Abs-compact-basis (approx (max i j)·w) in exI)
    apply (simp add: compact-le-def)
    apply (simp add: Abs-compact-basis-inverse approx-less)
    apply (erule subst, erule subst)
    apply (simp add: monofun-cfun chain-mono [OF chain-approx])
    done
 next
  fix x y :: 'a compact-basis
  assume x ⊑ y y ∈ approximants w thus x ∈ approximants w
    unfolding approximants-def
    apply simp
    apply (simp add: compact-le-def)
    apply (erule (1) trans-less)
    done
 qed
next
 fix Y :: nat ⇒ 'a
 assume Y: chain Y
 show approximants (⨆ i. Y i) = (⋃ i. approximants (Y i))
  unfolding approximants-def
  apply safe
  apply (simp add: compactD2 [OF compact-Rep-compact-basis Y])
  apply (erule trans-less, rule is-ub-thelub [OF Y])
  done
next
 fix a :: 'a compact-basis
 show approximants (Rep-compact-basis a) = {b. b ⊑ a}
  unfolding approximants-def compact-le-def ..
next
 fix x y :: 'a
 assume approximants x ⊆ approximants y thus x ⊑ y
  apply (subgoal-tac (⨆ i. approx i·x) ⊑ y, simp)
  apply (rule admD, simp, simp)
  apply (drule-tac c=Abs-compact-basis (approx i·x) in subsetD)
  apply (simp add: approximants-def Abs-compact-basis-inverse approx-less)
  apply (simp add: approximants-def Abs-compact-basis-inverse)
  done
qed
```

minimal compact element

```
definition
  compact-bot :: 'a::bifinite compact-basis where
  compact-bot = Abs-compact-basis ⊥

lemma Rep-compact-bot: Rep-compact-basis compact-bot = ⊥
unfolding compact-bot-def by (simp add: Abs-compact-basis-inverse)
```

lemma *compact-bot-minimal* [*simp*]: *compact-bot* $\sqsubseteq$ *a*
unfolding *compact-le-def Rep-compact-bot* **by** *simp*

24.2 A compact basis for powerdomains

typedef *'a pd-basis* =
{*S*::*'a*::*profinite compact-basis set*. *finite S* $\wedge$ *S* $\neq$ {}}
by (*rule-tac x*={*arbitrary*} **in** *exI*, *simp*)

lemma *finite-Rep-pd-basis* [*simp*]: *finite* (*Rep-pd-basis u*)
by (*insert Rep-pd-basis* [*of u*, *unfolded pd-basis-def*]) *simp*

lemma *Rep-pd-basis-nonempty* [*simp*]: *Rep-pd-basis u* $\neq$ {}
by (*insert Rep-pd-basis* [*of u*, *unfolded pd-basis-def*]) *simp*

unit and plus

definition
PDUnit :: *'a compact-basis* $\Rightarrow$ *'a pd-basis* **where**
PDUnit = (λx. *Abs-pd-basis* {*x*})

definition
PDPlus :: *'a pd-basis* $\Rightarrow$ *'a pd-basis* $\Rightarrow$ *'a pd-basis* **where**
PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* $\cup$ *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* $\cup$ *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

fold-pd

definition
fold-pd ::
$('a\ compact\text{-}basis \Rightarrow 'b{::}type) \Rightarrow ('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ pd\text{-}basis \Rightarrow 'b$
where *fold-pd g f t* = *fold1 f* (*g* ‘ *Rep-pd-basis t*)

lemma *fold-pd-PDUnit*:
assumes *ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDUnit x*) = *g x*
unfolding *fold-pd-def Rep-PDUnit* **by** *simp*

lemma *fold-pd-PDPlus*:
assumes *ab-semigroup-idem-mult f*
shows *fold-pd g f* (*PDPlus t u*) = *f* (*fold-pd g f t*) (*fold-pd g f u*)
proof −
interpret *ab-semigroup-idem-mult f* **by** *fact*
show *?thesis* **unfolding** *fold-pd-def Rep-PDPlus* **by** (*simp add*: *image-Un fold1-Un2*)
qed

Take function for powerdomain basis

definition
pd-take :: *nat* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
pd-take n = (λ*t*. *Abs-pd-basis* (*compact-take n* ‘ *Rep-pd-basis t*))

lemma *Rep-pd-take*:
Rep-pd-basis (*pd-take n t*) = *compact-take n* ‘ *Rep-pd-basis t*
unfolding *pd-take-def*
apply (*rule Abs-pd-basis-inverse*)
apply (*simp add*: *pd-basis-def*)
done

lemma *pd-take-simps* [*simp*]:
pd-take n (*PDUnit a*) = *PDUnit* (*compact-take n a*)
pd-take n (*PDPlus t u*) = *PDPlus* (*pd-take n t*) (*pd-take n u*)

apply (*simp-all add*: *Rep-pd-basis-inject* [*symmetric*])
apply (*simp-all add*: *Rep-pd-take Rep-PDUnit Rep-PDPlus image-Un*)
done

lemma *pd-take-idem*: *pd-take n* (*pd-take n t*) = *pd-take n t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-take*)
apply *simp*
done

lemma *finite-range-pd-take*: *finite* (*range* (*pd-take n*))
apply (*rule finite-imageD* [**where** *f*=*Rep-pd-basis*])
apply (*rule finite-subset* [**where** *B*=*Pow* (*range* (*compact-take n*))])
apply (*clarsimp simp add*: *Rep-pd-take*)
apply (*simp add*: *compact-basis.finite-range-take*)
apply (*rule inj-onI*, *simp add*: *Rep-pd-basis-inject*)
done

lemma *pd-take-covers*: ∃ *n*. *pd-take n t* = *t*
apply (*subgoal-tac* ∃ *n*. ∀ *m*≥*n*. *pd-take m t* = *t*, *fast*)
apply (*induct t rule*: *pd-basis-induct*)
apply (*cut-tac a*=*a* **in** *compact-basis.take-covers*)
apply (*clarify*, *rule-tac x*=*n* **in** *exI*)
apply (*clarify*, *simp*)
apply (*rule antisym-less*)
apply (*rule compact-basis.take-less*)
apply (*drule-tac a*=*a* **in** *compact-basis.take-chain-le*)
apply *simp*
apply (*clarify*, *rename-tac i j*)
apply (*rule-tac x*=*max i j* **in** *exI*, *simp*)
done

end

25 UpperPD: Upper powerdomain

theory *UpperPD*
imports *CompactBasis*
begin

25.1 Basis preorder

definition
upper-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♯ *50*) **where**
upper-le = (λ*u v*. ∀ *y*∈*Rep-pd-basis v*. ∃ *x*∈*Rep-pd-basis u*. *x* ⊑ *y*)

lemma *upper-le-refl* [*simp*]: *t* ≤♯ *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*) *trans-less*)
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-less*: *PDPlus t u* $\leq\sharp$ *t*
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* $\leq\sharp$ *PDUnit b*) = $a \sqsubseteq b$
unfolding *upper-le-def Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-PDUnit-iff*:
(*PDPlus t u* $\leq\sharp$ *PDUnit a*) = ($t \leq\sharp$ *PDUnit a* $\vee$ $u \leq\sharp$ *PDUnit a*)
unfolding *upper-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-iff*: ($t \leq\sharp$ *PDPlus u v*) = ($t \leq\sharp u \wedge t \leq\sharp v$)
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-induct* [*induct set*: *upper-le*]:
assumes *le*: $t \leq\sharp u$
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P$ (*PDUnit a*) (*PDUnit b*)
assumes *2*: $\bigwedge t\ u\ a.\ P\ t$ (*PDUnit a*) $\Longrightarrow P$ (*PDPlus t u*) (*PDUnit a*)
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ t\ v]\!] \Longrightarrow P\ t$ (*PDPlus u v*)
shows *P t u*
using *le* **apply** (*induct u arbitrary*: *t rule*: *pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp add*: *1*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply (*simp add*: *2*)
apply (*subst PDPlus-commute*)

apply (*simp add*: *2*)
apply (*simp add*: *upper-le-PDPlus-iff 3*)
done

lemma *pd-take-upper-chain*:
  pd-take n t $\leq\sharp$ *pd-take* (*Suc n*) *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-chain*)
apply (*simp add*: *PDPlus-upper-mono*)
done

lemma *pd-take-upper-le*: *pd-take i t* $\leq\sharp$ *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-less*)
apply (*simp add*: *PDPlus-upper-mono*)
done

lemma *pd-take-upper-mono*:
  t $\leq\sharp$ *u* $\Longrightarrow$ *pd-take n t* $\leq\sharp$ *pd-take n u*
apply (*erule upper-le-induct*)
apply (*simp add*: *compact-basis.take-mono*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply (*simp add*: *upper-le-PDPlus-iff*)
done

25.2 Type definition

typedef (**open**) *'a upper-pd* =
  {*S*::*'a pd-basis set*. *upper-le.ideal S*}
by (*fast intro*: *upper-le.ideal-principal*)

instantiation *upper-pd* :: (*profinite*) *sq-ord*
begin

definition
  x $\sqsubseteq$ *y* $\longleftrightarrow$ *Rep-upper-pd x* $\subseteq$ *Rep-upper-pd y*

instance ..
end

instance *upper-pd* :: (*profinite*) *po*
by (*rule upper-le.typedef-ideal-po*
    [*OF type-definition-upper-pd sq-le-upper-pd-def*])

instance *upper-pd* :: (*profinite*) *cpo*
by (*rule upper-le.typedef-ideal-cpo*
    [*OF type-definition-upper-pd sq-le-upper-pd-def*])

lemma *Rep-upper-pd-lub*:

chain Y $\Longrightarrow$ *Rep-upper-pd* $(\bigsqcup i.\ Y\ i) = (\bigcup i.\ \textit{Rep-upper-pd}\ (Y\ i))$
by (*rule upper-le.typedef-ideal-rep-contlub*
[*OF type-definition-upper-pd sq-le-upper-pd-def*])

lemma *ideal-Rep-upper-pd*: *upper-le.ideal* (*Rep-upper-pd xs*)
by (*rule Rep-upper-pd* [*unfolded mem-Collect-eq*])

definition
upper-principal :: *'a pd-basis* $\Rightarrow$ *'a upper-pd* **where**
upper-principal t = *Abs-upper-pd* $\{u.\ u \leq\sharp\ t\}$

lemma *Rep-upper-principal*:
Rep-upper-pd (*upper-principal t*) = $\{u.\ u \leq\sharp\ t\}$
unfolding *upper-principal-def*
by (*simp add*: *Abs-upper-pd-inverse upper-le.ideal-principal*)

interpretation *upper-pd*:
ideal-completion upper-le pd-take upper-principal Rep-upper-pd
apply *unfold-locales*
apply (*rule pd-take-upper-le*)
apply (*rule pd-take-idem*)
apply (*erule pd-take-upper-mono*)
apply (*rule pd-take-upper-chain*)
apply (*rule finite-range-pd-take*)
apply (*rule pd-take-covers*)
apply (*rule ideal-Rep-upper-pd*)
apply (*erule Rep-upper-pd-lub*)
apply (*rule Rep-upper-principal*)
apply (*simp only*: *sq-le-upper-pd-def*)
done

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 UU-I*, *symmetric*])

Upper powerdomain is profinite

instantiation *upper-pd* :: (*profinite*) *profinite*
begin

definition
approx-upper-pd-def: *approx* = *upper-pd.completion-approx*

instance

apply (*intro-classes*, *unfold approx-upper-pd-def*)
apply (*rule upper-pd.chain-completion-approx*)
apply (*rule upper-pd.lub-completion-approx*)
apply (*rule upper-pd.completion-approx-idem*)
apply (*rule upper-pd.finite-fixes-completion-approx*)
done

end

instance *upper-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-upper-principal* [*simp*]:
approx n·(*upper-principal t*) = *upper-principal* (*pd-take n t*)
unfolding *approx-upper-pd-def*
by (*rule upper-pd.completion-approx-principal*)

lemma *approx-eq-upper-principal*:
∃ *t*∈*Rep-upper-pd xs*. *approx n·xs* = *upper-principal* (*pd-take n t*)
unfolding *approx-upper-pd-def*
by (*rule upper-pd.completion-approx-eq-principal*)

25.3 Monadic unit and plus

definition
upper-unit :: *'a* → *'a upper-pd* **where**
upper-unit = *compact-basis.basis-fun* (λ*a*. *upper-principal* (*PDUnit a*))

definition
upper-plus :: *'a upper-pd* → *'a upper-pd* → *'a upper-pd* **where**
upper-plus = *upper-pd.basis-fun* (λ*t*. *upper-pd.basis-fun* (λ*u*.
upper-principal (*PDPlus t u*)))

abbreviation
upper-add :: *'a upper-pd* ⇒ *'a upper-pd* ⇒ *'a upper-pd*
(**infixl** +♯ *65*) **where**
xs +♯ *ys* == *upper-plus·xs·ys*

syntax
-upper-pd :: *args* ⇒ *'a upper-pd* ({-}♯)

translations
{*x*,*xs*}♯ == {*x*}♯ +♯ {*xs*}♯
{*x*}♯ == *CONST upper-unit·x*

lemma *upper-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♯ = *upper-principal* (*PDUnit a*)
unfolding *upper-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal PDUnit-upper-mono*)

lemma *upper-plus-principal* [*simp*]:
upper-principal t +♯ *upper-principal u* = *upper-principal* (*PDPlus t u*)
unfolding *upper-plus-def*
by (*simp add*: *upper-pd.basis-fun-principal*
upper-pd.basis-fun-mono PDPlus-upper-mono)

lemma *approx-upper-unit* [*simp*]:
approx n·{*x*}♯ = {*approx n*·*x*}♯
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-upper-plus* [*simp*]:
approx n·(*xs* +♯ *ys*) = (*approx n*·*xs*) +♯ (*approx n*·*ys*)
by (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *upper-plus-assoc*: (*xs* +♯ *ys*) +♯ *zs* = *xs* +♯ (*ys* +♯ *zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *upper-pd.principal-induct2*, *simp*, *simp*)
apply (*rule-tac x*=*zs* **in** *upper-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done

lemma *upper-plus-commute*: *xs* +♯ *ys* = *ys* +♯ *xs*
apply (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done

lemma *upper-plus-absorb* [*simp*]: *xs* +♯ *xs* = *xs*
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done

lemma *upper-plus-left-commute*: *xs* +♯ (*ys* +♯ *zs*) = *ys* +♯ (*xs* +♯ *zs*)
by (*rule mk-left-commute* [*of op* +♯, *OF upper-plus-assoc upper-plus-commute*])

lemma *upper-plus-left-absorb* [*simp*]: *xs* +♯ (*xs* +♯ *ys*) = *xs* +♯ *ys*
by (*simp only*: *upper-plus-assoc* [*symmetric*] *upper-plus-absorb*)

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-less1*: *xs* +♯ *ys* ⊑ *xs*
apply (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-upper-less*)

done

lemma *upper-plus-less2*: $xs +\sharp ys \sqsubseteq ys$
by (*subst upper-plus-commute*, *rule upper-plus-less1*)

lemma *upper-plus-greatest*: $[\![xs \sqsubseteq ys; xs \sqsubseteq zs]\!] \Longrightarrow xs \sqsubseteq ys +\sharp zs$
apply (*subst upper-plus-absorb* [*of xs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *upper-less-plus-iff*:
$xs \sqsubseteq ys +\sharp zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
apply *safe*
apply (*erule trans-less* [*OF - upper-plus-less1*])
apply (*erule trans-less* [*OF - upper-plus-less2*])
apply (*erule* (*1*) *upper-plus-greatest*)
done

lemma *upper-plus-less-unit-iff*:
$xs +\sharp ys \sqsubseteq \{z\}\sharp \longleftrightarrow xs \sqsubseteq \{z\}\sharp \vee ys \sqsubseteq \{z\}\sharp$
apply (*rule iffI*)
apply (*subgoal-tac*
adm ($\lambda f.\ f{\cdot}xs \sqsubseteq f{\cdot}\{z\}\sharp \vee f{\cdot}ys \sqsubseteq f{\cdot}\{z\}\sharp$))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x=approx i·xs* **in** *upper-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·ys* **in** *upper-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·z* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*clarify*, *simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply *simp*
apply *simp*
apply (*erule disjE*)
apply (*erule trans-less* [*OF upper-plus-less1*])
apply (*erule trans-less* [*OF upper-plus-less2*])
done

lemma *upper-unit-less-iff* [*simp*]: $\{x\}\sharp \sqsubseteq \{y\}\sharp \longleftrightarrow x \sqsubseteq y$
apply (*rule iffI*)
apply (*rule profinite-less-ext*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
apply (*cut-tac x=approx i·x* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*cut-tac x=approx i·y* **in** *compact-basis.compact-imp-principal*, *simp*)
apply *clarsimp*
apply (*erule monofun-cfun-arg*)
done

lemmas *upper-pd-less-simps* =
upper-unit-less-iff
upper-less-plus-iff

upper-plus-less-unit-iff

lemma *upper-unit-eq-iff* [*simp*]: $\{x\}\sharp = \{y\}\sharp \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *upper-unit-strict* [*simp*]: $\{\bot\}\sharp = \bot$
unfolding *inst-upper-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *upper-plus-strict1* [*simp*]: $\bot +\sharp\ ys = \bot$
by (*rule UU-I*, *rule upper-plus-less1*)

lemma *upper-plus-strict2* [*simp*]: $xs +\sharp \bot = \bot$
by (*rule UU-I*, *rule upper-plus-less2*)

lemma *upper-unit-strict-iff* [*simp*]: $\{x\}\sharp = \bot \longleftrightarrow x = \bot$
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-unit-eq-iff*)

lemma *upper-plus-strict-iff* [*simp*]:
$xs +\sharp ys = \bot \longleftrightarrow xs = \bot \vee ys = \bot$
apply (*rule iffI*)
apply (*erule rev-mp*)
apply (*rule upper-pd.principal-induct2* [**where** *x*=*xs* **and** *y*=*ys*], *simp*, *simp*)
apply (*simp add*: *inst-upper-pd-pcpo upper-pd.principal-eq-iff*
upper-le-PDPlus-PDUnit-iff)
apply *auto*
done

lemma *compact-upper-unit-iff* [*simp*]: *compact* $\{x\}\sharp \longleftrightarrow$ *compact* x
unfolding *profinite-compact-iff* **by** *simp*

lemma *compact-upper-plus* [*simp*]:
$[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* $(xs +\sharp ys)$
by (*auto dest*!: *upper-pd.compact-imp-principal*)

25.4 Induction rules

lemma *upper-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\sharp;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\sharp +\sharp ys)$
shows *P* $(xs::'a\ upper\text{-}pd)$
apply (*induct xs rule*: *upper-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct1*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*]
upper-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *upper-pd-induct*:
  assumes *P*: *adm P*
  assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
  assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\sharp ys)$
  shows *P* (*xs*::'*a upper-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

25.5 Monadic bind

definition
  upper-bind-basis ::
  '*a pd-basis* ⇒ ('*a* → '*b upper-pd*) → '*b upper-pd* **where**
  upper-bind-basis = *fold-pd*
    (λ*a*. Λ *f*. *f*·(*Rep-compact-basis a*))
    (λ*x y*. Λ *f*. *x*·*f* +♯ *y*·*f*)

lemma *ACI-upper-bind*:
  ab-semigroup-idem-mult (λ*x y*. Λ *f*. *x*·*f* +♯ *y*·*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*) =
    (Λ *f*. *f*·(*Rep-compact-basis a*))
  upper-bind-basis (*PDPlus t u*) =
    (Λ *f*. *upper-bind-basis t*·*f* +♯ *upper-bind-basis u*·*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 ≤♯ *u* ⟹ *upper-bind-basis t* ⊑ *upper-bind-basis u*
unfolding *expand-cfun-less*
apply (*erule upper-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *trans-less* [*OF upper-plus-less1*])
apply (*simp add*: *upper-less-plus-iff*)
done

definition
upper-bind :: *'a upper-pd* → (*'a* → *'b upper-pd*) → *'b upper-pd* **where**
upper-bind = *upper-pd.basis-fun upper-bind-basis*

lemma *upper-bind-principal* [*simp*]:
upper-bind·(*upper-principal t*) = *upper-bind-basis t*
unfolding *upper-bind-def*
apply (*rule upper-pd.basis-fun-principal*)
apply (*erule upper-bind-basis-mono*)
done

lemma *upper-bind-unit* [*simp*]:
upper-bind·{*x*}♯·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *upper-bind-plus* [*simp*]:
upper-bind·(*xs* +♯ *ys*)·*f* = *upper-bind*·*xs*·*f* +♯ *upper-bind*·*ys*·*f*
by (*induct xs ys rule*: *upper-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *upper-bind-strict* [*simp*]: *upper-bind*·⊥·*f* = *f*·⊥
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-bind-unit*)

25.6 Map and join

definition
upper-map :: (*'a* → *'b*) → *'a upper-pd* → *'b upper-pd* **where**
upper-map = (Λ *f xs*. *upper-bind*·*xs*·(Λ *x*. {*f*·*x*}♯))

definition
upper-join :: *'a upper-pd upper-pd* → *'a upper-pd* **where**
upper-join = (Λ *xss*. *upper-bind*·*xss*·(Λ *xs*. *xs*))

lemma *upper-map-unit* [*simp*]:
upper-map·*f*·{*x*}♯ = {*f*·*x*}♯
unfolding *upper-map-def* **by** *simp*

lemma *upper-map-plus* [*simp*]:
upper-map·*f*·(*xs* +♯ *ys*) = *upper-map*·*f*·*xs* +♯ *upper-map*·*f*·*ys*
unfolding *upper-map-def* **by** *simp*

lemma *upper-join-unit* [*simp*]:
upper-join·{*xs*}♯ = *xs*
unfolding *upper-join-def* **by** *simp*

lemma *upper-join-plus* [*simp*]:
upper-join·(*xss* +♯ *yss*) = *upper-join*·*xss* +♯ *upper-join*·*yss*
unfolding *upper-join-def* **by** *simp*

lemma *upper-map-ident*: *upper-map*·(Λ *x*. *x*)·*xs* = *xs*

by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-map-map*:
upper-map·*f*·(*upper-map*·*g*·*xs*) = *upper-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-unit*:
upper-join·(*upper-map*·*upper-unit*·*xs*) = *xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-join*:
upper-join·(*upper-map*·*upper-join*·*xsss*) = *upper-join*·(*upper-join*·*xsss*)
by (*induct xsss rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-map*:
upper-join·(*upper-map*·(*upper-map*·*f*)·*xss*) =
upper-map·*f*·(*upper-join*·*xss*)
by (*induct xss rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-map-approx*: *upper-map*·(*approx n*)·*xs* = *approx n*·*xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

end

26 LowerPD: Lower powerdomain

theory *LowerPD*
imports *CompactBasis*
begin

26.1 Basis preorder

definition
lower-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♭ *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* ≤♭ *t*
unfolding *lower-le-def* **by** *fast*

lemma *lower-le-trans*: ⟦*t* ≤♭ *u*; *u* ≤♭ *v*⟧ ⟹ *t* ≤♭ *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*) *trans-less*)
done

interpretation *lower-le*: *preorder lower-le*
by (*rule preorder.intro*, *rule lower-le-refl*, *rule lower-le-trans*)

lemma *lower-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\flat$ *t*
unfolding *lower-le-def Rep-PDUnit*
by (*simp*, *rule Rep-pd-basis-nonempty* [*folded ex-in-conv*])

lemma *PDUnit-lower-mono*: $x \sqsubseteq y \Longrightarrow$ *PDUnit x* $\leq\flat$ *PDUnit y*
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *PDPlus-lower-mono*: $[\![s \leq\flat t;\ u \leq\flat v]\!] \Longrightarrow$ *PDPlus s u* $\leq\flat$ *PDPlus t v*
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-lower-less*: *t* $\leq\flat$ *PDPlus t u*
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* $\leq\flat$ *PDUnit b*) = *a* $\sqsubseteq$ *b*
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *lower-le-PDUnit-PDPlus-iff*:
(*PDUnit a* $\leq\flat$ *PDPlus t u*) = (*PDUnit a* $\leq\flat$ *t* $\vee$ *PDUnit a* $\leq\flat$ *u*)
unfolding *lower-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *lower-le-PDPlus-iff*: (*PDPlus t u* $\leq\flat$ *v*) = (*t* $\leq\flat$ *v* $\wedge$ *u* $\leq\flat$ *v*)
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-induct* [*induct set*: *lower-le*]:
assumes *le*: *t* $\leq\flat$ *u*
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P$ (*PDUnit a*) (*PDUnit b*)
assumes *2*: $\bigwedge t\ u\ a.\ P$ (*PDUnit a*) *t* $\Longrightarrow P$ (*PDUnit a*) (*PDPlus t u*)
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ v;\ P\ u\ v]\!] \Longrightarrow P$ (*PDPlus t u*) *v*
shows *P t u*
using *le*
apply (*induct t arbitrary*: *u rule*: *pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac u rule*: *pd-basis-induct*)
apply (*simp add*: *1*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
apply (*simp add*: *2*)
apply (*subst PDPlus-commute*)
apply (*simp add*: *2*)
apply (*simp add*: *lower-le-PDPlus-iff 3*)
done

lemma *pd-take-lower-chain*:
pd-take n t $\leq\flat$ *pd-take* (*Suc n*) *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-chain*)

apply (*simp add*: *PDPlus-lower-mono*)
done

lemma *pd-take-lower-le*: *pd-take i t* $\leq\flat$ *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-basis.take-less*)
apply (*simp add*: *PDPlus-lower-mono*)
done

lemma *pd-take-lower-mono*:
t $\leq\flat$ *u* $\Longrightarrow$ *pd-take n t* $\leq\flat$ *pd-take n u*
apply (*erule lower-le-induct*)
apply (*simp add*: *compact-basis.take-mono*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
apply (*simp add*: *lower-le-PDPlus-iff*)
done

26.2 Type definition

typedef (**open**) *'a lower-pd* =
{*S*::*'a pd-basis set*. *lower-le.ideal S*}
by (*fast intro*: *lower-le.ideal-principal*)

instantiation *lower-pd* :: (*profinite*) *sq-ord*
begin

definition
x $\sqsubseteq$ *y* $\longleftrightarrow$ *Rep-lower-pd x* $\subseteq$ *Rep-lower-pd y*

instance ..
end

instance *lower-pd* :: (*profinite*) *po*
by (*rule lower-le.typedef-ideal-po*
[*OF type-definition-lower-pd sq-le-lower-pd-def*])

instance *lower-pd* :: (*profinite*) *cpo*
by (*rule lower-le.typedef-ideal-cpo*
[*OF type-definition-lower-pd sq-le-lower-pd-def*])

lemma *Rep-lower-pd-lub*:
chain Y $\Longrightarrow$ *Rep-lower-pd* ($\bigsqcup i.\ Y\ i$) = ($\bigcup i.$ *Rep-lower-pd* (*Y i*))
by (*rule lower-le.typedef-ideal-rep-contlub*
[*OF type-definition-lower-pd sq-le-lower-pd-def*])

lemma *ideal-Rep-lower-pd*: *lower-le.ideal* (*Rep-lower-pd xs*)
by (*rule Rep-lower-pd* [*unfolded mem-Collect-eq*])

definition

lower-principal :: *'a pd-basis* ⇒ *'a lower-pd* **where**
lower-principal t = *Abs-lower-pd* {*u*. *u* ≤♭ *t*}

lemma *Rep-lower-principal*:
Rep-lower-pd (*lower-principal t*) = {*u*. *u* ≤♭ *t*}
unfolding *lower-principal-def*
by (*simp add*: *Abs-lower-pd-inverse lower-le.ideal-principal*)

interpretation *lower-pd*:
ideal-completion lower-le pd-take lower-principal Rep-lower-pd
apply *unfold-locales*
apply (*rule pd-take-lower-le*)
apply (*rule pd-take-idem*)
apply (*erule pd-take-lower-mono*)
apply (*rule pd-take-lower-chain*)
apply (*rule finite-range-pd-take*)
apply (*rule pd-take-covers*)
apply (*rule ideal-Rep-lower-pd*)
apply (*erule Rep-lower-pd-lub*)
apply (*rule Rep-lower-principal*)
apply (*simp only*: *sq-le-lower-pd-def*)
done

Lower powerdomain is pointed

lemma *lower-pd-minimal*: *lower-principal* (*PDUnit compact-bot*) ⊑ *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 UU-I*, *symmetric*])

Lower powerdomain is profinite

instantiation *lower-pd* :: (*profinite*) *profinite*
begin

definition
approx-lower-pd-def: *approx* = *lower-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-lower-pd-def*)
apply (*rule lower-pd.chain-completion-approx*)
apply (*rule lower-pd.lub-completion-approx*)
apply (*rule lower-pd.completion-approx-idem*)
apply (*rule lower-pd.finite-fixes-completion-approx*)
done

end

instance *lower-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-lower-principal* [*simp*]:
approx n·(*lower-principal t*) = *lower-principal* (*pd-take n t*)
unfolding *approx-lower-pd-def*
by (*rule lower-pd.completion-approx-principal*)

lemma *approx-eq-lower-principal*:
∃ *t*∈*Rep-lower-pd xs*. *approx n*·*xs* = *lower-principal* (*pd-take n t*)
unfolding *approx-lower-pd-def*
by (*rule lower-pd.completion-approx-eq-principal*)

26.3 Monadic unit and plus

definition
lower-unit :: *'a* → *'a lower-pd* **where**
lower-unit = *compact-basis.basis-fun* (λ*a*. *lower-principal* (*PDUnit a*))

definition
lower-plus :: *'a lower-pd* → *'a lower-pd* → *'a lower-pd* **where**
lower-plus = *lower-pd.basis-fun* (λ*t*. *lower-pd.basis-fun* (λ*u*.
lower-principal (*PDPlus t u*)))

abbreviation
lower-add :: *'a lower-pd* ⇒ *'a lower-pd* ⇒ *'a lower-pd*
(**infixl** +♭ *65*) **where**
xs +♭ *ys* == *lower-plus*·*xs*·*ys*

syntax
-lower-pd :: *args* ⇒ *'a lower-pd* ({-}♭)

translations
{*x*,*xs*}♭ == {*x*}♭ +♭ {*xs*}♭
{*x*}♭ == *CONST lower-unit*·*x*

lemma *lower-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♭ = *lower-principal* (*PDUnit a*)
unfolding *lower-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal PDUnit-lower-mono*)

lemma *lower-plus-principal* [*simp*]:
lower-principal t +♭ *lower-principal u* = *lower-principal* (*PDPlus t u*)
unfolding *lower-plus-def*
by (*simp add*: *lower-pd.basis-fun-principal*
lower-pd.basis-fun-mono PDPlus-lower-mono)

lemma *approx-lower-unit* [*simp*]:
approx n·{*x*}♭ = {*approx n*·*x*}♭

apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-lower-plus* [*simp*]:
  approx n·(*xs* +♭ *ys*) = (*approx n·xs*) +♭ (*approx n·ys*)
by (*induct xs ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *lower-plus-assoc*: (*xs* +♭ *ys*) +♭ *zs* = *xs* +♭ (*ys* +♭ *zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *lower-pd.principal-induct2*, *simp*, *simp*)
apply (*rule-tac x*=*zs* **in** *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done

lemma *lower-plus-commute*: *xs* +♭ *ys* = *ys* +♭ *xs*
apply (*induct xs ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done

lemma *lower-plus-absorb* [*simp*]: *xs* +♭ *xs* = *xs*
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done

lemma *lower-plus-left-commute*: *xs* +♭ (*ys* +♭ *zs*) = *ys* +♭ (*xs* +♭ *zs*)
by (*rule mk-left-commute* [*of op* +♭, *OF lower-plus-assoc lower-plus-commute*])

lemma *lower-plus-left-absorb* [*simp*]: *xs* +♭ (*xs* +♭ *ys*) = *xs* +♭ *ys*
by (*simp only*: *lower-plus-assoc* [*symmetric*] *lower-plus-absorb*)

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-less1*: *xs* ⊑ *xs* +♭ *ys*
apply (*induct xs ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-lower-less*)
done

lemma *lower-plus-less2*: *ys* ⊑ *xs* +♭ *ys*
by (*subst lower-plus-commute*, *rule lower-plus-less1*)

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-less-iff*:
  $xs +\flat ys \sqsubseteq zs \longleftrightarrow xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
apply *safe*
apply (*erule trans-less* [*OF lower-plus-less1*])
apply (*erule trans-less* [*OF lower-plus-less2*])
apply (*erule* (*1*) *lower-plus-least*)
done

lemma *lower-unit-less-plus-iff*:
  $\{x\}\flat \sqsubseteq ys +\flat zs \longleftrightarrow \{x\}\flat \sqsubseteq ys \vee \{x\}\flat \sqsubseteq zs$
 apply (*rule iffI*)
  apply (*subgoal-tac*
    adm ($\lambda f.\ f{\cdot}\{x\}\flat \sqsubseteq f{\cdot}ys \vee f{\cdot}\{x\}\flat \sqsubseteq f{\cdot}zs$))
   apply (*drule admD*, *rule chain-approx*)
    apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
    apply (*cut-tac x=approx i·x* **in** *compact-basis.compact-imp-principal*, *simp*)
    apply (*cut-tac x=approx i·ys* **in** *lower-pd.compact-imp-principal*, *simp*)
    apply (*cut-tac x=approx i·zs* **in** *lower-pd.compact-imp-principal*, *simp*)
    apply (*clarify*, *simp add*: *lower-le-PDUnit-PDPlus-iff*)
   apply *simp*
  apply *simp*
 apply (*erule disjE*)
  apply (*erule trans-less* [*OF - lower-plus-less1*])
 apply (*erule trans-less* [*OF - lower-plus-less2*])
done

lemma *lower-unit-less-iff* [*simp*]: $\{x\}\flat \sqsubseteq \{y\}\flat \longleftrightarrow x \sqsubseteq y$
 apply (*rule iffI*)
  apply (*rule profinite-less-ext*)
  apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
  apply (*cut-tac x=approx i·x* **in** *compact-basis.compact-imp-principal*, *simp*)
  apply (*cut-tac x=approx i·y* **in** *compact-basis.compact-imp-principal*, *simp*)
  apply *clarsimp*
 apply (*erule monofun-cfun-arg*)
done

lemmas *lower-pd-less-simps* =
  lower-unit-less-iff
  lower-plus-less-iff
  lower-unit-less-plus-iff

lemma *lower-unit-eq-iff* [*simp*]: $\{x\}\flat = \{y\}\flat \longleftrightarrow x = y$
by (*simp add*: *po-eq-conv*)

lemma *lower-unit-strict* [*simp*]: $\{\bot\}\flat = \bot$
unfolding *inst-lower-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *lower-unit-strict-iff* [*simp*]: $\{x\}\flat = \bot \longleftrightarrow x = \bot$
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-unit-eq-iff*)

lemma *lower-plus-strict-iff* [*simp*]:
$xs +\flat ys = \bot \longleftrightarrow xs = \bot \wedge ys = \bot$
apply *safe*
apply (*rule UU-I*, *erule subst*, *rule lower-plus-less1*)
apply (*rule UU-I*, *erule subst*, *rule lower-plus-less2*)
apply (*rule lower-plus-absorb*)
done

lemma *lower-plus-strict1* [*simp*]: $\bot +\flat ys = ys$
apply (*rule antisym-less* [*OF - lower-plus-less2*])
apply (*simp add*: *lower-plus-least*)
done

lemma *lower-plus-strict2* [*simp*]: $xs +\flat \bot = xs$
apply (*rule antisym-less* [*OF - lower-plus-less1*])
apply (*simp add*: *lower-plus-least*)
done

lemma *compact-lower-unit-iff* [*simp*]: *compact* $\{x\}\flat \longleftrightarrow$ *compact* x
unfolding *profinite-compact-iff* **by** *simp*

lemma *compact-lower-plus* [*simp*]:
$[\![$*compact* xs; *compact* $ys]\!] \Longrightarrow$ *compact* $(xs +\flat ys)$
by (*auto dest*!: *lower-pd.compact-imp-principal*)

26.4 Induction rules

lemma *lower-pd-induct1*:
assumes P: *adm* P
assumes *unit*: $\bigwedge x.\ P\ \{x\}\flat$
assumes *insert*:
$\bigwedge x\ ys.\ [\![P\ \{x\}\flat; P\ ys]\!] \Longrightarrow P\ (\{x\}\flat +\flat ys)$
shows P (xs::'*a lower-pd*)
apply (*induct xs rule*: *lower-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct1*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*]
lower-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *lower-pd-induct*:
assumes P: *adm* P
assumes *unit*: $\bigwedge x.\ P\ \{x\}\flat$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs; P\ ys]\!] \Longrightarrow P\ (xs +\flat ys)$

shows P (xs::$'a$ *lower-pd*)
apply (*induct xs rule*: *lower-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only*: *lower-plus-principal* [*symmetric*] *plus*)
done

26.5 Monadic bind

definition
lower-bind-basis ::
$'a$ *pd-basis* $\Rightarrow$ ($'a \rightarrow$ $'b$ *lower-pd*) $\rightarrow$ $'b$ *lower-pd* **where**
lower-bind-basis = *fold-pd*
($\lambda a.\ \Lambda\ f.\ f{\cdot}$(*Rep-compact-basis a*))
($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f\ +\flat\ y{\cdot}f$)

lemma *ACI-lower-bind*:
ab-semigroup-idem-mult ($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f\ +\flat\ y{\cdot}f$)
apply *unfold-locales*
apply (*simp add*: *lower-plus-assoc*)
apply (*simp add*: *lower-plus-commute*)
apply (*simp add*: *eta-cfun*)
done

lemma *lower-bind-basis-simps* [*simp*]:
lower-bind-basis (*PDUnit a*) =
($\Lambda\ f.\ f{\cdot}$(*Rep-compact-basis a*))
lower-bind-basis (*PDPlus t u*) =
($\Lambda\ f.$ *lower-bind-basis* $t{\cdot}f\ +\flat$ *lower-bind-basis* $u{\cdot}f$)
unfolding *lower-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-lower-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-lower-bind*])
done

lemma *lower-bind-basis-mono*:
$t \leq\flat\ u \Longrightarrow$ *lower-bind-basis* $t \sqsubseteq$ *lower-bind-basis u*
unfolding *expand-cfun-less*
apply (*erule lower-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *rev-trans-less* [*OF lower-plus-less1*])
apply (*simp add*: *lower-plus-less-iff*)
done

definition
lower-bind :: $'a$ *lower-pd* $\rightarrow$ ($'a \rightarrow$ $'b$ *lower-pd*) $\rightarrow$ $'b$ *lower-pd* **where**
lower-bind = *lower-pd.basis-fun lower-bind-basis*

lemma *lower-bind-principal* [*simp*]:

lower-bind·(*lower-principal t*) = *lower-bind-basis t*
unfolding *lower-bind-def*
apply (*rule lower-pd.basis-fun-principal*)
apply (*erule lower-bind-basis-mono*)
done

lemma *lower-bind-unit* [*simp*]:
lower-bind·{*x*}♭·*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 ys rule*: *lower-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *lower-bind-strict* [*simp*]: *lower-bind*·⊥·*f* = *f*·⊥
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-bind-unit*)

26.6 Map and join

definition
lower-map :: ('*a* → '*b*) → '*a lower-pd* → '*b lower-pd* **where**
lower-map = (Λ *f xs*. *lower-bind*·*xs*·(Λ *x*. {*f*·*x*}♭))

definition
lower-join :: '*a lower-pd lower-pd* → '*a lower-pd* **where**
lower-join = (Λ *xss*. *lower-bind*·*xss*·(Λ *xs*. *xs*))

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-join-unit* [*simp*]:
lower-join·{*xs*}♭ = *xs*
unfolding *lower-join-def* **by** *simp*

lemma *lower-join-plus* [*simp*]:
lower-join·(*xss* +♭ *yss*) = *lower-join*·*xss* +♭ *lower-join*·*yss*
unfolding *lower-join-def* **by** *simp*

lemma *lower-map-ident*: *lower-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-map-map*:
lower-map·*f*·(*lower-map*·*g*·*xs*) = *lower-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-unit*:
lower-join·(*lower-map*·*lower-unit*·*xs*) = *xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-join*:
lower-join·(*lower-map*·*lower-join*·*xsss*) = *lower-join*·(*lower-join*·*xsss*)
by (*induct xsss rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-map*:
lower-join·(*lower-map*·(*lower-map*·*f*)·*xss*) =
lower-map·*f*·(*lower-join*·*xss*)
by (*induct xss rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-map-approx*: *lower-map*·(*approx n*)·*xs* = *approx n*·*xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

end

27 ConvexPD: Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

27.1 Basis preorder

definition
convex-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♮ *50*) **where**
convex-le = (λ*u v*. *u* ≤♯ *v* ∧ *u* ≤♭ *v*)

lemma *convex-le-refl* [*simp*]: *t* ≤♮ *t*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-refl lower-le-refl*)

lemma *convex-le-trans*: ⟦*t* ≤♮ *u*; *u* ≤♮ *v*⟧ ⟹ *t* ≤♮ *v*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-trans lower-le-trans*)

interpretation *convex-le*: *preorder convex-le*
by (*rule preorder.intro*, *rule convex-le-refl*, *rule convex-le-trans*)

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* ≤♮ *t*
unfolding *convex-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-convex-mono*: *x* ⊑ *y* ⟹ *PDUnit x* ≤♮ *PDUnit y*
unfolding *convex-le-def* **by** (*fast intro*: *PDUnit-upper-mono PDUnit-lower-mono*)

lemma *PDPlus-convex-mono*: ⟦*s* ≤♮ *t*; *u* ≤♮ *v*⟧ ⟹ *PDPlus s u* ≤♮ *PDPlus t v*
unfolding *convex-le-def* **by** (*fast intro*: *PDPlus-upper-mono PDPlus-lower-mono*)

lemma *convex-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* $\leq\natural$ *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* $\leq\natural$ *t*) = ($\forall$ *b*$\in$*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* $\leq\natural$ *PDPlus t u*) = (*PDUnit a* $\leq\natural$ *t* $\wedge$ *PDUnit a* $\leq\natural$ *u*)
unfolding *convex-le-PDUnit-lemma1 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDUnit-lemma2*:
(*t* $\leq\natural$ *PDUnit b*) = ($\forall$ *a*$\in$*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* $\leq\natural$ *PDUnit a*) = (*t* $\leq\natural$ *PDUnit a* $\wedge$ *u* $\leq\natural$ *PDUnit a*)
unfolding *convex-le-PDUnit-lemma2 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDPlus-lemma*:
assumes *z*: *PDPlus t u* $\leq\natural$ *z*
shows $\exists$ *v w*. *z* = *PDPlus v w* $\wedge$ *t* $\leq\natural$ *v* $\wedge$ *u* $\leq\natural$ *w*
proof (*intro exI conjI*)
let *?A* = {*b*$\in$*Rep-pd-basis z*. $\exists$ *a*$\in$*Rep-pd-basis t*. *a* $\sqsubseteq$ *b*}
let *?B* = {*b*$\in$*Rep-pd-basis z*. $\exists$ *a*$\in$*Rep-pd-basis u*. *a* $\sqsubseteq$ *b*}
let *?v* = *Abs-pd-basis ?A*
let *?w* = *Abs-pd-basis ?B*
have *Rep-v*: *Rep-pd-basis ?v* = *?A*
apply (*rule Abs-pd-basis-inverse*)
apply (*rule Rep-pd-basis-nonempty* [*of t*, *folded ex-in-conv*, *THEN exE*])
apply (*cut-tac z*, *simp only*: *convex-le-def lower-le-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *bspec*, *simp add*: *Rep-PDPlus*, *erule bexE*)
apply (*simp add*: *pd-basis-def*)
apply *fast*
done
have *Rep-w*: *Rep-pd-basis ?w* = *?B*
apply (*rule Abs-pd-basis-inverse*)
apply (*rule Rep-pd-basis-nonempty* [*of u*, *folded ex-in-conv*, *THEN exE*])
apply (*cut-tac z*, *simp only*: *convex-le-def lower-le-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *bspec*, *simp add*: *Rep-PDPlus*, *erule bexE*)
apply (*simp add*: *pd-basis-def*)
apply *fast*
done
show *z* = *PDPlus ?v ?w*
apply (*insert z*)

```
    apply (simp add: convex-le-def, erule conjE)
    apply (simp add: Rep-pd-basis-inject [symmetric] Rep-PDPlus)
    apply (simp add: Rep-v Rep-w)
    apply (rule equalityI)
     apply (rule subsetI)
     apply (simp only: upper-le-def)
     apply (drule (1) bspec, erule bexE)
     apply (simp add: Rep-PDPlus)
     apply fast
    apply fast
    done
  show t ≤♮ ?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: ⋀t u v. ⟦P t u; P u v⟧ ⟹ P t v
  assumes 3: ⋀a b. a ⊑ b ⟹ P (PDUnit a) (PDUnit b)
  assumes 4: ⋀t u v w. ⟦P t v; P u w⟧ ⟹ P (PDPlus t u) (PDPlus v w)
  shows P t u
using le apply (induct t arbitrary: u rule: pd-basis-induct)
apply (erule rev-mp)
apply (induct-tac u rule: pd-basis-induct1)
apply (simp add: 3)
apply (simp, clarify, rename-tac a b t)
apply (subgoal-tac P (PDPlus (PDUnit a) (PDUnit a)) (PDPlus (PDUnit b) t))
apply (simp add: PDPlus-absorb)
apply (erule (1) 4 [OF 3])
apply (drule convex-le-PDPlus-lemma, clarify)
apply (simp add: 4)
done

lemma pd-take-convex-chain:
  pd-take n t ≤♮ pd-take (Suc n) t
apply (induct t rule: pd-basis-induct)
apply (simp add: compact-basis.take-chain)
apply (simp add: PDPlus-convex-mono)
done

lemma pd-take-convex-le: pd-take i t ≤♮ t
apply (induct t rule: pd-basis-induct)
apply (simp add: compact-basis.take-less)
apply (simp add: PDPlus-convex-mono)
done
```

lemma *pd-take-convex-mono*:
  $t \leq\natural u \Longrightarrow$ *pd-take n t* $\leq\natural$ *pd-take n u*
apply (*erule convex-le-induct*)
apply (*erule* (*1*) *convex-le-trans*)
apply (*simp add*: *compact-basis.take-mono*)
apply (*simp add*: *PDPlus-convex-mono*)
done

27.2 Type definition

typedef (**open**) *'a convex-pd* =
  {*S*::*'a pd-basis set*. *convex-le.ideal S*}
by (*fast intro*: *convex-le.ideal-principal*)

instantiation *convex-pd* :: (*profinite*) *sq-ord*
begin

definition
  $x \sqsubseteq y \longleftrightarrow$ *Rep-convex-pd x* $\subseteq$ *Rep-convex-pd y*

instance ..
end

instance *convex-pd* :: (*profinite*) *po*
by (*rule convex-le.typedef-ideal-po*
    [*OF type-definition-convex-pd sq-le-convex-pd-def*])

instance *convex-pd* :: (*profinite*) *cpo*
by (*rule convex-le.typedef-ideal-cpo*
    [*OF type-definition-convex-pd sq-le-convex-pd-def*])

lemma *Rep-convex-pd-lub*:
  chain Y $\Longrightarrow$ *Rep-convex-pd* ($\bigsqcup i.\ Y\ i$) = ($\bigcup i.$ *Rep-convex-pd* (*Y i*))
by (*rule convex-le.typedef-ideal-rep-contlub*
    [*OF type-definition-convex-pd sq-le-convex-pd-def*])

lemma *ideal-Rep-convex-pd*: *convex-le.ideal* (*Rep-convex-pd xs*)
by (*rule Rep-convex-pd* [*unfolded mem-Collect-eq*])

definition
  convex-principal :: *'a pd-basis* $\Rightarrow$ *'a convex-pd* **where**
  convex-principal t = *Abs-convex-pd* {*u*. *u* $\leq\natural$ *t*}

lemma *Rep-convex-principal*:
  Rep-convex-pd (*convex-principal t*) = {*u*. *u* $\leq\natural$ *t*}
unfolding *convex-principal-def*
by (*simp add*: *Abs-convex-pd-inverse convex-le.ideal-principal*)

interpretation *convex-pd*:
ideal-completion convex-le pd-take convex-principal Rep-convex-pd
apply *unfold-locales*
apply (*rule pd-take-convex-le*)
apply (*rule pd-take-idem*)
apply (*erule pd-take-convex-mono*)
apply (*rule pd-take-convex-chain*)
apply (*rule finite-range-pd-take*)
apply (*rule pd-take-covers*)
apply (*rule ideal-Rep-convex-pd*)
apply (*erule Rep-convex-pd-lub*)
apply (*rule Rep-convex-principal*)
apply (*simp only*: *sq-le-convex-pd-def*)
done

Convex powerdomain is pointed

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*
by (*induct ys rule*: *convex-pd.principal-induct*, *simp*, *simp*)

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

lemma *inst-convex-pd-pcpo*: $\bot$ = *convex-principal* (*PDUnit compact-bot*)
by (*rule convex-pd-minimal* [*THEN UU-I*, *symmetric*])

Convex powerdomain is profinite

instantiation *convex-pd* :: (*profinite*) *profinite*
begin

definition
approx-convex-pd-def: *approx* = *convex-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-convex-pd-def*)
apply (*rule convex-pd.chain-completion-approx*)
apply (*rule convex-pd.lub-completion-approx*)
apply (*rule convex-pd.completion-approx-idem*)
apply (*rule convex-pd.finite-fixes-completion-approx*)
done

end

instance *convex-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-convex-principal* [*simp*]:
approx n·(*convex-principal t*) = *convex-principal* (*pd-take n t*)
unfolding *approx-convex-pd-def*
by (*rule convex-pd.completion-approx-principal*)

lemma *approx-eq-convex-principal*:
∃ *t*∈*Rep-convex-pd xs*. *approx n*·*xs* = *convex-principal* (*pd-take n t*)
unfolding *approx-convex-pd-def*
by (*rule convex-pd.completion-approx-eq-principal*)

27.3 Monadic unit and plus

definition
convex-unit :: *'a* → *'a convex-pd* **where**
convex-unit = *compact-basis.basis-fun* (λ*a*. *convex-principal* (*PDUnit a*))

definition
convex-plus :: *'a convex-pd* → *'a convex-pd* → *'a convex-pd* **where**
convex-plus = *convex-pd.basis-fun* (λ*t*. *convex-pd.basis-fun* (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* ⇒ *'a convex-pd* ({-}♮)

translations
{*x*,*xs*}♮ == {*x*}♮ +♮ {*xs*}♮
{*x*}♮ == *CONST convex-unit*·*x*

lemma *convex-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♮ = *convex-principal* (*PDUnit a*)
unfolding *convex-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal PDUnit-convex-mono*)

lemma *convex-plus-principal* [*simp*]:
convex-principal t +♮ *convex-principal u* = *convex-principal* (*PDPlus t u*)
unfolding *convex-plus-def*
by (*simp add*: *convex-pd.basis-fun-principal*
convex-pd.basis-fun-mono PDPlus-convex-mono)

lemma *approx-convex-unit* [*simp*]:
approx n·{*x*}♮ = {*approx n*·*x*}♮
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-convex-plus* [*simp*]:
approx n·(*xs* +♮ *ys*) = *approx n*·*xs* +♮ *approx n*·*ys*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *convex-plus-assoc*:
(*xs* +♮ *ys*) +♮ *zs* = *xs* +♮ (*ys* +♮ *zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *convex-pd.principal-induct2*, *simp*, *simp*)
apply (*rule-tac x*=*zs* **in** *convex-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done

lemma *convex-plus-commute*: *xs* +♮ *ys* = *ys* +♮ *xs*
apply (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done

lemma *convex-plus-absorb* [*simp*]: *xs* +♮ *xs* = *xs*
apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done

lemma *convex-plus-left-commute*: *xs* +♮ (*ys* +♮ *zs*) = *ys* +♮ (*xs* +♮ *zs*)
by (*rule mk-left-commute*
[*of op* +♮, *OF convex-plus-assoc convex-plus-commute*])

lemma *convex-plus-left-absorb* [*simp*]: *xs* +♮ (*xs* +♮ *ys*) = *xs* +♮ *ys*
by (*simp only*: *convex-plus-assoc* [*symmetric*] *convex-plus-absorb*)

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-less-plus-iff* [*simp*]:
{*x*}♮ ⊑ *ys* +♮ *zs* ⟷ {*x*}♮ ⊑ *ys* ∧ {*x*}♮ ⊑ *zs*
apply (*rule iffI*)
apply (*subgoal-tac*
adm (λ*f*. *f*·{*x*}♮ ⊑ *f*·*ys* ∧ *f*·{*x*}♮ ⊑ *f*·*zs*))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x*=*approx i*·*x* **in** *compact-basis.compact-imp-principal*, *simp*)
apply (*cut-tac x*=*approx i*·*ys* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x*=*approx i*·*zs* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*clarify*, *simp*)
apply *simp*
apply *simp*
apply (*erule conjE*)
apply (*subst convex-plus-absorb* [*of* {*x*}♮, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *convex-plus-less-unit-iff* [*simp*]:
  $xs +\natural ys \sqsubseteq \{z\}\natural \longleftrightarrow xs \sqsubseteq \{z\}\natural \wedge ys \sqsubseteq \{z\}\natural$
 apply (*rule iffI*)
  apply (*subgoal-tac*
    adm ($\lambda f.\ f{\cdot}xs \sqsubseteq f{\cdot}\{z\}\natural \wedge f{\cdot}ys \sqsubseteq f{\cdot}\{z\}\natural$))
   apply (*drule admD*, *rule chain-approx*)
    apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
    apply (*cut-tac x=approx i·xs* **in** *convex-pd.compact-imp-principal*, *simp*)
    apply (*cut-tac x=approx i·ys* **in** *convex-pd.compact-imp-principal*, *simp*)
    apply (*cut-tac x=approx i·z* **in** *compact-basis.compact-imp-principal*, *simp*)
    apply (*clarify*, *simp*)
   apply *simp*
  apply *simp*
 apply (*erule conjE*)
 apply (*subst convex-plus-absorb* [*of* $\{z\}\natural$, *symmetric*])
 apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *convex-unit-less-iff* [*simp*]: $\{x\}\natural \sqsubseteq \{y\}\natural \longleftrightarrow x \sqsubseteq y$
 apply (*rule iffI*)
  apply (*rule profinite-less-ext*)
  apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
  apply (*cut-tac x=approx i·x* **in** *compact-basis.compact-imp-principal*, *simp*)
  apply (*cut-tac x=approx i·y* **in** *compact-basis.compact-imp-principal*, *simp*)
  apply *clarsimp*
 apply (*erule monofun-cfun-arg*)
done

lemma *convex-unit-eq-iff* [*simp*]: $\{x\}\natural = \{y\}\natural \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *convex-unit-strict* [*simp*]: $\{\bot\}\natural = \bot$
unfolding *inst-convex-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *convex-unit-strict-iff* [*simp*]: $\{x\}\natural = \bot \longleftrightarrow x = \bot$
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-unit-eq-iff*)

lemma *compact-convex-unit-iff* [*simp*]:
  compact $\{x\}\natural \longleftrightarrow$ *compact x*
unfolding *profinite-compact-iff* **by** *simp*

lemma *compact-convex-plus* [*simp*]:
  $[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* ($xs +\natural ys$)
by (*auto dest*!: *convex-pd.compact-imp-principal*)

27.4 Induction rules

lemma *convex-pd-induct1*:

assumes P: $adm\ P$
assumes $unit$: $\bigwedge x.\ P\ \{x\}\natural$
assumes $insert$: $\bigwedge x\ ys.\ [\![P\ \{x\}\natural;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\natural +\natural\ ys)$
shows $P\ (xs{::}'a\ convex\text{-}pd)$
apply (*induct xs rule*: *convex-pd.principal-induct*, *rule P*)
apply (*induct-tac a rule*: *pd-basis-induct1*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*]
convex-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *convex-pd-induct*:
assumes P: $adm\ P$
assumes $unit$: $\bigwedge x.\ P\ \{x\}\natural$
assumes $plus$: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\natural\ ys)$
shows $P\ (xs{::}'a\ convex\text{-}pd)$
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

27.5 Monadic bind

definition
convex-bind-basis ::
$'a\ pd\text{-}basis \Rightarrow ('a \to 'b\ convex\text{-}pd) \to 'b\ convex\text{-}pd$ **where**
convex-bind-basis = *fold-pd*
$(\lambda a.\ \Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\natural\ y{\cdot}f)$

lemma *ACI-convex-bind*:
ab-semigroup-idem-mult $(\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\natural\ y{\cdot}f)$
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 $(PDUnit\ a) =$
$(\Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a))$
convex-bind-basis $(PDPlus\ t\ u) =$
$(\Lambda\ f.\ convex\text{-}bind\text{-}basis\ t{\cdot}f +\natural\ convex\text{-}bind\text{-}basis\ u{\cdot}f)$
unfolding *convex-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-convex-bind*])

apply (*rule fold-pd-PDPlus* [*OF ACI-convex-bind*])
done

lemma *monofun-LAM*:
⟦*cont f*; *cont g*; ⋀*x*. *f x* ⊑ *g x*⟧ ⟹ (Λ *x*. *f x*) ⊑ (Λ *x*. *g x*)
by (*simp add*: *expand-cfun-less*)

lemma *convex-bind-basis-mono*:
t ≤♮ *u* ⟹ *convex-bind-basis t* ⊑ *convex-bind-basis u*
apply (*erule convex-le-induct*)
apply (*erule* (*1*) *trans-less*)
apply (*simp add*: *monofun-LAM monofun-cfun*)
apply (*simp add*: *monofun-LAM monofun-cfun*)
done

definition
convex-bind :: 'a *convex-pd* → ('a → 'b *convex-pd*) → 'b *convex-pd* **where**
convex-bind = *convex-pd.basis-fun convex-bind-basis*

lemma *convex-bind-principal* [*simp*]:
convex-bind·(*convex-principal t*) = *convex-bind-basis t*
unfolding *convex-bind-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*erule convex-bind-basis-mono*)
done

lemma *convex-bind-unit* [*simp*]:
convex-bind·{*x*}♮·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-bind-plus* [*simp*]:
convex-bind·(*xs* +♮ *ys*)·*f* = *convex-bind*·*xs*·*f* +♮ *convex-bind*·*ys*·*f*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *convex-bind-strict* [*simp*]: *convex-bind*·⊥·*f* = *f*·⊥
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-bind-unit*)

27.6 Map and join

definition
convex-map :: ('a → 'b) → 'a *convex-pd* → 'b *convex-pd* **where**
convex-map = (Λ *f xs*. *convex-bind*·*xs*·(Λ *x*. {*f*·*x*}♮))

definition
convex-join :: 'a *convex-pd convex-pd* → 'a *convex-pd* **where**
convex-join = (Λ *xss*. *convex-bind*·*xss*·(Λ *xs*. *xs*))

lemma *convex-map-unit* [*simp*]:
convex-map·*f*·(*convex-unit*·*x*) = *convex-unit*·(*f*·*x*)

unfolding *convex-map-def* **by** *simp*

lemma *convex-map-plus* [*simp*]:
convex-map·f·(xs +♮ ys) = convex-map·f·xs +♮ convex-map·f·ys
unfolding *convex-map-def* **by** *simp*

lemma *convex-join-unit* [*simp*]:
convex-join·{xs}♮ = xs
unfolding *convex-join-def* **by** *simp*

lemma *convex-join-plus* [*simp*]:
convex-join·(xss +♮ yss) = convex-join·xss +♮ convex-join·yss
unfolding *convex-join-def* **by** *simp*

lemma *convex-map-ident*: *convex-map·(Λ x. x)·xs = xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-map-map*:
convex-map·f·(convex-map·g·xs) = convex-map·(Λ x. f·(g·x))·xs
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-unit*:
convex-join·(convex-map·convex-unit·xs) = xs
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-join*:
convex-join·(convex-map·convex-join·xsss) = convex-join·(convex-join·xsss)
by (*induct xsss rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-map*:
convex-join·(convex-map·(convex-map·f)·xss) =
convex-map·f·(convex-join·xss)
by (*induct xss rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-map-approx*: *convex-map·(approx n)·xs = approx n·xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

27.7 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: *t ≤♮ u ⟹ t ≤♯ u*
unfolding *convex-le-def* **by** *simp*

definition
convex-to-upper :: *'a convex-pd → 'a upper-pd* **where**
convex-to-upper = convex-pd.basis-fun upper-principal

lemma *convex-to-upper-principal* [*simp*]:
convex-to-upper·(convex-principal t) = upper-principal t

unfolding *convex-to-upper-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*rule upper-pd.principal-mono*)
apply (*erule convex-le-imp-upper-le*)
done

lemma *convex-to-upper-unit* [*simp*]:
convex-to-upper·{*x*}♮ = {*x*}♯
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-to-upper-plus* [*simp*]:
convex-to-upper·(*xs* +♮ *ys*) = *convex-to-upper*·*xs* +♯ *convex-to-upper*·*ys*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *approx-convex-to-upper*:
approx i·(*convex-to-upper*·*xs*) = *convex-to-upper*·(*approx i*·*xs*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-upper-bind* [*simp*]:
convex-to-upper·(*convex-bind*·*xs*·*f*) =
upper-bind·(*convex-to-upper*·*xs*)·(*convex-to-upper oo f*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-upper-map* [*simp*]:
convex-to-upper·(*convex-map*·*f*·*xs*) = *upper-map*·*f*·(*convex-to-upper*·*xs*)
by (*simp add*: *convex-map-def upper-map-def cfcomp-LAM*)

lemma *convex-to-upper-join* [*simp*]:
convex-to-upper·(*convex-join*·*xss*) =
upper-bind·(*convex-to-upper*·*xss*)·*convex-to-upper*
by (*simp add*: *convex-join-def upper-join-def cfcomp-LAM eta-cfun*)

Convex to lower

lemma *convex-le-imp-lower-le*: *t* ≤♮ *u* ⟹ *t* ≤♭ *u*
unfolding *convex-le-def* **by** *simp*

definition
convex-to-lower :: *'a convex-pd* → *'a lower-pd* **where**
convex-to-lower = *convex-pd.basis-fun lower-principal*

lemma *convex-to-lower-principal* [*simp*]:
convex-to-lower·(*convex-principal t*) = *lower-principal t*
unfolding *convex-to-lower-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*rule lower-pd.principal-mono*)
apply (*erule convex-le-imp-lower-le*)
done

lemma *convex-to-lower-unit* [*simp*]:

convex-to-lower·{*x*}♮ = {*x*}♭
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-to-lower-plus* [*simp*]:
convex-to-lower·(*xs* +♮ *ys*) = *convex-to-lower*·*xs* +♭ *convex-to-lower*·*ys*
by (*induct xs ys rule*: *convex-pd.principal-induct2*, *simp*, *simp*, *simp*)

lemma *approx-convex-to-lower*:
approx i·(*convex-to-lower*·*xs*) = *convex-to-lower*·(*approx i*·*xs*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-lower-bind* [*simp*]:
convex-to-lower·(*convex-bind*·*xs*·*f*) =
lower-bind·(*convex-to-lower*·*xs*)·(*convex-to-lower oo f*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-lower-map* [*simp*]:
convex-to-lower·(*convex-map*·*f*·*xs*) = *lower-map*·*f*·(*convex-to-lower*·*xs*)
by (*simp add*: *convex-map-def lower-map-def cfcomp-LAM*)

lemma *convex-to-lower-join* [*simp*]:
convex-to-lower·(*convex-join*·*xss*) =
lower-bind·(*convex-to-lower*·*xss*)·*convex-to-lower*
by (*simp add*: *convex-join-def lower-join-def cfcomp-LAM eta-cfun*)

Ordering property

lemma *convex-pd-less-iff*:
(*xs* ⊑ *ys*) =
(*convex-to-upper*·*xs* ⊑ *convex-to-upper*·*ys* ∧
convex-to-lower·*xs* ⊑ *convex-to-lower*·*ys*)
apply (*safe elim*!: *monofun-cfun-arg*)
apply (*rule profinite-less-ext*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x*=*approx i*·*xs* **in** *convex-pd.compact-imp-principal*, *simp*)
apply (*cut-tac x*=*approx i*·*ys* **in** *convex-pd.compact-imp-principal*, *simp*)
apply *clarify*
apply (*simp add*: *approx-convex-to-upper approx-convex-to-lower convex-le-def*)
done

lemmas *convex-plus-less-plus-iff* =
convex-pd-less-iff [**where** *xs*=*xs* +♮ *ys* **and** *ys*=*zs* +♮ *ws*, *standard*]

lemmas *convex-pd-less-simps* =
convex-unit-less-plus-iff
convex-plus-less-unit-iff
convex-plus-less-plus-iff
convex-unit-less-iff
convex-to-upper-unit

convex-to-upper-plus
convex-to-lower-unit
convex-to-lower-plus
upper-pd-less-simps
lower-pd-less-simps

end

28 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

28.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
infinite :: *'a set* ⇒ *bool* **where**
infinite S == ¬ *finite S*

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-imp-nonempty*: *infinite S* ==> $S \neq \{\}$
by *auto*

lemma *infinite-remove*:
infinite S ⟹ *infinite* $(S - \{a\})$
by *simp*

lemma *Diff-infinite-finite*:
assumes *T*: *finite T* **and** *S*: *infinite S*
shows *infinite* $(S - T)$
using *T*
proof *induct*
from *S*
show *infinite* $(S - \{\})$ **by** *auto*
next
fix *T x*
assume *ih*: *infinite* $(S - T)$
have $S - (insert\ x\ T) = (S - T) - \{x\}$
by (*rule Diff-insert*)
with *ih*

```
  show infinite (S − (insert x T))
    by (simp add: infinite-remove)
qed

lemma Un-infinite: infinite S ⟹ infinite (S ∪ T)
  by simp

lemma infinite-super:
  assumes T: S ⊆ T and S: infinite S
  shows infinite T
proof
  assume finite T
  with T have finite S by (simp add: finite-subset)
  with S show False by simp
qed
```

As a concrete example, we prove that the set of natural numbers is infinite.

```
lemma finite-nat-bounded:
  assumes S: finite (S::nat set)
  shows ∃ k. S ⊆ {..<k}  (is ∃ k. ?bounded S k)
using S
proof induct
  have ?bounded {} 0 by simp
  then show ∃ k. ?bounded {} k ..
next
  fix S x
  assume ∃ k. ?bounded S k
  then obtain k where k: ?bounded S k ..
  show ∃ k. ?bounded (insert x S) k
  proof (cases x < k)
    case True
    with k show ?thesis by auto
  next
    case False
    with k have ?bounded S (Suc x) by auto
    then show ?thesis by auto
  qed
qed

lemma finite-nat-iff-bounded:
  finite (S::nat set) = (∃ k. S ⊆ {..<k})  (is ?lhs = ?rhs)
proof
  assume ?lhs
  then show ?rhs by (rule finite-nat-bounded)
next
  assume ?rhs
  then obtain k where S ⊆ {..<k} ..
  then show finite S
    by (rule finite-subset) simp
```

qed

lemma *finite-nat-iff-bounded-le*:
finite (*S*::*nat set*) = (∃ *k*. *S* ⊆ {..*k*}) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *k* **where** *S* ⊆ {..<*k*}
by (*blast dest*: *finite-nat-bounded*)
then have *S* ⊆ {..*k*} **by** *auto*
then show *?rhs* **..**
next
assume *?rhs*
then obtain *k* **where** *S* ⊆ {..*k*} **..**
then show *finite S*
by (*rule finite-subset*) *simp*
qed

lemma *infinite-nat-iff-unbounded*:
infinite (*S*::*nat set*) = (∀ *m*. ∃ *n*. *m*<*n* ∧ *n*∈*S*)
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
show *?rhs*
proof (*rule ccontr*)
assume ¬ *?rhs*
then obtain *m* **where** *m*: ∀ *n*. *m*<*n* ⟶ *n*∉*S* **by** *blast*
then have *S* ⊆ {..*m*}
by (*auto simp add*: *sym* [*OF linorder-not-less*])
with ‹*?lhs*› **show** *False*
by (*simp add*: *finite-nat-iff-bounded-le*)
qed
next
assume *?rhs*
show *?lhs*
proof
assume *finite S*
then obtain *m* **where** *S* ⊆ {..*m*}
by (*auto simp add*: *finite-nat-iff-bounded-le*)
then have ∀ *n*. *m*<*n* ⟶ *n*∉*S* **by** *auto*
with ‹*?rhs*› **show** *False* **by** *blast*
qed
qed

lemma *infinite-nat-iff-unbounded-le*:
infinite (*S*::*nat set*) = (∀ *m*. ∃ *n*. *m*≤*n* ∧ *n*∈*S*)
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
show *?rhs*

```
  proof
    fix m
    from ‹?lhs› obtain n where m<n ∧ n∈S
      by (auto simp add: infinite-nat-iff-unbounded)
    then have m≤n ∧ n∈S by simp
    then show ∃ n. m ≤ n ∧ n ∈ S ..
  qed
next
  assume ?rhs
  show ?lhs
  proof (auto simp add: infinite-nat-iff-unbounded)
    fix m
    from ‹?rhs› obtain n where Suc m ≤ n ∧ n∈S
      by blast
    then have m<n ∧ n∈S by simp
    then show ∃ n. m < n ∧ n ∈ S ..
  qed
qed
```

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

```
lemma unbounded-k-infinite:
  assumes k: ∀ m. k<m ⟶ (∃ n. m<n ∧ n∈S)
  shows infinite (S::nat set)
proof −
  {
    fix m have ∃ n. m<n ∧ n∈S
    proof (cases k<m)
      case True
      with k show ?thesis by blast
    next
      case False
      from k obtain n where Suc k < n ∧ n∈S by auto
      with False have m<n ∧ n∈S by auto
      then show ?thesis ..
    qed
  }
  then show ?thesis
    by (auto simp add: infinite-nat-iff-unbounded)
qed

lemma nat-infinite [simp]: infinite (UNIV :: nat set)
  by (auto simp add: infinite-nat-iff-unbounded)

lemma nat-not-finite [elim]: finite (UNIV::nat set) ⟹ R
  by simp
```

Every infinite set contains a countable subset. More precisely we show that

a set S is infinite if and only if there exists an injective function from the naturals into S.

lemma *range-inj-infinite*:
  inj (f::*nat* ⇒ '*a*) ⟹ *infinite* (*range f*)
proof
  assume *finite* (*range f*) **and** *inj f*
  then have *finite* (*UNIV*::*nat set*)
    by (*rule finite-imageD*)
  then show *False* **by** *simp*
qed

lemma *int-infinite* [*simp*]:
  shows *infinite* (*UNIV*::*int set*)
proof −
  from *inj-int* **have** *infinite* (*range int*) **by** (*rule range-inj-infinite*)
  moreover
  have *range int* ⊆ (*UNIV*::*int set*) **by** *simp*
  ultimately show *infinite* (*UNIV*::*int set*) **by** (*simp add*: *infinite-super*)
qed

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *linorder-injI*:
  assumes *hyp*: !!x y. x < (y::'a::*linorder*) ==> $f\ x \neq f\ y$
  shows *inj f*
proof (*rule inj-onI*)
  fix x y
  assume *f-eq*: $f\ x = f\ y$
  show $x = y$
  proof (*rule linorder-cases*)
    assume $x < y$
    with *hyp* **have** $f\ x \neq f\ y$ **by** *blast*
    with *f-eq* **show** *?thesis* **by** *simp*
  next
    assume $x = y$
    then show *?thesis* **.**
  next
    assume $y < x$
    with *hyp* **have** $f\ y \neq f\ x$ **by** *blast*
    with *f-eq* **show** *?thesis* **by** *simp*
  qed
qed

lemma *infinite-countable-subset*:
  assumes *inf*: *infinite* (S::'a *set*)
  shows ∃f. *inj* (f::*nat* ⇒ '*a*) ∧ *range f* ⊆ S

proof −
def *Sseq* ≡ *nat-rec S* (λ*n T*. *T* − {*SOME e*. *e* ∈ *T*})
def *pick* ≡ λ*n*. (*SOME e*. *e* ∈ *Sseq n*)
have *Sseq-inf*: ⋀*n*. *infinite* (*Sseq n*)
proof −
fix *n*
show *infinite* (*Sseq n*)
proof (*induct n*)
from *inf* **show** *infinite* (*Sseq 0*)
by (*simp add*: *Sseq-def*)
next
fix *n*
assume *infinite* (*Sseq n*) **then show** *infinite* (*Sseq* (*Suc n*))
by (*simp add*: *Sseq-def infinite-remove*)
qed
qed
have *Sseq-S*: ⋀*n*. *Sseq n* ⊆ *S*
proof −
fix *n*
show *Sseq n* ⊆ *S*
by (*induct n*) (*auto simp add*: *Sseq-def*)
qed
have *Sseq-pick*: ⋀*n*. *pick n* ∈ *Sseq n*
proof −
fix *n*
show *pick n* ∈ *Sseq n*
proof (*unfold pick-def*, *rule someI-ex*)
from *Sseq-inf* **have** *infinite* (*Sseq n*) **.**
then have *Sseq n* ≠ {} **by** *auto*
then show ∃*x*. *x* ∈ *Sseq n* **by** *auto*
qed
qed
with *Sseq-S* **have** *rng*: *range pick* ⊆ *S*
by *auto*
have *pick-Sseq-gt*: ⋀*n m*. *pick n* ∉ *Sseq* (*n* + *Suc m*)
proof −
fix *n m*
show *pick n* ∉ *Sseq* (*n* + *Suc m*)
by (*induct m*) (*auto simp add*: *Sseq-def pick-def*)
qed
have *pick-pick*: ⋀*n m*. *pick n* ≠ *pick* (*n* + *Suc m*)
proof −
fix *n m*
from *Sseq-pick* **have** *pick* (*n* + *Suc m*) ∈ *Sseq* (*n* + *Suc m*) **.**
moreover from *pick-Sseq-gt*
have *pick n* ∉ *Sseq* (*n* + *Suc m*) **.**
ultimately show *pick n* ≠ *pick* (*n* + *Suc m*)
by *auto*
qed

```
  have inj: inj pick
  proof (rule linorder-injI)
    fix i j :: nat
    assume i < j
    show pick i ≠ pick j
    proof
      assume eq: pick i = pick j
      from ‹i < j› obtain k where j = i + Suc k
        by (auto simp add: less-iff-Suc-add)
      with pick-pick have pick i ≠ pick j by simp
      with eq show False by simp
    qed
  qed
  from rng inj show ?thesis by auto
qed

lemma infinite-iff-countable-subset:
    infinite S = (∃f. inj (f::nat ⇒ 'a) ∧ range f ⊆ S)
  by (auto simp add: infinite-countable-subset range-inj-infinite infinite-super)
```

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

```
lemma inf-img-fin-dom:
  assumes img: finite (f`A) and dom: infinite A
  shows ∃y ∈ f`A. infinite (f -` {y})
proof (rule ccontr)
  assume ¬ ?thesis
  with img have finite (UN y:f`A. f -` {y}) by (blast intro: finite-UN-I)
  moreover have A ⊆ (UN y:f`A. f -` {y}) by auto
  moreover note dom
  ultimately show False by (simp add: infinite-super)
qed

lemma inf-img-fin-domE:
  assumes finite (f`A) and infinite A
  obtains y where y ∈ f`A and infinite (f -` {y})
  using assms by (blast dest: inf-img-fin-dom)
```

28.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

```
definition
  Inf-many :: ('a ⇒ bool) ⇒ bool  (binder INFM  10) where
  Inf-many P = infinite {x. P x}
```

definition
Alm-all :: (*'a* ⇒ *bool*) ⇒ *bool* (**binder** *MOST* *10*) **where**
Alm-all P = (¬ (*INFM x*. ¬ *P x*))

notation (*xsymbols*)
Inf-many (**binder** ∃$_\infty$ *10*) **and**
Alm-all (**binder** ∀$_\infty$ *10*)

notation (*HTML* **output**)
Inf-many (**binder** ∃$_\infty$ *10*) **and**
Alm-all (**binder** ∀$_\infty$ *10*)

lemma *INFM-EX*:
(∃$_\infty x$. *P x*) ⟹ (∃ *x*. *P x*)
unfolding *Inf-many-def*
proof (*rule ccontr*)
assume *inf*: *infinite* {*x*. *P x*}
assume ¬ *?thesis* **then have** {*x*. *P x*} = {} **by** *simp*
then have *finite* {*x*. *P x*} **by** *simp*
with *inf* **show** *False* **by** *simp*
qed

lemma *MOST-iff-finiteNeg*: (∀$_\infty x$. *P x*) = *finite* {*x*. ¬ *P x*}
by (*simp add*: *Alm-all-def Inf-many-def*)

lemma *ALL-MOST*: ∀ *x*. *P x* ⟹ ∀$_\infty x$. *P x*
by (*simp add*: *MOST-iff-finiteNeg*)

lemma *INFM-mono*:
assumes *inf*: ∃$_\infty x$. *P x* **and** *q*: ⋀*x*. *P x* ⟹ *Q x*
shows ∃$_\infty x$. *Q x*
proof −
from *inf* **have** *infinite* {*x*. *P x*} **unfolding** *Inf-many-def* **.**
moreover from *q* **have** {*x*. *P x*} ⊆ {*x*. *Q x*} **by** *auto*
ultimately show *?thesis*
by (*simp add*: *Inf-many-def infinite-super*)
qed

lemma *MOST-mono*: ∀$_\infty x$. *P x* ⟹ (⋀*x*. *P x* ⟹ *Q x*) ⟹ ∀$_\infty x$. *Q x*
unfolding *Alm-all-def* **by** (*blast intro*: *INFM-mono*)

lemma *INFM-disj-distrib*:
(∃$_\infty x$. *P x* ∨ *Q x*) ⟷ (∃$_\infty x$. *P x*) ∨ (∃$_\infty x$. *Q x*)
unfolding *Inf-many-def* **by** (*simp add*: *Collect-disj-eq*)

lemma *MOST-conj-distrib*:
(∀$_\infty x$. *P x* ∧ *Q x*) ⟷ (∀$_\infty x$. *P x*) ∧ (∀$_\infty x$. *Q x*)
unfolding *Alm-all-def* **by** (*simp add*: *INFM-disj-distrib del*: *disj-not1*)

lemma *MOST-rev-mp*:
assumes $\forall_\infty x.\ P\ x$ **and** $\forall_\infty x.\ P\ x \longrightarrow Q\ x$
shows $\forall_\infty x.\ Q\ x$
proof −
have $\forall_\infty x.\ P\ x \wedge (P\ x \longrightarrow Q\ x)$
using *prems* **by** (*simp add*: *MOST-conj-distrib*)
thus *?thesis* **by** (*rule MOST-mono*) *simp*
qed

lemma *not-INFM* [*simp*]: $\neg$ (*INFM x*. *P x*) $\longleftrightarrow$ (*MOST x*. $\neg$ *P x*)
unfolding *Alm-all-def not-not* **..**

lemma *not-MOST* [*simp*]: $\neg$ (*MOST x*. *P x*) $\longleftrightarrow$ (*INFM x*. $\neg$ *P x*)
unfolding *Alm-all-def not-not* **..**

lemma *INFM-const* [*simp*]: (*INFM x*::'*a*. *P*) $\longleftrightarrow$ *P* $\wedge$ *infinite* (*UNIV*::'*a set*)
unfolding *Inf-many-def* **by** *simp*

lemma *MOST-const* [*simp*]: (*MOST x*::'*a*. *P*) $\longleftrightarrow$ *P* $\vee$ *finite* (*UNIV*::'*a set*)
unfolding *Alm-all-def* **by** *simp*

lemma *INFM-nat*: ($\exists_\infty n.\ P\ (n::nat)$) = ($\forall\, m.\ \exists\, n.\ m<n \wedge P\ n$)
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: ($\exists_\infty n.\ P\ (n::nat)$) = ($\forall\, m.\ \exists\, n.\ m\leq n \wedge P\ n$)
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded-le*)

lemma *MOST-nat*: ($\forall_\infty n.\ P\ (n::nat)$) = ($\exists\, m.\ \forall\, n.\ m<n \longrightarrow P\ n$)
by (*simp add*: *Alm-all-def INFM-nat*)

lemma *MOST-nat-le*: ($\forall_\infty n.\ P\ (n::nat)$) = ($\exists\, m.\ \forall\, n.\ m\leq n \longrightarrow P\ n$)
by (*simp add*: *Alm-all-def INFM-nat-le*)

28.3 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

consts
enumerate :: '*a*::*wellorder set* => (*nat* => '*a*::*wellorder*)
primrec
enumerate-0: *enumerate S 0* = (*LEAST n*. *n* $\in$ *S*)
enumerate-Suc: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*LEAST n*. *n* $\in$ *S*}) *n*

lemma *enumerate-Suc'*:
enumerate S (*Suc n*) = *enumerate* (*S* − {*enumerate S 0*}) *n*
by *simp*

lemma *enumerate-in-set*: *infinite S* $\Longrightarrow$ *enumerate S n* : *S*
apply (*induct n arbitrary*: *S*)

apply (*fastsimp intro*: *LeastI dest*!: *infinite-imp-nonempty*)
apply *simp*
apply (*metis Collect-def Collect-mem-eq DiffE infinite-remove*)
done

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite S* $\Longrightarrow$ *enumerate S n* $<$ *enumerate S* (*Suc n*)
apply (*induct n arbitrary*: *S*)
apply (*rule order-le-neq-trans*)
apply (*simp add*: *enumerate-0 Least-le enumerate-in-set*)
apply (*simp only*: *enumerate-Suc'*)
apply (*subgoal-tac enumerate* (*S* $-$ {*enumerate S 0*}) *0* $:$ *S* $-$ {*enumerate S 0*})
apply (*blast intro*: *sym*)
apply (*simp add*: *enumerate-in-set del*: *Diff-iff*)
apply (*simp add*: *enumerate-Suc'*)
done

lemma *enumerate-mono*: *m*$<$*n* $\Longrightarrow$ *infinite S* $\Longrightarrow$ *enumerate S m* $<$ *enumerate S n*
apply (*erule less-Suc-induct*)
apply (*auto intro*: *enumerate-step*)
done

28.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one :: *'a set* $\Rightarrow$ *bool* **where**
atmost-one S = ($\forall$ *x y*. *x*$\in$*S* $\wedge$ *y*$\in$*S* $\longrightarrow$ *x*=*y*)

lemma *atmost-one-empty*: *S* = {} $\Longrightarrow$ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-singleton*: *S* = {*x*} $\Longrightarrow$ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-unique* [*elim*]: *atmost-one S* $\Longrightarrow$ *x* $\in$ *S* $\Longrightarrow$ *y* $\in$ *S* $\Longrightarrow$ *y* = *x*
by (*simp add*: *atmost-one-def*)

end

theory *Eventual*
imports *Infinite-Set*
begin

28.5 Lemmas about MOST

lemma *MOST-INFM*:
assumes *inf*: *infinite* (*UNIV*::'*a set*)
shows *MOST x*::'*a*. *P x* ⟹ *INFM x*::'*a*. *P x*
unfolding *Alm-all-def Inf-many-def*
apply (*auto simp add*: *Collect-neg-eq*)
apply (*drule* (*1*) *finite-UnI*)
apply (*simp add*: *Compl-partition2 inf*)
done

lemma *MOST-comp*: ⟦*inj f*; *MOST x*. *P x*⟧ ⟹ *MOST x*. *P* (*f x*)
unfolding *MOST-iff-finiteNeg*
by (*drule* (*1*) *finite-vimageI*, *simp*)

lemma *INFM-comp*: ⟦*inj f*; *INFM x*. *P* (*f x*)⟧ ⟹ *INFM x*. *P x*
unfolding *Inf-many-def*
by (*clarify*, *drule* (*1*) *finite-vimageI*, *simp*)

lemma *MOST-SucI*: *MOST n*. *P n* ⟹ *MOST n*. *P* (*Suc n*)
by (*rule MOST-comp* [*OF inj-Suc*])

lemma *MOST-SucD*: *MOST n*. *P* (*Suc n*) ⟹ *MOST n*. *P n*
unfolding *MOST-nat*
apply (*clarify*, *rule-tac x*=*Suc m* **in** *exI*, *clarify*)
apply (*erule Suc-lessE*, *simp*)
done

lemma *MOST-Suc-iff*: (*MOST n*. *P* (*Suc n*)) ⟷ (*MOST n*. *P n*)
by (*rule iffI* [*OF MOST-SucD MOST-SucI*])

lemma *INFM-finite-Bex-distrib*:
finite A ⟹ (*INFM y*. ∃ *x*∈*A*. *P x y*) ⟷ (∃ *x*∈*A*. *INFM y*. *P x y*)
by (*induct set*: *finite*, *simp*, *simp add*: *INFM-disj-distrib*)

lemma *MOST-finite-Ball-distrib*:
finite A ⟹ (*MOST y*. ∀ *x*∈*A*. *P x y*) ⟷ (∀ *x*∈*A*. *MOST y*. *P x y*)
by (*induct set*: *finite*, *simp*, *simp add*: *MOST-conj-distrib*)

lemma *MOST-ge-nat*: *MOST n*::*nat*. *m* ≤ *n*
unfolding *MOST-nat-le* **by** *fast*

28.6 Eventually constant sequences

definition
eventually-constant :: (*nat* ⇒ '*a*) ⇒ *bool*
where
eventually-constant S = (∃ *x*. *MOST i*. *S i* = *x*)

lemma *eventually-constant-MOST-MOST*:

eventually-constant S ⟷ (*MOST m. MOST n. S n = S m*)
unfolding *eventually-constant-def MOST-nat*
apply *safe*
apply (*rule-tac x=m* **in** *exI*, *clarify*)
apply (*rule-tac x=m* **in** *exI*, *clarify*)
apply *simp*
apply *fast*
done

lemma *eventually-constantI*: *MOST i. S i = x* ⟹ *eventually-constant S*
unfolding *eventually-constant-def* **by** *fast*

lemma *eventually-constant-comp*:
eventually-constant (λ*i. S i*) ⟹ *eventually-constant* (λ*i. f* (*S i*))
unfolding *eventually-constant-def*
apply (*erule exE*, *rule-tac x=f x* **in** *exI*)
apply (*erule MOST-mono*, *simp*)
done

lemma *eventually-constant-Suc-iff*:
eventually-constant (λ*i. S* (*Suc i*)) ⟷ *eventually-constant* (λ*i. S i*)
unfolding *eventually-constant-def*
by (*subst MOST-Suc-iff*, *rule refl*)

lemma *eventually-constant-SucD*:
eventually-constant (λ*i. S* (*Suc i*)) ⟹ *eventually-constant* (λ*i. S i*)
by (*rule eventually-constant-Suc-iff* [*THEN iffD1*])

28.7 Limits of eventually constant sequences

definition
eventual :: (*nat* ⇒ *'a*) ⇒ *'a* **where**
eventual S = (*THE x. MOST i. S i = x*)

lemma *eventual-eqI*: *MOST i. S i = x* ⟹ *eventual S = x*
unfolding *eventual-def*
apply (*rule the-equality*, *assumption*)
apply (*rename-tac y*)
apply (*subgoal-tac MOST i*::*nat. y = x*, *simp*)
apply (*erule MOST-rev-mp*)
apply (*erule MOST-rev-mp*)
apply *simp*
done

lemma *MOST-eq-eventual*:
eventually-constant S ⟹ *MOST i. S i = eventual S*
unfolding *eventually-constant-def*
by (*erule exE*, *simp add*: *eventual-eqI*)

lemma *eventual-mem-range*:
eventually-constant S $\Longrightarrow$ *eventual S* $\in$ *range S*
apply (*drule MOST-eq-eventual*)
apply (*simp only*: *MOST-nat-le*, *clarify*)
apply (*drule spec*, *drule mp*, *rule order-refl*)
apply (*erule range-eqI* [*OF sym*])
done

lemma *eventually-constant-MOST-iff*:
assumes *S*: *eventually-constant S*
shows (*MOST n. P* (*S n*)) $\longleftrightarrow$ *P* (*eventual S*)
apply (*subgoal-tac* (*MOST n. P* (*S n*)) $\longleftrightarrow$ (*MOST n*::*nat. P* (*eventual S*)))
apply *simp*
apply (*rule iffI*)
apply (*rule MOST-rev-mp* [*OF MOST-eq-eventual* [*OF S*]])
apply (*erule MOST-mono*, *force*)
apply (*rule MOST-rev-mp* [*OF MOST-eq-eventual* [*OF S*]])
apply (*erule MOST-mono*, *simp*)
done

lemma *MOST-eventual*:
⟦*eventually-constant S*; *MOST n. P* (*S n*)⟧ $\Longrightarrow$ *P* (*eventual S*)
proof −
assume *eventually-constant S*
hence *MOST n. S n* = *eventual S*
by (*rule MOST-eq-eventual*)
moreover assume *MOST n. P* (*S n*)
ultimately have *MOST n. S n* = *eventual S* ∧ *P* (*S n*)
by (*rule MOST-conj-distrib* [*THEN iffD2*, *OF conjI*])
hence *MOST n*::*nat. P* (*eventual S*)
by (*rule MOST-mono*) *auto*
thus *?thesis* **by** *simp*
qed

lemma *eventually-constant-MOST-Suc-eq*:
eventually-constant S $\Longrightarrow$ *MOST n. S* (*Suc n*) = *S n*
apply (*drule MOST-eq-eventual*)
apply (*frule MOST-Suc-iff* [*THEN iffD2*])
apply (*erule MOST-rev-mp*)
apply (*erule MOST-rev-mp*)
apply *simp*
done

lemma *eventual-comp*:
eventually-constant S $\Longrightarrow$ *eventual* (λ*i. f* (*S i*)) = *f* (*eventual* (λ*i. S i*))
apply (*rule eventual-eqI*)
apply (*rule MOST-mono*)
apply (*erule MOST-eq-eventual*)
apply *simp*

done

end

29 Algebraic: Algebraic deflations

theory *Algebraic*
imports *Completion Fix Eventual*
begin

29.1 Constructing finite deflations by iteration

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

lemma *le-Suc-induct*:
  assumes *le*: $i \leq j$
  assumes *step*: $\bigwedge i.\ P\ i\ (Suc\ i)$
  assumes *refl*: $\bigwedge i.\ P\ i\ i$
  assumes *trans*: $\bigwedge i\ j\ k.\ [\![P\ i\ j;\ P\ j\ k]\!] \Longrightarrow P\ i\ k$
  shows *P i j*
proof (*cases* $i = j$)
  assume $i = j$
  thus *P i j* **by** (*simp add*: *refl*)
next
  assume $i \neq j$
  with *le* **have** $i < j$ **by** *simp*
  thus *P i j* **using** *step trans* **by** (*rule less-Suc-induct*)
qed

A pre-deflation is like a deflation, but not idempotent.

locale *pre-deflation* =
  fixes $f :: 'a \to 'a$::*cpo*
  assumes *less*: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
  assumes *finite-range*: *finite* (*range* ($\lambda x.\ f{\cdot}x$))
begin

lemma *iterate-less*: *iterate i*·*f*·*x* $\sqsubseteq$ *x*
by (*induct i*, *simp-all add*: *trans-less* [*OF less*])

lemma *iterate-fixed*: $f{\cdot}x = x \Longrightarrow$ *iterate i*·*f*·*x* = *x*
by (*induct i*, *simp-all*)

lemma *antichain-iterate-app*: $i \leq j \Longrightarrow$ *iterate j*·*f*·*x* $\sqsubseteq$ *iterate i*·*f*·*x*
apply (*erule le-Suc-induct*)
apply (*simp add*: *less*)
apply (*rule refl-less*)

apply (*erule* (*1*) *trans-less*)
done

lemma *finite-range-iterate-app*: *finite* (*range* (λ*i*. *iterate i*·*f*·*x*))
proof (*rule finite-subset*)
  show *range* (λ*i*. *iterate i*·*f*·*x*) ⊆ *insert x* (*range* (λ*x*. *f*·*x*))
    by (*clarify*, *case-tac i*, *simp-all*)
  show *finite* (*insert x* (*range* (λ*x*. *f*·*x*)))
    by (*simp add*: *finite-range*)
qed

lemma *eventually-constant-iterate-app*:
  eventually-constant (λ*i*. *iterate i*·*f*·*x*)
unfolding *eventually-constant-def MOST-nat-le*
proof −
  let *?Y* = λ*i*. *iterate i*·*f*·*x*
  have ∃*j*. ∀*k*. *?Y j* ⊑ *?Y k*
    apply (*rule finite-range-has-max*)
    apply (*erule antichain-iterate-app*)
    apply (*rule finite-range-iterate-app*)
    done
  then obtain *j* **where** *j*: ⋀*k*. *?Y j* ⊑ *?Y k* **by** *fast*
  show ∃ *z m*. ∀ *n*≥*m*. *?Y n* = *z*
  proof (*intro exI allI impI*)
    fix *k*
    assume *j* ≤ *k*
    hence *?Y k* ⊑ *?Y j* **by** (*rule antichain-iterate-app*)
    also have *?Y j* ⊑ *?Y k* **by** (*rule j*)
    finally show *?Y k* = *?Y j* **.**
  qed
qed

lemma *eventually-constant-iterate*:
  eventually-constant (λ*n*. *iterate n*·*f*)
proof −
  have ∀ *y*∈*range* (λ*x*. *f*·*x*). *eventually-constant* (λ*i*. *iterate i*·*f*·*y*)
    by (*simp add*: *eventually-constant-iterate-app*)
  hence ∀ *y*∈*range* (λ*x*. *f*·*x*). *MOST i*. *MOST j*. *iterate j*·*f*·*y* = *iterate i*·*f*·*y*
    unfolding *eventually-constant-MOST-MOST* **.**
  hence *MOST i*. *MOST j*. ∀ *y*∈*range* (λ*x*. *f*·*x*). *iterate j*·*f*·*y* = *iterate i*·*f*·*y*
    by (*simp only*: *MOST-finite-Ball-distrib* [*OF finite-range*])
  hence *MOST i*. *MOST j*. ∀ *x*. *iterate j*·*f*·(*f*·*x*) = *iterate i*·*f*·(*f*·*x*)
    by *simp*
  hence *MOST i*. *MOST j*. ∀ *x*. *iterate* (*Suc j*)·*f*·*x* = *iterate* (*Suc i*)·*f*·*x*
    by (*simp only*: *iterate-Suc2*)
  hence *MOST i*. *MOST j*. *iterate* (*Suc j*)·*f* = *iterate* (*Suc i*)·*f*
    by (*simp only*: *expand-cfun-eq*)
  hence *eventually-constant* (λ*i*. *iterate* (*Suc i*)·*f*)
    unfolding *eventually-constant-MOST-MOST* **.**

thus *eventually-constant* ($\lambda i.$ *iterate* $i{\cdot}f$)
by (*rule eventually-constant-SucD*)
qed

abbreviation
$d :: 'a \to 'a$
where
$d \equiv$ *eventual* ($\lambda n.$ *iterate* $n{\cdot}f$)

lemma *MOST-d*: *MOST* $n.$ P (*iterate* $n{\cdot}f$) $\Longrightarrow$ P d
using *eventually-constant-iterate* **by** (*rule MOST-eventual*)

lemma *f-d*: $f{\cdot}(d{\cdot}x) = d{\cdot}x$
apply (*rule MOST-d*)
apply (*subst iterate-Suc* [*symmetric*])
apply (*rule eventually-constant-MOST-Suc-eq*)
apply (*rule eventually-constant-iterate-app*)
done

lemma *d-fixed-iff*: $d{\cdot}x = x \longleftrightarrow f{\cdot}x = x$
proof
assume $d{\cdot}x = x$
with *f-d* [**where** $x{=}x$]
show $f{\cdot}x = x$ **by** *simp*
next
assume f: $f{\cdot}x = x$
have $\forall\, n.$ *iterate* $n{\cdot}f{\cdot}x = x$
by (*rule allI*, *rule nat.induct*, *simp*, *simp add*: f)
hence *MOST* $n.$ *iterate* $n{\cdot}f{\cdot}x = x$
by (*rule ALL-MOST*)
thus $d{\cdot}x = x$
by (*rule MOST-d*)
qed

lemma *finite-deflation-d*: *finite-deflation* d
proof
fix $x :: 'a$
have $d \in$ *range* ($\lambda n.$ *iterate* $n{\cdot}f$)
using *eventually-constant-iterate*
by (*rule eventual-mem-range*)
then obtain n **where** n: $d =$ *iterate* $n{\cdot}f$ **..**
have *iterate* $n{\cdot}f{\cdot}(d{\cdot}x) = d{\cdot}x$
using *f-d* **by** (*rule iterate-fixed*)
thus $d{\cdot}(d{\cdot}x) = d{\cdot}x$
by (*simp add*: n)
next
fix $x :: 'a$
show $d{\cdot}x \sqsubseteq x$
by (*rule MOST-d*, *simp add*: *iterate-less*)

next
from *finite-range*
have *finite* $\{x.\ f{\cdot}x = x\}$
by (*rule finite-range-imp-finite-fixes*)
thus *finite* $\{x.\ d{\cdot}x = x\}$
by (*simp add*: *d-fixed-iff*)
qed

end

lemma *pre-deflation-d-f*:
assumes *finite-deflation d*
assumes f: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
shows *pre-deflation* $(d\ oo\ f)$
proof
interpret d: *finite-deflation d* **by** *fact*
fix x
show $\bigwedge x.\ (d\ oo\ f){\cdot}x \sqsubseteq x$
by (*simp*, *rule trans-less* [*OF d.less f*])
show *finite* (*range* $(\lambda x.\ (d\ oo\ f){\cdot}x)$)
by (*rule finite-subset* [*OF - d.finite-range*], *auto*)
qed

lemma *eventual-iterate-oo-fixed-iff*:
assumes *finite-deflation d*
assumes f: $\bigwedge x.\ f{\cdot}x \sqsubseteq x$
shows *eventual* $(\lambda n.\ iterate\ n{\cdot}(d\ oo\ f)){\cdot}x = x \longleftrightarrow d{\cdot}x = x \wedge f{\cdot}x = x$
proof −
interpret d: *finite-deflation d* **by** *fact*
let *?e* = $d\ oo\ f$
interpret e: *pre-deflation d oo f*
using ‹*finite-deflation d*› f
by (*rule pre-deflation-d-f*)
let *?g* = *eventual* $(\lambda n.\ iterate\ n{\cdot}?e)$
show *?thesis*
apply (*subst e.d-fixed-iff*)
apply *simp*
apply *safe*
apply (*erule subst*)
apply (*rule d.idem*)
apply (*rule antisym-less*)
apply (*rule f*)
apply (*erule subst*, *rule d.less*)
apply *simp*
done
qed

29.2 Type constructor for finite deflations

defaultsort *profinite*

typedef (**open**) *'a fin-defl* = $\{d::'a \to 'a.$ *finite-deflation* $d\}$
by (*fast intro*: *finite-deflation-approx*)

instantiation *fin-defl* :: (*profinite*) *sq-ord*
begin

definition
sq-le-fin-defl-def:
$op \sqsubseteq \equiv \lambda x\ y.$ *Rep-fin-defl* $x \sqsubseteq$ *Rep-fin-defl* y

instance ..
end

instance *fin-defl* :: (*profinite*) *po*
by (*rule typedef-po* [*OF type-definition-fin-defl sq-le-fin-defl-def*])

lemma *finite-deflation-Rep-fin-defl*: *finite-deflation* (*Rep-fin-defl d*)
using *Rep-fin-defl* **by** *simp*

interpretation *Rep-fin-defl*: *finite-deflation Rep-fin-defl d*
by (*rule finite-deflation-Rep-fin-defl*)

lemma *fin-defl-lessI*:
$(\bigwedge x.$ *Rep-fin-defl* $a{\cdot}x = x \Longrightarrow$ *Rep-fin-defl* $b{\cdot}x = x) \Longrightarrow a \sqsubseteq b$
unfolding *sq-le-fin-defl-def*
by (*rule Rep-fin-defl.lessI*)

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

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

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

29.3 Take function for finite deflations

definition
fd-take :: *nat* ⇒ *'a fin-defl* ⇒ *'a fin-defl*
where
fd-take i d = *Abs-fin-defl* (*eventual* (λ*n*. *iterate n*·(*approx i oo Rep-fin-defl d*)))

lemma *Rep-fin-defl-fd-take*:
Rep-fin-defl (*fd-take i d*) =
eventual (λ*n*. *iterate n*·(*approx i oo Rep-fin-defl d*))
unfolding *fd-take-def*
apply (*rule Abs-fin-defl-inverse* [*unfolded mem-Collect-eq*])
apply (*rule pre-deflation.finite-deflation-d*)
apply (*rule pre-deflation-d-f*)
apply (*rule finite-deflation-approx*)
apply (*rule Rep-fin-defl.less*)
done

lemma *fd-take-fixed-iff*:
Rep-fin-defl (*fd-take i d*)·*x* = *x* ⟷
approx i·*x* = *x* ∧ *Rep-fin-defl d*·*x* = *x*
unfolding *Rep-fin-defl-fd-take*
by (*rule eventual-iterate-oo-fixed-iff*
[*OF finite-deflation-approx Rep-fin-defl.less*])

lemma *fd-take-less*: *fd-take n d* ⊑ *d*
apply (*rule fin-defl-lessI*)
apply (*simp add*: *fd-take-fixed-iff*)
done

lemma *fd-take-idem*: *fd-take n* (*fd-take n d*) = *fd-take n d*
apply (*rule fin-defl-eqI*)
apply (*simp add*: *fd-take-fixed-iff*)
done

lemma *fd-take-mono*: *a* ⊑ *b* ⟹ *fd-take n a* ⊑ *fd-take n b*
apply (*rule fin-defl-lessI*)
apply (*simp add*: *fd-take-fixed-iff*)
apply (*simp add*: *fin-defl-lessD*)
done

lemma *approx-fixed-le-lemma*: ⟦*i* ≤ *j*; *approx i*·*x* = *x*⟧ ⟹ *approx j*·*x* = *x*
by (*erule subst*, *simp add*: *min-def*)

lemma *fd-take-chain*: *m* ≤ *n* ⟹ *fd-take m a* ⊑ *fd-take n a*
apply (*rule fin-defl-lessI*)
apply (*simp add*: *fd-take-fixed-iff*)
apply (*simp add*: *approx-fixed-le-lemma*)
done

lemma *finite-range-fd-take*: *finite* (*range* (*fd-take n*))
apply (*rule finite-imageD* [**where** *f*=λ*a*. {*x*. *Rep-fin-defl a*·*x* = *x*}])
apply (*rule finite-subset* [**where** *B*=*Pow* {*x*. *approx n*·*x* = *x*}])
apply (*clarify*, *simp add*: *fd-take-fixed-iff*)
apply (*simp add*: *finite-fixes-approx*)
apply (*rule inj-onI*, *clarify*)
apply (*simp add*: *expand-set-eq fin-defl-eqI*)
done

lemma *fd-take-covers*: ∃ *n*. *fd-take n a* = *a*
apply (*rule-tac x*=
  Max ((λ*x*. *LEAST n*. *approx n*·*x* = *x*) ' {*x*. *Rep-fin-defl a*·*x* = *x*}) **in** *exI*)
apply (*rule antisym-less*)
apply (*rule fd-take-less*)
apply (*rule fin-defl-lessI*)
apply (*simp add*: *fd-take-fixed-iff*)
apply (*rule approx-fixed-le-lemma*)
apply (*rule Max-ge*)
apply (*rule finite-imageI*)
apply (*rule Rep-fin-defl.finite-fixes*)
apply (*rule imageI*)
apply (*erule CollectI*)
apply (*rule LeastI-ex*)
apply (*rule profinite-compact-eq-approx*)
apply (*erule subst*)
apply (*rule Rep-fin-defl.compact*)
done

interpretation *fin-defl*: *basis-take sq-le fd-take*
apply *default*
apply (*rule fd-take-less*)
apply (*rule fd-take-idem*)
apply (*erule fd-take-mono*)
apply (*rule fd-take-chain*, *simp*)
apply (*rule finite-range-fd-take*)
apply (*rule fd-take-covers*)
done

29.4 Defining algebraic deflations by ideal completion

typedef (**open**) *'a alg-defl* =
  {*S*::*'a fin-defl set*. *sq-le.ideal S*}
by (*fast intro*: *sq-le.ideal-principal*)

instantiation *alg-defl* :: (*profinite*) *sq-ord*
begin

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

instance ..
end

instance *alg-defl* :: (*profinite*) *po*
by (*rule sq-le.typedef-ideal-po*
[*OF type-definition-alg-defl sq-le-alg-defl-def*])

instance *alg-defl* :: (*profinite*) *cpo*
by (*rule sq-le.typedef-ideal-cpo*
[*OF type-definition-alg-defl sq-le-alg-defl-def*])

lemma *Rep-alg-defl-lub*:
chain Y $\Longrightarrow$ *Rep-alg-defl* ($\bigsqcup i.\ Y\ i$) = ($\bigcup i.$ *Rep-alg-defl* (*Y i*))
by (*rule sq-le.typedef-ideal-rep-contlub*
[*OF type-definition-alg-defl sq-le-alg-defl-def*])

lemma *ideal-Rep-alg-defl*: *sq-le.ideal* (*Rep-alg-defl xs*)
by (*rule Rep-alg-defl* [*unfolded mem-Collect-eq*])

definition
alg-defl-principal :: *'a fin-defl* $\Rightarrow$ *'a alg-defl* **where**
alg-defl-principal t = *Abs-alg-defl* $\{u.\ u \sqsubseteq t\}$

lemma *Rep-alg-defl-principal*:
Rep-alg-defl (*alg-defl-principal t*) = $\{u.\ u \sqsubseteq t\}$
unfolding *alg-defl-principal-def*
by (*simp add*: *Abs-alg-defl-inverse sq-le.ideal-principal*)

interpretation *alg-defl*:
ideal-completion sq-le fd-take alg-defl-principal Rep-alg-defl
apply *default*
apply (*rule ideal-Rep-alg-defl*)
apply (*erule Rep-alg-defl-lub*)
apply (*rule Rep-alg-defl-principal*)
apply (*simp only*: *sq-le-alg-defl-def*)
done

Algebraic deflations are pointed

lemma *finite-deflation-UU*: *finite-deflation* $\bot$
by *default simp-all*

lemma *alg-defl-minimal*:
alg-defl-principal (*Abs-fin-defl* $\bot$) $\sqsubseteq x$
apply (*induct x rule*: *alg-defl.principal-induct*, *simp*)
apply (*rule alg-defl.principal-mono*)
apply (*induct-tac a*)
apply (*rule Abs-fin-defl-mono*)
apply (*rule finite-deflation-UU*)

apply *simp*
apply (*rule minimal*)
done

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

lemma *inst-alg-defl-pcpo*: $\bot$ = *alg-defl-principal* (*Abs-fin-defl* $\bot$)
by (*rule alg-defl-minimal* [*THEN UU-I*, *symmetric*])

Algebraic deflations are profinite

instantiation *alg-defl* :: (*profinite*) *profinite*
begin

definition
approx-alg-defl-def: *approx* = *alg-defl.completion-approx*

instance
apply (*intro-classes*, *unfold approx-alg-defl-def*)
apply (*rule alg-defl.chain-completion-approx*)
apply (*rule alg-defl.lub-completion-approx*)
apply (*rule alg-defl.completion-approx-idem*)
apply (*rule alg-defl.finite-fixes-completion-approx*)
done

end

instance *alg-defl* :: (*bifinite*) *bifinite* **..**

lemma *approx-alg-defl-principal* [*simp*]:
approx $n{\cdot}$(*alg-defl-principal* t) = *alg-defl-principal* (*fd-take* n t)
unfolding *approx-alg-defl-def*
by (*rule alg-defl.completion-approx-principal*)

lemma *approx-eq-alg-defl-principal*:
$\exists\, t{\in}$*Rep-alg-defl* xs. *approx* $n{\cdot}xs$ = *alg-defl-principal* (*fd-take* n t)
unfolding *approx-alg-defl-def*
by (*rule alg-defl.completion-approx-eq-principal*)

29.5 Applying algebraic deflations

definition
cast :: $'a$ *alg-defl* $\rightarrow$ $'a$ $\rightarrow$ $'a$
where
cast = *alg-defl.basis-fun Rep-fin-defl*

lemma *cast-alg-defl-principal*:
cast$\cdot$(*alg-defl-principal* a) = *Rep-fin-defl* a
unfolding *cast-def*

apply (*rule alg-defl.basis-fun-principal*)
apply (*simp only*: *sq-le-fin-defl-def*)
done

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

lemma *finite-deflation-cast*:
compact d ⟹ *finite-deflation* (*cast*·*d*)
apply (*drule alg-defl.compact-imp-principal*, *clarify*)
apply (*simp add*: *cast-alg-defl-principal*)
apply (*rule finite-deflation-Rep-fin-defl*)
done

interpretation *cast*: *deflation cast*·*d*
by (*rule deflation-cast*)

lemma *cast*·(⨆ *i*. *alg-defl-principal* (*Abs-fin-defl* (*approx i*)))·*x* = *x*
apply (*subst contlub-cfun-arg*)
apply (*rule chainI*)
apply (*rule alg-defl.principal-mono*)
apply (*rule Abs-fin-defl-mono*)
apply (*rule finite-deflation-approx*)
apply (*rule finite-deflation-approx*)
apply (*rule chainE*)
apply (*rule chain-approx*)
apply (*simp add*: *cast-alg-defl-principal Abs-fin-defl-inverse finite-deflation-approx*)
done

This lemma says that if we have an ep-pair from a bifinite domain into a universal domain, then e oo p is an algebraic deflation.

lemma
assumes *ep-pair e p*
constrains *e* :: *'a*::*profinite* → *'b*::*profinite*
shows ∃ *d*. *cast*·*d* = *e oo p*
proof
interpret *ep-pair e p* **by** *fact*
let *?a* = λ*i*. *e oo approx i oo p*
have *a*: ⋀*i*. *finite-deflation* (*?a i*)
apply (*rule finite-deflation-e-d-p*)
apply (*rule finite-deflation-approx*)
done
let *?d* = ⨆ *i*. *alg-defl-principal* (*Abs-fin-defl* (*?a i*))
show *cast*·*?d* = *e oo p*

```
    apply (subst contlub-cfun-arg)
    apply (rule chainI)
    apply (rule alg-defl.principal-mono)
    apply (rule Abs-fin-defl-mono [OF a a])
    apply (rule chainE, simp)
    apply (subst cast-alg-defl-principal)
    apply (simp add: Abs-fin-defl-inverse a)
    apply (simp add: expand-cfun-eq lub-distribs)
    done
qed
```

This lemma says that if we have an ep-pair from a cpo into a bifinite domain, and e oo p is an algebraic deflation, then the cpo is bifinite.

```
lemma
  assumes ep-pair e p
  constrains e :: 'a::cpo → 'b::profinite
  assumes d: ⋀x. cast·d·x = e·(p·x)
  obtains a :: nat ⇒ 'a → 'a where
    ⋀i. finite-deflation (a i)
    (⨆ i. a i) = ID
proof
  interpret ep-pair e p by fact
  let ?a = λi. p oo cast·(approx i·d) oo e
  show ⋀i. finite-deflation (?a i)
    apply (rule finite-deflation-p-d-e)
    apply (rule finite-deflation-cast)
    apply (rule compact-approx)
    apply (rule sq-ord-less-eq-trans [OF - d])
    apply (rule monofun-cfun-fun)
    apply (rule monofun-cfun-arg)
    apply (rule approx-less)
    done
  show (⨆ i. ?a i) = ID
    apply (rule ext-cfun, simp)
    apply (simp add: lub-distribs)
    apply (simp add: d)
    done
qed

end
```

30 NatIso: Isomorphisms of the Natural Numbers

```
theory NatIso
imports Parity
begin
```

30.1 Isomorphism between naturals and sums of naturals

primrec
  sum2nat :: *nat* + *nat* ⇒ *nat*
where
  sum2nat (*Inl a*) = *a* + *a*
| *sum2nat* (*Inr b*) = *Suc* (*b* + *b*)

primrec
  nat2sum :: *nat* ⇒ *nat* + *nat*
where
  nat2sum 0 = *Inl 0*
| *nat2sum* (*Suc n*) = (*case nat2sum n of*
    Inl a ⇒ *Inr a* | *Inr b* ⇒ *Inl* (*Suc b*))

lemma *nat2sum-sum2nat* [*simp*]: *nat2sum* (*sum2nat x*) = *x*
by (*induct x*, *rule-tac* [!] *nat.induct*, *simp-all*)

lemma *sum2nat-nat2sum* [*simp*]: *sum2nat* (*nat2sum n*) = *n*
by (*induct n*, *auto split*: *sum.split*)

lemma *inj-sum2nat*: *inj sum2nat*
by (*rule inj-on-inverseI*, *rule nat2sum-sum2nat*)

lemma *sum2nat-eq-iff* [*simp*]: *sum2nat x* = *sum2nat y* ⟷ *x* = *y*
by (*rule inj-sum2nat* [*THEN inj-eq*])

lemma *inj-nat2sum*: *inj nat2sum*
by (*rule inj-on-inverseI*, *rule sum2nat-nat2sum*)

lemma *nat2sum-eq-iff* [*simp*]: *nat2sum x* = *nat2sum y* ⟷ *x* = *y*
by (*rule inj-nat2sum* [*THEN inj-eq*])

declare *sum2nat.simps* [*simp del*]
declare *nat2sum.simps* [*simp del*]

30.2 Isomorphism between naturals and pairs of naturals

function
  prod2nat :: *nat* × *nat* ⇒ *nat*
where
  prod2nat (*0*, *0*) = *0*
| *prod2nat* (*0*, *Suc n*) = *Suc* (*prod2nat* (*n*, *0*))
| *prod2nat* (*Suc m*, *n*) = *Suc* (*prod2nat* (*m*, *Suc n*))
by *pat-completeness auto*

termination by (*relation*
  inv-image (*measure id* <∗*lex*∗> *measure id*) (λ(*x*, *y*). (*x*+*y*, *x*))) *auto*

primrec

```
  nat2prod :: nat ⇒ nat × nat
where
  nat2prod 0 = (0, 0)
| nat2prod (Suc x) =
    (case nat2prod x of (m, n) ⇒
      (case n of 0 ⇒ (0, Suc m) | Suc n ⇒ (Suc m, n)))

lemma nat2prod-prod2nat [simp]: nat2prod (prod2nat x) = x
by (induct x, rule prod2nat.induct, simp-all)

lemma prod2nat-nat2prod [simp]: prod2nat (nat2prod n) = n
by (induct n, auto split: prod.split nat.split)

lemma inj-prod2nat: inj prod2nat
by (rule inj-on-inverseI, rule nat2prod-prod2nat)

lemma prod2nat-eq-iff [simp]: prod2nat x = prod2nat y ⟷ x = y
by (rule inj-prod2nat [THEN inj-eq])

lemma inj-nat2prod: inj nat2prod
by (rule inj-on-inverseI, rule prod2nat-nat2prod)

lemma nat2prod-eq-iff [simp]: nat2prod x = nat2prod y ⟷ x = y
by (rule inj-nat2prod [THEN inj-eq])
```

30.2.1 Ordering properties

```
lemma fst-snd-le-prod2nat: fst x ≤ prod2nat x ∧ snd x ≤ prod2nat x
by (induct x rule: prod2nat.induct) auto

lemma fst-le-prod2nat: fst x ≤ prod2nat x
by (rule fst-snd-le-prod2nat [THEN conjunct1])

lemma snd-le-prod2nat: snd x ≤ prod2nat x
by (rule fst-snd-le-prod2nat [THEN conjunct2])

lemma le-prod2nat-1: a ≤ prod2nat (a, b)
using fst-le-prod2nat [where x=(a, b)] by simp

lemma le-prod2nat-2: b ≤ prod2nat (a, b)
using snd-le-prod2nat [where x=(a, b)] by simp

declare prod2nat.simps [simp del]
declare nat2prod.simps [simp del]
```

30.3 Isomorphism between naturals and finite sets of naturals

30.3.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite* (*Suc* −‘ *F*) ⟷ *finite F*
apply (*safe intro*!: *finite-vimageI inj-Suc*)
apply (*rule finite-subset* [**where** *B*=*insert 0* (*Suc* ‘ *Suc* −‘ *F*)])
apply (*rule subsetI*, *case-tac x*, *simp*, *simp*)
apply (*rule finite-insert* [*THEN iffD2*])
apply (*erule finite-imageI*)
done

lemma *vimage-Suc-insert-0*: *Suc* −‘ *insert 0 A* = *Suc* −‘ *A*
by *auto*

lemma *vimage-Suc-insert-Suc*:
Suc −‘ *insert* (*Suc n*) *A* = *insert n* (*Suc* −‘ *A*)
by *auto*

lemma *even-nat-Suc-div-2*: *even x* ⟹ *Suc x div 2* = *x div 2*
by (*simp only*: *numeral-2-eq-2 even-nat-plus-one-div-two*)

lemma *div2-even-ext-nat*:
⟦*x div 2* = *y div 2*; *even x* = *even y*⟧ ⟹ (*x*::*nat*) = *y*
apply (*rule mod-div-equality* [*of x 2*, *THEN subst*])
apply (*rule mod-div-equality* [*of y 2*, *THEN subst*])
apply (*case-tac even x*)
apply (*simp add*: *numeral-2-eq-2 even-nat-equiv-def*)
apply (*simp add*: *numeral-2-eq-2 odd-nat-equiv-def*)
done

30.3.2 From sets to naturals

definition
set2nat :: *nat set* ⇒ *nat* **where**
set2nat = *setsum* (*op* ^ *2*)

lemma *set2nat-empty* [*simp*]: *set2nat* {} = *0*
by (*simp add*: *set2nat-def*)

lemma *set2nat-insert* [*simp*]:
⟦*finite A*; *n* ∉ *A*⟧ ⟹ *set2nat* (*insert n A*) = *2*^*n* + *set2nat A*
by (*simp add*: *set2nat-def*)

lemma *even-set2nat-iff*: *finite A* ⟹ *even* (*set2nat A*) = (*0* ∉ *A*)
by (*unfold set2nat-def*, *erule finite-induct*, *auto*)

lemma *set2nat-vimage-Suc*: *set2nat* (*Suc* −‘ *A*) = *set2nat A div 2*
apply (*case-tac finite A*)

apply (*erule finite-induct*, *simp*)
apply (*case-tac x*)
apply (*simp add*: *even-nat-Suc-div-2 even-set2nat-iff vimage-Suc-insert-0*)
apply (*simp add*: *finite-vimageI add-commute vimage-Suc-insert-Suc*)
apply (*simp add*: *set2nat-def finite-vimage-Suc-iff*)
done

lemmas *set2nat-div-2* = *set2nat-vimage-Suc* [*symmetric*]

30.3.3 From naturals to sets

definition
nat2set :: *nat* ⇒ *nat set* **where**
nat2set x = {*n*. *odd* (*x div 2* ^ *n*)}

lemma *nat2set-0* [*simp*]: *0* ∈ *nat2set x* ⟷ *odd x*
by (*simp add*: *nat2set-def*)

lemma *nat2set-Suc* [*simp*]:
Suc n ∈ *nat2set x* ⟷ *n* ∈ *nat2set* (*x div 2*)
by (*simp add*: *nat2set-def div-mult2-eq*)

lemma *nat2set-zero* [*simp*]: *nat2set 0* = {}
by (*simp add*: *nat2set-def*)

lemma *nat2set-div-2*: *nat2set* (*x div 2*) = *Suc* −‘ *nat2set x*
by *auto*

lemma *nat2set-plus-power-2*:
n ∉ *nat2set z* ⟹ *nat2set* (*2* ^ *n* + *z*) = *insert n* (*nat2set z*)
apply (*induct n arbitrary*: *z*, *simp-all*)
apply (*rule set-ext*, *induct-tac x*, *simp*, *simp add*: *even-nat-Suc-div-2*)
apply (*rule set-ext*, *induct-tac x*, *simp*, *simp add*: *add-commute*)
done

lemma *finite-nat2set* [*simp*]: *finite* (*nat2set n*)
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac n* = *0*, *simp*)
apply (*drule-tac x*=*n div 2* **in** *spec*, *simp*)
apply (*simp add*: *nat2set-div-2*)
apply (*simp add*: *finite-vimage-Suc-iff*)
done

30.3.4 Proof of isomorphism

lemma *set2nat-nat2set* [*simp*]: *set2nat* (*nat2set n*) = *n*
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac n* = *0*, *simp*)
apply (*drule-tac x*=*n div 2* **in** *spec*, *simp*)
apply (*simp add*: *nat2set-div-2 set2nat-vimage-Suc*)

apply (*erule div2-even-ext-nat*)
apply (*simp add*: *even-set2nat-iff*)
done

lemma *nat2set-set2nat* [*simp*]: *finite A* $\Longrightarrow$ *nat2set* (*set2nat A*) = *A*
apply (*erule finite-induct*, *simp-all*)
apply (*simp add*: *nat2set-plus-power-2*)
done

lemma *inj-nat2set*: *inj nat2set*
by (*rule inj-on-inverseI*, *rule set2nat-nat2set*)

lemma *nat2set-eq-iff* [*simp*]: *nat2set x* = *nat2set y* $\longleftrightarrow$ *x* = *y*
by (*rule inj-eq* [*OF inj-nat2set*])

lemma *inj-on-set2nat*: *inj-on set2nat* (*Collect finite*)
by (*rule inj-on-inverseI* [**where** *g*=*nat2set*], *simp*)

lemma *set2nat-eq-iff* [*simp*]:
  $[\![$*finite A*; *finite B*$]\!] \Longrightarrow$ *set2nat A* = *set2nat B* $\longleftrightarrow$ *A* = *B*
by (*rule iffI*, *simp add*: *inj-onD* [*OF inj-on-set2nat*], *simp*)

30.3.5 Ordering properties

lemma *nat-less-power2*: *n* < *2^n*
by (*induct n*) *simp-all*

lemma *less-set2nat*: $[\![$*finite A*; *x* $\in$ *A*$]\!] \Longrightarrow$ *x* < *set2nat A*
unfolding *set2nat-def*
apply (*rule order-less-le-trans* [**where** *y*=*setsum* (*op ^ 2*) {*x*}])
apply (*simp add*: *nat-less-power2*)
apply (*erule setsum-mono2*, *simp*, *simp*)
done

end

theory *Universal*
imports *CompactBasis NatIso*
begin

30.4 Basis datatype

types *ubasis* = *nat*

definition
  node :: *nat* $\Rightarrow$ *ubasis* $\Rightarrow$ *ubasis set* $\Rightarrow$ *ubasis*
where
  node i a S = *Suc* (*prod2nat* (*i*, *prod2nat* (*a*, *set2nat S*)))

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

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

lemma *node-inject* [*simp*]:
⟦*finite S*; *finite T*⟧
⟹ *node i a S* = *node j b T* ⟷ *i* = *j* ∧ *a* = *b* ∧ *S* = *T*
unfolding *node-def* **by** *simp*

lemma *node-gt0*: *i* < *node i a S*
unfolding *node-def less-Suc-eq-le*
by (*rule le-prod2nat-1*)

lemma *node-gt1*: *a* < *node i a S*
unfolding *node-def less-Suc-eq-le*
by (*rule order-trans* [*OF le-prod2nat-1 le-prod2nat-2*])

lemma *nat-less-power2*: *n* < *2^n*
by (*induct n*) *simp-all*

lemma *node-gt2*: ⟦*finite S*; *b* ∈ *S*⟧ ⟹ *b* < *node i a S*
unfolding *node-def less-Suc-eq-le set2nat-def*
apply (*rule order-trans* [*OF - le-prod2nat-2*])
apply (*rule order-trans* [*OF - le-prod2nat-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-prod2nat-pairI*:
⟦*fst* (*nat2prod x*) = *a*; *snd* (*nat2prod x*) = *b*⟧ ⟹ *x* = *prod2nat* (*a*, *b*)
by (*erule subst*, *erule subst*, *simp*)

lemma *node-cases*:
assumes *1*: *x* = *0* ⟹ *P*
assumes *2*: ⋀*i a S*. ⟦*finite S*; *x* = *node i a S*⟧ ⟹ *P*
shows *P*
apply (*cases x*)
apply (*erule 1*)
apply (*rule 2*)
apply (*rule finite-nat2set*)
apply (*simp add*: *node-def*)
apply (*rule eq-prod2nat-pairI* [*OF refl*])
apply (*rule eq-prod2nat-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

30.5 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* ∈ *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

30.5.1 Generic take function

function
ubasis-until :: (*ubasis* ⇒ *bool*) ⇒ *ubasis* ⇒ *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*
apply *simp*
done

termination *ubasis-until*
apply (*relation measure snd*)
apply (*rule wf-measure*)

apply (*simp add*: *node-gt1*)
done

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

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

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

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

lemma *ubasis-until-0*:
$\forall x.\ x \neq 0 \longrightarrow \neg\ P\ x \Longrightarrow \mathit{ubasis\text{-}until}\ P\ x = 0$
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 $\bigwedge i\ a\ S\ b.\ [\![\mathit{finite}\ S;\ P\ (\mathit{node}\ i\ a\ S);\ b \in S;\ \mathit{ubasis\text{-}le}\ a\ b]\!] \Longrightarrow P\ b$
shows *ubasis-le a b* $\Longrightarrow$ *ubasis-le* (*ubasis-until P a*) (*ubasis-until P b*)
proof (*induct set*: *ubasis-le*)
case (*ubasis-le-refl a*) **show** *?case* **by** (*rule ubasis-le.ubasis-le-refl*)
next
case (*ubasis-le-trans a b c*) **thus** *?case* **by** − (*rule ubasis-le.ubasis-le-trans*)

next
case (*ubasis-le-lower S a i*) **thus** *?case*
apply (*clarsimp simp add*: *ubasis-le-refl*)
apply (*rule ubasis-le-trans* [*OF ubasis-until-less*])
apply (*erule ubasis-le.ubasis-le-lower*)
done
next
case (*ubasis-le-upper S b a i*) **thus** *?case*
apply *clarsimp*
apply (*subst ubasis-until-same*)
apply (*erule* (*3*) *prems*)
apply (*erule* (*2*) *ubasis-le.ubasis-le-upper*)
done
qed

lemma *finite-range-ubasis-until*:
finite {*x*. *P x*} ⟹ *finite* (*range* (*ubasis-until P*))
apply (*rule finite-subset* [**where** *B*=*insert 0* {*x*. *P x*}])
apply (*clarsimp simp add*: *ubasis-until'*)
apply *simp*
done

30.5.2 Take function for *ubasis*

definition
ubasis-take :: *nat* ⇒ *ubasis* ⇒ *ubasis*
where
ubasis-take n = *ubasis-until* (λ*x*. *x* ≤ *n*)

lemma *ubasis-take-le*: *ubasis-take n x* ≤ *n*
unfolding *ubasis-take-def* **by** (*rule ubasis-until*, *rule le0*)

lemma *ubasis-take-same*: *x* ≤ *n* ⟹ *ubasis-take n x* = *x*
unfolding *ubasis-take-def* **by** (*rule ubasis-until-same*)

lemma *ubasis-take-idem*: *ubasis-take n* (*ubasis-take n x*) = *ubasis-take n x*
by (*rule ubasis-take-same* [*OF ubasis-take-le*])

lemma *ubasis-take-0* [*simp*]: *ubasis-take 0 x* = *0*
unfolding *ubasis-take-def* **by** (*simp add*: *ubasis-until-0*)

lemma *ubasis-take-less*: *ubasis-le* (*ubasis-take n x*) *x*
unfolding *ubasis-take-def* **by** (*rule ubasis-until-less*)

lemma *ubasis-take-chain*: *ubasis-le* (*ubasis-take n x*) (*ubasis-take* (*Suc n*) *x*)
unfolding *ubasis-take-def* **by** (*rule ubasis-until-chain*) *simp*

lemma *ubasis-take-mono*:
assumes *ubasis-le x y*

shows *ubasis-le* (*ubasis-take n x*) (*ubasis-take n y*)
unfolding *ubasis-take-def*
apply (*rule ubasis-until-mono* [*OF - prems*])
apply (*frule* (*2*) *order-less-le-trans* [*OF node-gt2*])
apply (*erule order-less-imp-le*)
done

lemma *finite-range-ubasis-take*: *finite* (*range* (*ubasis-take n*))
apply (*rule finite-subset* [**where** *B*={..*n*}])
apply (*simp add*: *subset-eq ubasis-take-le*)
apply *simp*
done

lemma *ubasis-take-covers*: ∃ *n*. *ubasis-take n x* = *x*
apply (*rule exI* [**where** *x*=*x*])
apply (*simp add*: *ubasis-take-same*)
done

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

interpretation *udom*: *basis-take ubasis-le ubasis-take*
apply *default*
apply (*rule ubasis-take-less*)
apply (*rule ubasis-take-idem*)
apply (*erule ubasis-take-mono*)
apply (*rule ubasis-take-chain*)
apply (*rule finite-range-ubasis-take*)
apply (*rule ubasis-take-covers*)
done

30.6 Defining the universal domain by ideal completion

typedef (**open**) *udom* = {*S*. *udom.ideal S*}
by (*fast intro*: *udom.ideal-principal*)

instantiation *udom* :: *sq-ord*
begin

definition
x ⊑ *y* ⟷ *Rep-udom x* ⊆ *Rep-udom y*

instance ..
end

instance *udom* :: *po*

by (*rule udom.typedef-ideal-po*
[*OF type-definition-udom sq-le-udom-def*])

instance *udom* :: *cpo*
by (*rule udom.typedef-ideal-cpo*
[*OF type-definition-udom sq-le-udom-def*])

lemma *Rep-udom-lub*:
chain Y $\Longrightarrow$ *Rep-udom* ($\bigsqcup i.\ Y\ i$) = ($\bigcup i.$ *Rep-udom* (*Y i*))
by (*rule udom.typedef-ideal-rep-contlub*
[*OF type-definition-udom sq-le-udom-def*])

lemma *ideal-Rep-udom*: *udom.ideal* (*Rep-udom xs*)
by (*rule Rep-udom* [*unfolded mem-Collect-eq*])

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

lemma *Rep-udom-principal*:
Rep-udom (*udom-principal t*) = {*u. ubasis-le u t*}
unfolding *udom-principal-def*
by (*simp add*: *Abs-udom-inverse udom.ideal-principal*)

interpretation *udom*:
ideal-completion ubasis-le ubasis-take udom-principal Rep-udom
apply *unfold-locales*
apply (*rule ideal-Rep-udom*)
apply (*erule Rep-udom-lub*)
apply (*rule Rep-udom-principal*)
apply (*simp only*: *sq-le-udom-def*)
done

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*: $\bot$ = *udom-principal 0*
by (*rule udom-minimal* [*THEN UU-I*, *symmetric*])

Universal domain is bifinite

instantiation *udom* :: *bifinite*
begin

definition
approx-udom-def: *approx* = *udom.completion-approx*

instance
apply (*intro-classes*, *unfold approx-udom-def*)
apply (*rule udom.chain-completion-approx*)
apply (*rule udom.lub-completion-approx*)
apply (*rule udom.completion-approx-idem*)
apply (*rule udom.finite-fixes-completion-approx*)
done

end

lemma *approx-udom-principal* [*simp*]:
approx n·(*udom-principal x*) = *udom-principal* (*ubasis-take n x*)
unfolding *approx-udom-def*
by (*rule udom.completion-approx-principal*)

lemma *approx-eq-udom-principal*:
∃ *a*∈*Rep-udom x*. *approx n*·*x* = *udom-principal* (*ubasis-take n a*)
unfolding *approx-udom-def*
by (*rule udom.completion-approx-eq-principal*)

30.7 Universality of *udom*

defaultsort *bifinite*

30.7.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
fixes *A* :: *'a*::*po set*
shows ⟦*finite A*; *A* ≠ {}⟧ ⟹ ∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*
proof (*induct rule*: *finite-ne-induct*)
case (*singleton x*)
show *?case* **by** *simp*
next
case (*insert a A*)
from ‹∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*›
obtain *x* **where** *x*: *x* ∈ *A*
and *x-eq*: ⋀*y*. ⟦*y* ∈ *A*; *x* ⊑ *y*⟧ ⟹ *x* = *y* **by** *fast*
show *?case*
proof (*intro bexI ballI impI*)
fix *y*
assume *y* ∈ *insert a A* **and** (*if x* ⊑ *a then a else x*) ⊑ *y*
thus (*if x* ⊑ *a then a else x*) = *y*
apply *auto*
apply (*frule* (*1*) *trans-less*)
apply (*frule* (*1*) *x-eq*)
apply (*rule antisym-less*, *assumption*)
apply *simp*

```
    apply (erule (1) x-eq)
    done
  next
    show (if x ⊑ a then a else x) ∈ insert a A
    by (simp add: x)
  qed
qed

definition
  choose :: 'a compact-basis set ⇒ 'a compact-basis
where
  choose A = (SOME x. x ∈ {x∈A. ∀ y∈A. x ⊑ y ⟶ x = y})

lemma choose-lemma:
  ⟦finite A; A ≠ {}⟧ ⟹ choose A ∈ {x∈A. ∀ y∈A. x ⊑ y ⟶ x = y}
unfolding choose-def
apply (rule someI-ex)
apply (frule (1) finite-has-maximal, fast)
done

lemma maximal-choose:
  ⟦finite A; y ∈ A; choose A ⊑ y⟧ ⟹ choose A = y
apply (cases A = {}, simp)
apply (frule (1) choose-lemma, simp)
done

lemma choose-in: ⟦finite A; A ≠ {}⟧ ⟹ choose A ∈ A
by (frule (1) choose-lemma, simp)

function
  choose-pos :: 'a compact-basis set ⇒ 'a compact-basis ⇒ nat
where
  choose-pos A x =
    (if finite A ∧ x ∈ A ∧ x ≠ choose A
      then Suc (choose-pos (A − {choose A}) x) else 0)
by auto

termination choose-pos
apply (relation measure (card ∘ fst), simp)
apply clarsimp
apply (rule card-Diff1-less)
apply assumption
apply (erule choose-in)
apply clarsimp
done

declare choose-pos.simps [simp del]

lemma choose-pos-choose: finite A ⟹ choose-pos A (choose A) = 0
```

by (*simp add*: *choose-pos.simps*)

lemma *inj-on-choose-pos* [*OF refl*]:
⟦*card A* = *n*; *finite A*⟧ ⟹ *inj-on* (*choose-pos A*) *A*
apply (*induct n arbitrary*: *A*)
apply *simp*
apply (*case-tac A* = {}, *simp*)
apply (*frule* (*1*) *choose-in*)
apply (*rule inj-onI*)
apply (*drule-tac x*=*A* − {*choose A*} **in** *meta-spec*, *simp*)
apply (*simp add*: *choose-pos.simps*)
apply (*simp split*: *split-if-asm*)
apply (*erule* (*1*) *inj-onD*, *simp*, *simp*)
done

lemma *choose-pos-bounded* [*OF refl*]:
⟦*card A* = *n*; *finite A*; *x* ∈ *A*⟧ ⟹ *choose-pos A x* < *n*
apply (*induct n arbitrary*: *A*)
apply *simp*
apply (*case-tac A* = {}, *simp*)
apply (*frule* (*1*) *choose-in*)
apply (*subst choose-pos.simps*)
apply *simp*
done

lemma *choose-pos-lessD*:
⟦*choose-pos A x* < *choose-pos A y*; *finite A*; *x* ∈ *A*; *y* ∈ *A*⟧ ⟹ ¬ *x* ⊑ *y*
apply (*induct A x arbitrary*: *y rule*: *choose-pos.induct*)
apply *simp*
apply (*case-tac x* = *choose A*)
apply *simp*
apply (*rule notI*)
apply (*frule* (*2*) *maximal-choose*)
apply *simp*
apply (*case-tac y* = *choose A*)
apply (*simp add*: *choose-pos-choose*)
apply (*drule-tac x*=*y* **in** *meta-spec*)
apply *simp*
apply (*erule meta-mp*)
apply (*simp add*: *choose-pos.simps*)
done

30.7.2 Rank of basis elements

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

lemma *cb-take-covers*: $\exists n.\ cb\text{-}take\ n\ x = x$
apply (*rule exE* [*OF compact-basis.take-covers* [**where** $a{=}x$]])
apply (*rename-tac n*, *rule-tac* $x{=}Suc\ n$ **in** *exI*, *simp*)
done

lemma *cb-take-less*: $cb\text{-}take\ n\ x \sqsubseteq x$
by (*cases n*, *simp*, *simp add*: *compact-basis.take-less*)

lemma *cb-take-idem*: $cb\text{-}take\ n\ (cb\text{-}take\ n\ x) = cb\text{-}take\ n\ x$
by (*cases n*, *simp*, *simp add*: *compact-basis.take-take*)

lemma *cb-take-mono*: $x \sqsubseteq y \Longrightarrow cb\text{-}take\ n\ x \sqsubseteq cb\text{-}take\ n\ y$
by (*cases n*, *simp*, *simp add*: *compact-basis.take-mono*)

lemma *cb-take-chain-le*: $m \leq n \Longrightarrow cb\text{-}take\ m\ x \sqsubseteq cb\text{-}take\ n\ x$
apply (*cases m*, *simp*)
apply (*cases n*, *simp*)
apply (*simp add*: *compact-basis.take-chain-le*)
done

lemma *range-const*: $range\ (\lambda x.\ c) = \{c\}$
by *auto*

lemma *finite-range-cb-take*: *finite* $(range\ (cb\text{-}take\ n))$
apply (*cases n*)
apply (*simp add*: *range-const*)
apply (*simp add*: *compact-basis.finite-range-take*)
done

definition
$rank :: {'a}\ compact\text{-}basis \Rightarrow nat$
where
$rank\ x = (LEAST\ n.\ cb\text{-}take\ n\ x = x)$

lemma *compact-approx-rank*: $cb\text{-}take\ (rank\ x)\ x = x$
unfolding *rank-def*
apply (*rule LeastI-ex*)
apply (*rule cb-take-covers*)
done

lemma *rank-leD*: $rank\ x \leq n \Longrightarrow cb\text{-}take\ n\ x = x$
apply (*rule antisym-less* [*OF cb-take-less*])
apply (*subst compact-approx-rank* [*symmetric*])
apply (*erule cb-take-chain-le*)
done

lemma *rank-leI*: $cb\text{-}take\ n\ x = x \Longrightarrow rank\ x \leq n$
unfolding *rank-def* **by** (*rule Least-le*)

lemma *rank-le-iff*: *rank x* $\leq$ *n* $\longleftrightarrow$ *cb-take n x* = *x*
by (*rule iffI* [*OF rank-leD rank-leI*])

lemma *rank-compact-bot* [*simp*]: *rank compact-bot* = *0*
using *rank-leI* [*of 0 compact-bot*] **by** *simp*

lemma *rank-eq-0-iff* [*simp*]: *rank x* = *0* $\longleftrightarrow$ *x* = *compact-bot*
using *rank-le-iff* [*of x 0*] **by** *auto*

definition
rank-le :: *'a compact-basis* $\Rightarrow$ *'a compact-basis set*
where
rank-le x = {*y*. *rank y* $\leq$ *rank x*}

definition
rank-lt :: *'a compact-basis* $\Rightarrow$ *'a compact-basis set*
where
rank-lt x = {*y*. *rank y* < *rank x*}

definition
rank-eq :: *'a compact-basis* $\Rightarrow$ *'a compact-basis set*
where
rank-eq x = {*y*. *rank y* = *rank x*}

lemma *rank-eq-cong*: *rank x* = *rank y* $\Longrightarrow$ *rank-eq x* = *rank-eq y*
unfolding *rank-eq-def* **by** *simp*

lemma *rank-lt-cong*: *rank x* = *rank y* $\Longrightarrow$ *rank-lt x* = *rank-lt y*
unfolding *rank-lt-def* **by** *simp*

lemma *rank-eq-subset*: *rank-eq x* $\subseteq$ *rank-le x*
unfolding *rank-eq-def rank-le-def* **by** *auto*

lemma *rank-lt-subset*: *rank-lt x* $\subseteq$ *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*

30.7.3 Sequencing basis elements

definition
place :: *'a compact-basis* ⇒ *nat*
where
place x = *card* (*rank-lt x*) + *choose-pos* (*rank-eq x*) *x*

lemma *place-bounded*: *place x* < *card* (*rank-le x*)
unfolding *place-def*
apply (*rule ord-less-eq-trans*)
apply (*rule add-strict-left-mono*)
apply (*rule choose-pos-bounded*)
apply (*rule finite-rank-eq*)
apply (*simp add*: *rank-eq-def*)
apply (*subst card-Un-disjoint* [*symmetric*])
apply (*rule finite-rank-lt*)
apply (*rule finite-rank-eq*)
apply (*rule rank-lt-Int-rank-eq*)
apply (*simp add*: *rank-lt-Un-rank-eq*)
done

lemma *place-ge*: *card* (*rank-lt x*) ≤ *place x*
unfolding *place-def* **by** *simp*

lemma *place-rank-mono*:
fixes *x y* :: *'a compact-basis*
shows *rank x* < *rank y* ⟹ *place x* < *place y*
apply (*rule less-le-trans* [*OF place-bounded*])
apply (*rule order-trans* [*OF - place-ge*])
apply (*rule card-mono*)
apply (*rule finite-rank-lt*)
apply (*simp add*: *rank-le-def rank-lt-def subset-eq*)
done

lemma *place-eqD*: *place x* = *place y* ⟹ *x* = *y*
apply (*rule linorder-cases* [**where** *x*=*rank x* **and** *y*=*rank y*])
apply (*drule place-rank-mono*, *simp*)
apply (*simp add*: *place-def*)
apply (*rule inj-on-choose-pos* [**where** *A*=*rank-eq x*, *THEN inj-onD*])
apply (*rule finite-rank-eq*)

apply (*simp cong*: *rank-lt-cong rank-eq-cong*)
apply (*simp add*: *rank-eq-def*)
apply (*simp add*: *rank-eq-def*)
apply (*drule place-rank-mono*, *simp*)
done

lemma *inj-place*: *inj place*
by (*rule inj-onI*, *erule place-eqD*)

30.7.4 Embedding and projection on basis elements

definition
sub :: *'a compact-basis* ⇒ *'a compact-basis*
where
sub x = (*case rank x of 0* ⇒ *compact-bot* | *Suc k* ⇒ *cb-take k x*)

lemma *rank-sub-less*: *x* ≠ *compact-bot* ⟹ *rank* (*sub x*) < *rank x*
unfolding *sub-def*
apply (*cases rank x*, *simp*)
apply (*simp add*: *less-Suc-eq-le*)
apply (*rule rank-leI*)
apply (*rule cb-take-idem*)
done

lemma *place-sub-less*: *x* ≠ *compact-bot* ⟹ *place* (*sub x*) < *place x*
apply (*rule place-rank-mono*)
apply (*erule rank-sub-less*)
done

lemma *sub-below*: *sub x* ⊑ *x*
unfolding *sub-def* **by** (*cases rank x*, *simp-all add*: *cb-take-less*)

lemma *rank-less-imp-below-sub*: ⟦*x* ⊑ *y*; *rank x* < *rank y*⟧ ⟹ *x* ⊑ *sub y*
unfolding *sub-def*
apply (*cases rank y*, *simp*)
apply (*simp add*: *less-Suc-eq-le*)
apply (*subgoal-tac cb-take nat x* ⊑ *cb-take nat y*)
apply (*simp add*: *rank-leD*)
apply (*erule cb-take-mono*)
done

function
basis-emb :: *'a compact-basis* ⇒ *ubasis*
where
basis-emb x = (*if x* = *compact-bot then 0 else*
node (*place x*) (*basis-emb* (*sub x*))
(*basis-emb* ‘ {*y*. *place y* < *place x* ∧ *x* ⊑ *y*}))
by *auto*

termination *basis-emb*
apply (*relation measure place*, *simp*)
apply (*simp add*: *place-sub-less*)
apply *simp*
done

declare *basis-emb.simps* [*simp del*]

lemma *basis-emb-compact-bot* [*simp*]: *basis-emb compact-bot* = *0*
by (*simp add*: *basis-emb.simps*)

lemma *fin1*: *finite* {*y*. *place y* < *place x* ∧ *x* ⊑ *y*}
apply (*subst Collect-conj-eq*)
apply (*rule finite-Int*)
apply (*rule disjI1*)
apply (*subgoal-tac finite* (*place* −‘ {*n*. *n* < *place x*}), *simp*)
apply (*rule finite-vimageI* [*OF - inj-place*])
apply (*simp add*: *lessThan-def* [*symmetric*])
done

lemma *fin2*: *finite* (*basis-emb* ‘ {*y*. *place y* < *place x* ∧ *x* ⊑ *y*})
by (*rule finite-imageI* [*OF fin1*])

lemma *rank-place-mono*:
⟦*place x* < *place y*; *x* ⊑ *y*⟧ ⟹ *rank x* < *rank y*
apply (*rule linorder-cases*, *assumption*)
apply (*simp add*: *place-def cong*: *rank-lt-cong rank-eq-cong*)
apply (*drule choose-pos-lessD*)
apply (*rule finite-rank-eq*)
apply (*simp add*: *rank-eq-def*)
apply (*simp add*: *rank-eq-def*)
apply *simp*
apply (*drule place-rank-mono*, *simp*)
done

lemma *basis-emb-mono*:
x ⊑ *y* ⟹ *ubasis-le* (*basis-emb x*) (*basis-emb y*)
proof (*induct n* ≡ *max* (*place x*) (*place y*) *arbitrary*: *x y rule*: *less-induct*)
case (*less n*)
hence *IH*:
⋀(*a*::′*a compact-basis*) *b*.
⟦*max* (*place a*) (*place b*) < *max* (*place x*) (*place y*); *a* ⊑ *b*⟧
⟹ *ubasis-le* (*basis-emb a*) (*basis-emb b*)
by *simp*
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 IH)
     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 IH)
     apply (simp add: less-max-iff-disj)
     apply (erule place-sub-less)
    apply (erule rev-trans-less)
    apply (rule sub-below)
    done
 qed
qed

lemma inj-basis-emb: inj basis-emb
 apply (rule inj-onI)
 apply (case-tac x = compact-bot)
  apply (case-tac [!] y = compact-bot)
   apply simp
  apply (simp add: basis-emb.simps)
 apply (simp add: basis-emb.simps)
 apply (simp add: basis-emb.simps)
 apply (simp add: fin2 inj-eq [OF inj-place])
done

definition
  basis-prj :: ubasis ⇒ 'a compact-basis
where
  basis-prj x = inv basis-emb
    (ubasis-until (λx. x ∈ range (basis-emb :: 'a compact-basis ⇒ ubasis)) x)

lemma basis-prj-basis-emb: ⋀x. basis-prj (basis-emb x) = x
unfolding basis-prj-def
 apply (subst ubasis-until-same)
  apply (rule rangeI)
 apply (rule inv-f-f)
```

apply (*rule inj-basis-emb*)
done

lemma *basis-prj-node*:
⟦*finite S*; *node i a S* ∉ *range* (*basis-emb* :: *'a compact-basis* ⇒ *nat*)⟧
⟹ *basis-prj* (*node i a S*) = (*basis-prj a* :: *'a compact-basis*)
unfolding *basis-prj-def* **by** *simp*

lemma *basis-prj-0*: *basis-prj 0* = *compact-bot*
apply (*subst basis-emb-compact-bot* [*symmetric*])
apply (*rule basis-prj-basis-emb*)
done

lemma *node-eq-basis-emb-iff*:
finite S ⟹ *node i a S* = *basis-emb x* ⟷
x ≠ *compact-bot* ∧ *i* = *place x* ∧ *a* = *basis-emb* (*sub x*) ∧
S = *basis-emb* ‘ {*y*. *place y* < *place x* ∧ *x* ⊑ *y*}
apply (*cases x* = *compact-bot*, *simp*)
apply (*simp add*: *basis-emb.simps* [*of x*])
apply (*simp add*: *fin2*)
done

lemma *basis-prj-mono*: *ubasis-le a b* ⟹ *basis-prj a* ⊑ *basis-prj b*
proof (*induct a b rule*: *ubasis-le.induct*)
case (*ubasis-le-refl a*) **show** *?case* **by** (*rule refl-less*)
next
case (*ubasis-le-trans a b c*) **thus** *?case* **by** − (*rule trans-less*)
next
case (*ubasis-le-lower S a i*) **thus** *?case*
apply (*cases node i a S* ∈ *range* (*basis-emb* :: *'a compact-basis* ⇒ *nat*))
apply (*erule rangeE*, *rename-tac x*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*simp add*: *node-eq-basis-emb-iff*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*rule sub-below*)
apply (*simp add*: *basis-prj-node*)
done
next
case (*ubasis-le-upper S b a i*) **thus** *?case*
apply (*cases node i a S* ∈ *range* (*basis-emb* :: *'a compact-basis* ⇒ *nat*))
apply (*erule rangeE*, *rename-tac x*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*clarsimp simp add*: *node-eq-basis-emb-iff*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*simp add*: *basis-prj-node*)
done
qed

lemma *basis-emb-prj-less*: *ubasis-le* (*basis-emb* (*basis-prj x*)) *x*

unfolding *basis-prj-def*
apply (*subst f-inv-f* [**where** *f*=*basis-emb*])
apply (*rule ubasis-until*)
apply (*rule range-eqI* [**where** *x*=*compact-bot*])
apply *simp*
apply (*rule ubasis-until-less*)
done

hide (**open**) *const*
node
choose
choose-pos
place
sub

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

definition
udom-emb :: *'a*::*bifinite* → *udom*
where
udom-emb = *compact-basis.basis-fun* (λ*x*. *udom-principal* (*basis-emb x*))

definition
udom-prj :: *udom* → *'a*::*bifinite*
where
udom-prj = *udom.basis-fun* (λ*x*. *Rep-compact-basis* (*basis-prj x*))

lemma *udom-emb-principal*:
udom-emb·(*Rep-compact-basis x*) = *udom-principal* (*basis-emb x*)
unfolding *udom-emb-def*
apply (*rule compact-basis.basis-fun-principal*)
apply (*rule udom.principal-mono*)
apply (*erule basis-emb-mono*)
done

lemma *udom-prj-principal*:
udom-prj·(*udom-principal x*) = *Rep-compact-basis* (*basis-prj x*)
unfolding *udom-prj-def*
apply (*rule udom.basis-fun-principal*)
apply (*rule compact-basis.principal-mono*)
apply (*erule basis-prj-mono*)
done

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
apply *default*
apply (*rule compact-basis.principal-induct*, *simp*)
apply (*simp add*: *udom-emb-principal udom-prj-principal*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*rule udom.principal-induct*, *simp*)

apply (*simp add*: *udom-emb-principal udom-prj-principal*)
apply (*rule basis-emb-prj-less*)
done

end

31 Sum-Cpo: The cpo of disjoint sums

theory *Sum-Cpo*
imports *Bifinite*
begin

31.1 Ordering on type $'a + 'b$

instantiation + :: (*sq-ord*, *sq-ord*) *sq-ord*
begin

definition
less-sum-def: $x \sqsubseteq y \equiv$ *case* x *of*
Inl $a \Rightarrow$ (*case* y *of Inl* $b \Rightarrow a \sqsubseteq b \mid$ *Inr* $b \Rightarrow$ *False*) $\mid$
Inr $a \Rightarrow$ (*case* y *of Inl* $b \Rightarrow$ *False* $\mid$ *Inr* $b \Rightarrow a \sqsubseteq b$)

instance ..
end

lemma *Inl-less-iff* [*simp*]: *Inl* $x \sqsubseteq$ *Inl* $y = x \sqsubseteq y$
unfolding *less-sum-def* **by** *simp*

lemma *Inr-less-iff* [*simp*]: *Inr* $x \sqsubseteq$ *Inr* $y = x \sqsubseteq y$
unfolding *less-sum-def* **by** *simp*

lemma *Inl-less-Inr* [*simp*]: $\neg$ *Inl* $x \sqsubseteq$ *Inr* y
unfolding *less-sum-def* **by** *simp*

lemma *Inr-less-Inl* [*simp*]: $\neg$ *Inr* $x \sqsubseteq$ *Inl* y
unfolding *less-sum-def* **by** *simp*

lemma *Inl-mono*: $x \sqsubseteq y \Longrightarrow$ *Inl* $x \sqsubseteq$ *Inl* y
by *simp*

lemma *Inr-mono*: $x \sqsubseteq y \Longrightarrow$ *Inr* $x \sqsubseteq$ *Inr* y
by *simp*

lemma *Inl-lessE*: $[\![$*Inl* $a \sqsubseteq x$; $\bigwedge b.\ [\![x =$ *Inl* b; $a \sqsubseteq b]\!] \Longrightarrow R]\!] \Longrightarrow R$
by (*cases* x, *simp-all*)

lemma *Inr-lessE*: $[\![$*Inr* $a \sqsubseteq x$; $\bigwedge b.\ [\![x =$ *Inr* b; $a \sqsubseteq b]\!] \Longrightarrow R]\!] \Longrightarrow R$
by (*cases* x, *simp-all*)

lemmas *sum-less-elims* = *Inl-lessE Inr-lessE*

lemma *sum-less-cases*:
  ⟦*x* ⊑ *y*;
    ⋀*a b*. ⟦*x* = *Inl a*; *y* = *Inl b*; *a* ⊑ *b*⟧ ⟹ *R*;
    ⋀*a b*. ⟦*x* = *Inr a*; *y* = *Inr b*; *a* ⊑ *b*⟧ ⟹ *R*⟧
    ⟹ *R*
by (*cases x*, *safe elim*!: *sum-less-elims*, *auto*)

31.2 Sum type is a complete partial order

instance + :: (*po*, *po*) *po*
proof
  fix *x* :: 'a + 'b
  show *x* ⊑ *x*
    by (*induct x*, *simp-all*)
next
  fix *x y* :: 'a + 'b
  assume *x* ⊑ *y* **and** *y* ⊑ *x* **thus** *x* = *y*
    by (*induct x*, *auto elim*!: *sum-less-elims intro*: *antisym-less*)
next
  fix *x y z* :: 'a + 'b
  assume *x* ⊑ *y* **and** *y* ⊑ *z* **thus** *x* ⊑ *z*
    by (*induct x*, *auto elim*!: *sum-less-elims intro*: *trans-less*)
qed

lemma *monofun-inv-Inl*: *monofun* (λ*p*. *THE a*. *p* = *Inl a*)
by (*rule monofunI*, *erule sum-less-cases*, *simp-all*)

lemma *monofun-inv-Inr*: *monofun* (λ*p*. *THE b*. *p* = *Inr b*)
by (*rule monofunI*, *erule sum-less-cases*, *simp-all*)

lemma *sum-chain-cases*:
  assumes *Y*: *chain Y*
  assumes *A*: ⋀*A*. ⟦*chain A*; *Y* = (λ*i*. *Inl* (*A i*))⟧ ⟹ *R*
  assumes *B*: ⋀*B*. ⟦*chain B*; *Y* = (λ*i*. *Inr* (*B i*))⟧ ⟹ *R*
  shows *R*
 apply (*cases Y 0*)
  apply (*rule A*)
   apply (*rule ch2ch-monofun* [*OF monofun-inv-Inl Y*])
  apply (*rule ext*)
  apply (*cut-tac j*=*i* **in** *chain-mono* [*OF Y le0*], *simp*)
  apply (*erule Inl-lessE*, *simp*)
 apply (*rule B*)
  apply (*rule ch2ch-monofun* [*OF monofun-inv-Inr Y*])
 apply (*rule ext*)
 apply (*cut-tac j*=*i* **in** *chain-mono* [*OF Y le0*], *simp*)
 apply (*erule Inr-lessE*, *simp*)

done

lemma *is-lub-Inl*: *range S <<| x ⟹ range (λi. Inl (S i)) <<| Inl x*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*simp add*: *is-ub-lub*)
apply (*frule ub-rangeD* [**where** *i*=*arbitrary*])
apply (*erule Inl-lessE*, *simp*)
apply (*erule is-lub-lub*)
apply (*rule ub-rangeI*)
apply (*drule ub-rangeD*, *simp*)
done

lemma *is-lub-Inr*: *range S <<| x ⟹ range (λi. Inr (S i)) <<| Inr x*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*simp add*: *is-ub-lub*)
apply (*frule ub-rangeD* [**where** *i*=*arbitrary*])
apply (*erule Inr-lessE*, *simp*)
apply (*erule is-lub-lub*)
apply (*rule ub-rangeI*)
apply (*drule ub-rangeD*, *simp*)
done

instance + :: (*cpo*, *cpo*) *cpo*
apply *intro-classes*
apply (*erule sum-chain-cases*, *safe*)
apply (*rule exI*)
apply (*rule is-lub-Inl*)
apply (*erule cpo-lubI*)
apply (*rule exI*)
apply (*rule is-lub-Inr*)
apply (*erule cpo-lubI*)
done

31.3 Continuity of *Inl*, *Inr*, *sum-case*

lemma *cont2cont-Inl* [*simp*]: *cont f ⟹ cont (λx. Inl (f x))*
by (*fast intro*: *contI is-lub-Inl elim*: *contE*)

lemma *cont2cont-Inr* [*simp*]: *cont f ⟹ cont (λx. Inr (f x))*
by (*fast intro*: *contI is-lub-Inr elim*: *contE*)

lemma *cont-Inl*: *cont Inl*
by (*rule cont2cont-Inl* [*OF cont-id*])

lemma *cont-Inr*: *cont Inr*
by (*rule cont2cont-Inr* [*OF cont-id*])

lemmas *ch2ch-Inl* [*simp*] = *ch2ch-cont* [*OF cont-Inl*]
lemmas *ch2ch-Inr* [*simp*] = *ch2ch-cont* [*OF cont-Inr*]

lemmas *lub-Inl* = *cont2contlubE* [*OF cont-Inl*, *symmetric*]
lemmas *lub-Inr* = *cont2contlubE* [*OF cont-Inr*, *symmetric*]

lemma *cont-sum-case1*:
  assumes *f*: $\bigwedge a.$ *cont* ($\lambda x.$ *f x a*)
  assumes *g*: $\bigwedge b.$ *cont* ($\lambda x.$ *g x b*)
  shows *cont* ($\lambda x.$ *case y of Inl a* $\Rightarrow$ *f x a* | *Inr b* $\Rightarrow$ *g x b*)
by (*induct y*, *simp add*: *f*, *simp add*: *g*)

lemma *cont-sum-case2*: $[\![$*cont f*; *cont g*$]\!] \Longrightarrow$ *cont* (*sum-case f g*)
apply (*rule contI*)
apply (*erule sum-chain-cases*)
apply (*simp add*: *cont2contlubE* [*OF cont-Inl*, *symmetric*] *contE*)
apply (*simp add*: *cont2contlubE* [*OF cont-Inr*, *symmetric*] *contE*)
done

lemma *cont2cont-sum-case* [*simp*]:
  assumes *f1*: $\bigwedge a.$ *cont* ($\lambda x.$ *f x a*) **and** *f2*: $\bigwedge x.$ *cont* ($\lambda a.$ *f x a*)
  assumes *g1*: $\bigwedge b.$ *cont* ($\lambda x.$ *g x b*) **and** *g2*: $\bigwedge x.$ *cont* ($\lambda b.$ *g x b*)
  assumes *h*: *cont* ($\lambda x.$ *h x*)
  shows *cont* ($\lambda x.$ *case h x of Inl a* $\Rightarrow$ *f x a* | *Inr b* $\Rightarrow$ *g x b*)
apply (*rule cont2cont-app2* [*OF cont2cont-lambda - h*])
apply (*rule cont-sum-case1* [*OF f1 g1*])
apply (*rule cont-sum-case2* [*OF f2 g2*])
done

31.4 Compactness and chain-finiteness

lemma *compact-Inl*: *compact a* $\Longrightarrow$ *compact* (*Inl a*)
apply (*rule compactI2*)
apply (*erule sum-chain-cases*, *safe*)
apply (*simp add*: *lub-Inl*)
apply (*erule* (*2*) *compactD2*)
apply (*simp add*: *lub-Inr*)
done

lemma *compact-Inr*: *compact a* $\Longrightarrow$ *compact* (*Inr a*)
apply (*rule compactI2*)
apply (*erule sum-chain-cases*, *safe*)
apply (*simp add*: *lub-Inl*)
apply (*simp add*: *lub-Inr*)
apply (*erule* (*2*) *compactD2*)
done

lemma *compact-Inl-rev*: *compact* (*Inl a*) $\Longrightarrow$ *compact a*
unfolding *compact-def*

by (*drule adm-subst* [*OF cont-Inl*], *simp*)

lemma *compact-Inr-rev*: *compact* (*Inr a*) ⟹ *compact a*
unfolding *compact-def*
by (*drule adm-subst* [*OF cont-Inr*], *simp*)

lemma *compact-Inl-iff* [*simp*]: *compact* (*Inl a*) = *compact a*
by (*safe elim*!: *compact-Inl compact-Inl-rev*)

lemma *compact-Inr-iff* [*simp*]: *compact* (*Inr a*) = *compact a*
by (*safe elim*!: *compact-Inr compact-Inr-rev*)

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

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

instance + :: (*discrete-cpo*, *discrete-cpo*) *discrete-cpo*
by *intro-classes* (*simp add*: *less-sum-def split*: *sum.split*)

31.5 Sum type is a bifinite domain

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

definition
approx-sum-def: *approx* =
(λ*n*. Λ *x*. *case x of Inl a* ⇒ *Inl* (*approx n*·*a*) | *Inr b* ⇒ *Inr* (*approx n*·*b*))

lemma *approx-Inl* [*simp*]: *approx n*·(*Inl x*) = *Inl* (*approx n*·*x*)
unfolding *approx-sum-def* **by** *simp*

lemma *approx-Inr* [*simp*]: *approx n*·(*Inr x*) = *Inr* (*approx n*·*x*)
unfolding *approx-sum-def* **by** *simp*

instance proof
fix *i* :: *nat* **and** *x* :: *'a* + *'b*
show *chain* (*approx* :: *nat* ⇒ *'a* + *'b* → *'a* + *'b*)
unfolding *approx-sum-def*
by (*rule ch2ch-LAM*, *case-tac x*, *simp-all*)
show (⨆ *i*. *approx i*·*x*) = *x*
by (*induct x*, *simp-all add*: *lub-Inl lub-Inr*)
show *approx i*·(*approx i*·*x*) = *approx i*·*x*
by (*induct x*, *simp-all*)
have {*x*::*'a* + *'b*. *approx i*·*x* = *x*} ⊆
{*x*::*'a*. *approx i*·*x* = *x*} <+> {*x*::*'b*. *approx i*·*x* = *x*}

```
    by (rule subsetI, case-tac x, simp-all add: InlI InrI)
  thus finite {x::'a + 'b. approx i·x = x}
    by (rule finite-subset,
        intro finite-Plus finite-fixes-approx)
qed

end

end
```

```
theory HOLCF
imports
  Domain ConvexPD Algebraic Universal Sum-Cpo Main
uses
  holcf-logic.ML
  Tools/cont-consts.ML
  Tools/cont-proc.ML
  Tools/domain/domain-library.ML
  Tools/domain/domain-syntax.ML
  Tools/domain/domain-axioms.ML
  Tools/domain/domain-theorems.ML
  Tools/domain/domain-extender.ML
  Tools/adm-tac.ML
begin

defaultsort pcpo

declaration ⟪ fn - =>
  Simplifier.map-ss (fn simpset => simpset addSolver
    (mk-solver' adm-tac (fn ss =>
      Adm.adm-tac (Simplifier.the-context ss)
        (cut-facts-tac (Simplifier.prems-of-ss ss) THEN' cont-tacRs ss))));
⟫

end
```