Isabelle/HOLCF — Higher-Order Logic of Computable Functions

March 13, 2025

Contents

1 Partial orders **3**
- 1.1 Type class for partial orders . . . 3
- 1.2 Upper bounds . . . 4
- 1.3 Least upper bounds . . . 5
- 1.4 Countable chains . . . 6
- 1.5 Finite chains . . . 7

2 Classes cpo and pcpo **9**
- 2.1 Complete partial orders . . . 9
- 2.2 Pointed cpos . . . 12
- 2.3 Chain-finite and flat cpos . . . 13
- 2.4 Discrete cpos . . . 14

3 Continuity and monotonicity **14**
- 3.1 Definitions . . . 14
- 3.2 Equivalence of alternate definition . . . 15
- 3.3 Collection of continuity rules . . . 16
- 3.4 Continuity of basic functions . . . 16
- 3.5 Finite chains and flat pcpos . . . 17

4 Admissibility and compactness **18**
- 4.1 Definitions . . . 18
- 4.2 Admissibility on chain-finite types . . . 19
- 4.3 Admissibility of special formulae and propagation . . . 19
- 4.4 Compactness . . . 21

5 Class instances for the full function space **22**
- 5.1 Full function space is a partial order . . . 22
- 5.2 Full function space is chain complete . . . 23
- 5.3 Full function space is pointed . . . 23
- 5.4 Propagation of monotonicity and continuity . . . 24

6 The cpo of cartesian products **25**
6.1 Unit type is a pcpo . . . 25
6.2 Product type is a partial order . . . 25
6.3 Monotonicity of *Pair*, *fst*, *snd* . . . 25
6.4 Product type is a cpo . . . 27
6.5 Product type is pointed . . . 27
6.6 Continuity of *Pair*, *fst*, *snd* . . . 28
6.7 Compactness and chain-finiteness . . . 30

7 Discrete cpo types **30**
7.1 Discrete cpo class instance . . . 30
7.2 *undiscr* . . . 31

8 Subtypes of pcpos **31**
8.1 Proving a subtype is a partial order . . . 31
8.2 Proving a subtype is finite . . . 31
8.3 Proving a subtype is chain-finite . . . 32
8.4 Proving a subtype is complete . . . 32
8.4.1 Continuity of *Rep* and *Abs* . . . 33
8.5 Proving subtype elements are compact . . . 34
8.6 Proving a subtype is pointed . . . 34
8.6.1 Strictness of *Rep* and *Abs* . . . 35
8.7 Proving a subtype is flat . . . 36
8.8 HOLCF type definition package . . . 36

9 The type of continuous functions **36**
9.1 Definition of continuous function type . . . 36
9.2 Syntax for continuous lambda abstraction . . . 36
9.3 Continuous function space is pointed . . . 38
9.4 Basic properties of continuous functions . . . 38
9.4.1 Beta-reduction simproc . . . 38
9.5 Continuity of application . . . 39
9.6 Continuity simplification procedure . . . 41
9.7 Miscellaneous . . . 42
9.8 Continuous injection-retraction pairs . . . 42
9.9 Identity and composition . . . 44
9.10 Strictified functions . . . 44
9.11 Continuity of let-bindings . . . 45

10 Continuous deflations and ep-pairs **46**
10.1 Continuous deflations . . . 46
10.2 Deflations with finite range . . . 48
10.3 Continuous embedding-projection pairs . . . 49
10.4 Uniqueness of ep-pairs . . . 52

10.5 Composing ep-pairs 53

11 The type of strict products 54
11.1 Definition of strict product type 54
11.2 Definitions of constants 54
11.3 Case analysis 55
11.4 Properties of *spair* 55
11.5 Properties of *sfst* and *ssnd* 56
11.6 Compactness 57
11.7 Properties of *ssplit* 57
11.8 Strict product preserves flatness 57

12 The type of lifted values 58
12.1 Definition of new type for lifting 58
12.2 Ordering on lifted cpo 58
12.3 Lifted cpo is a partial order 59
12.4 Lifted cpo is a cpo 59
12.5 Lifted cpo is pointed 60
12.6 Continuity of *Iup* and *Ifup* 60
12.7 Continuous versions of constants 61

13 Lifting types of class type to flat pcpo's 63
13.1 Lift as a datatype 63
13.2 Lift is flat 64
13.3 Continuity of *case-lift* 64
13.4 Further operations 64

14 The type of lifted booleans 65
14.1 Type definition and constructors 65
14.2 Case analysis 66
14.3 Boolean connectives 67
14.4 Rewriting of HOLCF operations to HOL functions 68
14.5 Compactness 68

15 The type of strict sums 69
15.1 Definition of strict sum type 69
15.2 Definitions of constructors 69
15.3 Properties of *sinl* and *sinr* 70
15.4 Case analysis 71
15.5 Case analysis combinator 72
15.6 Strict sum preserves flatness 72

16 The Strict Function Type 72

17 Map functions for various types 74
17.1 Map operator for continuous function space 74
17.2 Map operator for product type 76
17.3 Map function for lifted cpo 77
17.4 Map function for strict products 78
17.5 Map function for strict sums 80
17.6 Map operator for strict function space 82

18 The cpo of cartesian products 84
18.1 Continuous case function for unit type 84
18.2 Continuous version of split function 84
18.3 Convert all lemmas to the continuous versions 84

19 Profinite and bifinite cpos 84
19.1 Chains of finite deflations 84
19.2 Omega-profinite and bifinite domains 85
19.3 Building approx chains 85
19.4 Class instance proofs 87

20 Defining algebraic domains by ideal completion 90
20.1 Ideals over a preorder 90
20.2 Lemmas about least upper bounds 92
20.3 Locale for ideal completion 93
20.3.1 Principal ideals approximate all elements 94
20.4 Defining functions in terms of basis elements 96

21 A universal bifinite domain 99
21.1 Basis for universal domain 99
21.1.1 Basis datatype 99
21.1.2 Basis ordering 100
21.1.3 Generic take function 101
21.2 Defining the universal domain by ideal completion 102
21.3 Compact bases of domains 103
21.4 Universality of *udom* 104
21.4.1 Choosing a maximal element from a finite set 104
21.4.2 Compact basis take function 106
21.4.3 Rank of basis elements 107
21.4.4 Sequencing basis elements 109
21.4.5 Embedding and projection on basis elements 110
21.4.6 EP-pair from any bifinite domain into *udom* 115
21.5 Chain of approx functions for type *udom* 116

22 Algebraic deflations **118**
22.1 Type constructor for finite deflations 118
22.2 Defining algebraic deflations by ideal completion 120
22.3 Applying algebraic deflations 121
22.4 Deflation combinators 123

23 Representable domains **124**
23.1 Class of representable domains 124
23.2 Domains are bifinite 125
23.3 Universal domain ep-pairs 126
23.4 Type combinators 127
23.5 Class instance proofs 128
23.5.1 Universal domain 128
23.5.2 Lifted cpo 129
23.5.3 Strict function space 130
23.5.4 Continuous function space 131
23.5.5 Strict product 131
23.5.6 Cartesian product 132
23.5.7 Unit type 134
23.5.8 Discrete cpo 135
23.5.9 Strict sum 136
23.5.10 Lifted HOL type 136

24 The unit domain **137**

25 Fixed point operator and admissibility **139**
25.1 Iteration 139
25.2 Least fixed point operator 139
25.3 Fixed point induction 141
25.4 Fixed-points on product types 142

26 Package for defining recursive functions in HOLCF **143**
26.1 Pattern-match monad 143
26.1.1 Run operator 144
26.1.2 Monad plus operator 144
26.2 Match functions for built-in types 145
26.3 Mutual recursion 147
26.4 Initializing the fixrec package 147

27 Domain package **148**
27.1 Continuous isomorphisms 148
27.2 Proofs about take functions 150
27.3 Finiteness 151
27.4 Proofs about constructor functions 153

27.5 ML setup . . . 155
27.6 Representations of types . . . 155
27.7 Deflations as sets . . . 156
27.8 Proving a subtype is representable . . . 156
27.9 Isomorphic deflations . . . 158
27.10 Setting up the domain package . . . 161

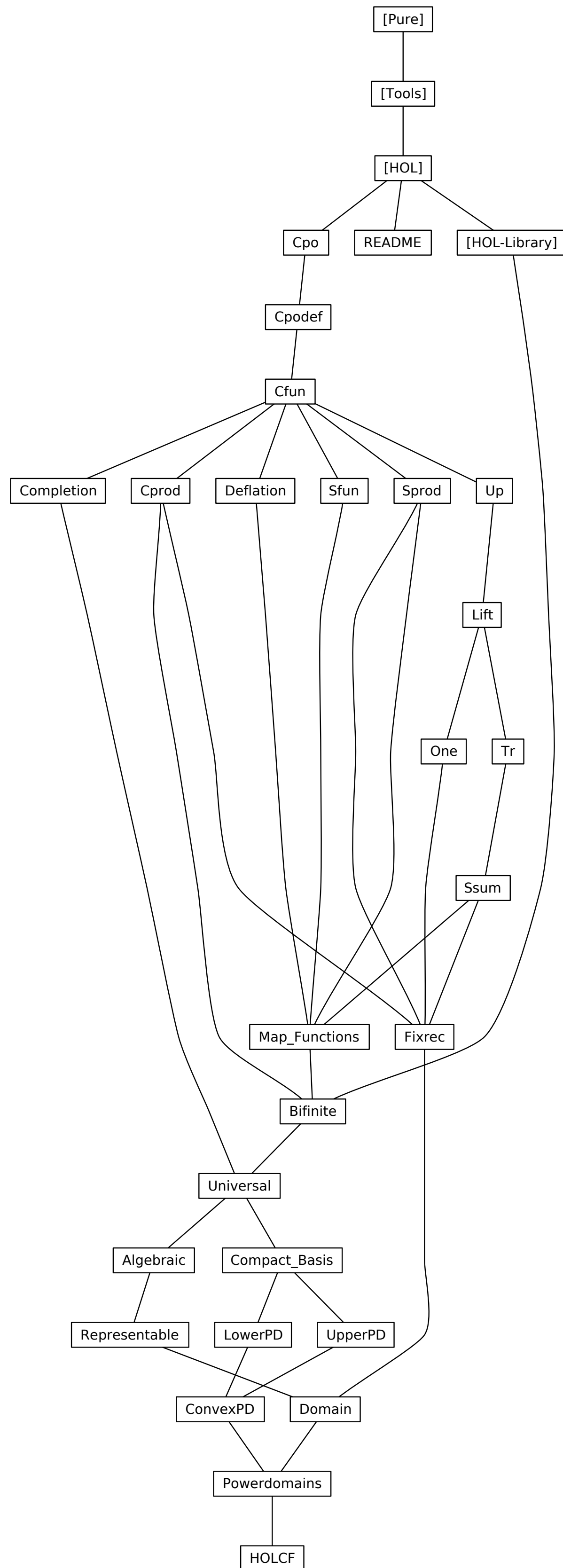
28 A compact basis for powerdomains **162**
28.1 A compact basis for powerdomains . . . 162
28.2 Unit and plus constructors . . . 163
28.3 Fold operator . . . 164

29 Upper powerdomain **164**
29.1 Basis preorder . . . 164
29.2 Type definition . . . 166
29.3 Monadic unit and plus . . . 167
29.4 Induction rules . . . 170
29.5 Monadic bind . . . 171
29.6 Map . . . 172
29.7 Upper powerdomain is bifinite . . . 174
29.8 Join . . . 174

30 Lower powerdomain **175**
30.1 Basis preorder . . . 175
30.2 Type definition . . . 177
30.3 Monadic unit and plus . . . 178
30.4 Induction rules . . . 181
30.5 Monadic bind . . . 182
30.6 Map . . . 183
30.7 Lower powerdomain is bifinite . . . 185
30.8 Join . . . 185

31 Convex powerdomain **186**
31.1 Basis preorder . . . 186
31.2 Type definition . . . 188
31.3 Monadic unit and plus . . . 189
31.4 Induction rules . . . 191
31.5 Monadic bind . . . 192
31.6 Map . . . 194
31.7 Convex powerdomain is bifinite . . . 196
31.8 Join . . . 196
31.9 Conversions to other powerdomains . . . 197

32 Powerdomains **199**
32.1 Universal domain embeddings 199
32.2 Deflation combinators 199
32.3 Domain class instances 200
32.4 Isomorphic deflations 202
32.5 Domain package setup for powerdomains 203


[Pure]
[Tools]
[HOL]
Cpo
README
[HOL-Library]
Cpodef
Cfun
Completion
Cprod
Deflation
Sfun
Sprod
Up
Lift
One
Tr
Ssum
Map_Functions
Fixrec
Bifinite
Universal
Algebraic
Compact_Basis
Representable
LowerPD
UpperPD
ConvexPD
Domain
Powerdomains
HOLCF

```
theory Cpo
  imports Main
begin
```

1 Partial orders

```
declare [[typedef-overloaded]]
```

1.1 Type class for partial orders

```
class below =
  fixes below :: 'a ⇒ 'a ⇒ bool
begin

notation (ASCII)
  below (infix ‹<<› 50)

notation
  below (infix ‹⊑› 50)

abbreviation not-below :: 'a ⇒ 'a ⇒ bool  (infix ‹⋢› 50)
  where not-below x y ≡ ¬ below x y

notation (ASCII)
  not-below  (infix ‹~<<› 50)

lemma below-eq-trans: a ⊑ b ⟹ b = c ⟹ a ⊑ c
  by (rule subst)

lemma eq-below-trans: a = b ⟹ b ⊑ c ⟹ a ⊑ c
  by (rule ssubst)

end

class po = below +
  assumes below-refl [iff]: x ⊑ x
  assumes below-trans: x ⊑ y ⟹ y ⊑ z ⟹ x ⊑ z
  assumes below-antisym: x ⊑ y ⟹ y ⊑ x ⟹ x = y
begin

lemma eq-imp-below: x = y ⟹ x ⊑ y
  by simp

lemma box-below: a ⊑ b ⟹ c ⊑ a ⟹ b ⊑ d ⟹ c ⊑ d
  by (rule below-trans [OF below-trans])

lemma po-eq-conv: x = y ⟷ x ⊑ y ∧ y ⊑ x
```

by (*fast intro*!: *below-antisym*)

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

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

end

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

context *po*
begin

1.2 Upper bounds

definition *is-ub* :: *'a set* ⇒ *'a* ⇒ *bool* (**infix** ‹<|› *55*)
where $S <| x \longleftrightarrow (\forall y \in S.\ y \sqsubseteq x)$

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

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

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

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

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

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

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

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

lemma *is-ub-upward*: $[\![S <| x;\ x \sqsubseteq y]\!] \Longrightarrow S <| y$

unfolding *is-ub-def* **by** (*fast intro*: *below-trans*)

1.3 Least upper bounds

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

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

end

syntax (*ASCII*)
-BLub :: [*pttrn*, *'a set*, *'b*] ⇒ *'b* (‹(‹*indent=3 notation=*‹*binder LUB*››*LUB* -:-./ -)› [*0*,*0*, *10*] *10*)

syntax
-BLub :: [*pttrn*, *'a set*, *'b*] ⇒ *'b* (‹(‹*indent=3 notation=*‹*binder* ⨆››⨆-∈-./ -)› [*0*,*0*, *10*] *10*)

syntax-consts
-BLub ⇌ *lub*

translations
LUB x:*A*. *t* ⇌ *CONST lub* ((λ*x*. *t*) ‘ *A*)

context *po*
begin

abbreviation *Lub* (**binder** ‹⨆› *10*)
where $\bigsqcup n.\ t\ n \equiv lub\ (range\ t)$

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

access to some definition as inference rule

lemma *is-lubD1*: $S <<| x \Longrightarrow S <| x$
unfolding *is-lub-def* **by** *fast*

lemma *is-lubD2*: $[\![S <<| x;\ S <| u]\!] \Longrightarrow x \sqsubseteq u$
unfolding *is-lub-def* **by** *fast*

lemma *is-lubI*: $[\![S <| x;\ \bigwedge u.\ S <| u \Longrightarrow x \sqsubseteq u]\!] \Longrightarrow S <<| x$
unfolding *is-lub-def* **by** *fast*

lemma *is-lub-below-iff*: $S <<| x \Longrightarrow x \sqsubseteq u \longleftrightarrow S <| u$
unfolding *is-lub-def is-ub-def* **by** (*metis below-trans*)

lubs are unique

lemma *is-lub-unique*: $S <<| x \Longrightarrow S <<| y \Longrightarrow x = y$
  unfolding *is-lub-def is-ub-def* **by** (*blast intro*: *below-antisym*)

technical lemmas about *lub* and $(<<|)$

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

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

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

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

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

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

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

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

1.4 Countable chains

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

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

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

chains are monotone functions

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

lemma *chain-mono*: $chain\ Y \Longrightarrow i \le j \Longrightarrow Y\ i \sqsubseteq Y\ j$
  by (*cases* $i = j$) (*simp-all add*: *chain-mono-less*)

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

technical lemmas about (least) upper bounds of chains

lemma *is-lub-rangeD1*: $\textit{range}\ S <<|\ x \Longrightarrow S\ i \sqsubseteq x$
by (*rule is-lubD1* [*THEN ub-rangeD*])

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

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

the lub of a constant chain is the constant

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

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

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

1.5 Finite chains

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

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

results about finite chains

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

lemma *max-in-chainD*: $\textit{max-in-chain}\ i\ Y \Longrightarrow i \leq j \Longrightarrow Y\ i = Y\ j$
unfolding *max-in-chain-def* **by** *fast*

lemma *finite-chainI*: $\textit{chain}\ C \Longrightarrow \textit{max-in-chain}\ i\ C \Longrightarrow \textit{finite-chain}\ C$
unfolding *finite-chain-def* **by** *fast*

lemma *finite-chainE*: $[\![$*finite-chain C*; $\bigwedge i$. $[\![$*chain C*; *max-in-chain i C*$]\!] \Longrightarrow R]\!]$ $\Longrightarrow R$
  unfolding *finite-chain-def* **by** *fast*

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

lemma *finch-imp-finite-range*: *finite-chain Y* $\Longrightarrow$ *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*: $\bigwedge i\ j.\ i \le j \Longrightarrow r\ (f\ i)\ (f\ j)$
  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* ((λ*x*. *LEAST k*. *f k* = *x*) ‘ *range f*)
  have *?j* $\le$ *?k*
  proof (*rule Max-ge*)
    show *finite* ((λ*x*. *LEAST k*. *f k* = *x*) ‘ *range f*)
      using *finite-range* **by** (*rule finite-imageI*)
    show *?j* $\in$ (λ*x*. *LEAST k*. *f k* = *x*) ‘ *range f*
      by (*intro imageI rangeI*)
  qed

```
  hence r (f ?j) (f ?k)
    by (rule mono)
  also have f ?j = f i
    by (rule LeastI, rule refl)
  finally show r (f i) (f ?k) .
qed

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

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

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

lemma is-lub-bin-chain: x ⊑ y ⟹ 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 ⟹ ∀ i. Y i ⊑ c ⟹ lub (range Y) = c
  by (blast dest: ub-rangeD intro: lub-eqI is-lubI ub-rangeI)

end
```

2 Classes cpo and pcpo

2.1 Complete partial orders

The class cpo of chain complete partial orders

```
class cpo = po +
  assumes cpo: chain S ⟹ ∃ x. range S <<| x

default-sort cpo
```

context *cpo*
begin

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

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

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

Properties of the lub

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

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

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

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

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

lemma *lub-range-mono*: $[\![range\ X \subseteq range\ Y;\ chain\ Y;\ chain\ X]\!] \Longrightarrow (\bigsqcup i.\ X\ i) \sqsubseteq (\bigsqcup i.\ Y\ i)$
  apply (*erule lub-below*)
  apply (*subgoal-tac* $\exists j.\ X\ i = Y\ j$)
   apply *clarsimp*
   apply (*erule is-ub-thelub*)
  apply *auto*
  done

lemma *lub-range-shift*: $chain\ Y \Longrightarrow (\bigsqcup i.\ Y\ (i + j)) = (\bigsqcup i.\ Y\ i)$
  apply (*rule below-antisym*)
   apply (*rule lub-range-mono*)
     apply *fast*
    apply *assumption*
   apply (*erule chain-shift*)
  apply (*rule lub-below*)
   apply *assumption*
  apply (*rule-tac* $i=i$ **in** *below-lub*)
   apply (*erule chain-shift*)
  apply (*erule chain-mono*)
  apply (*rule le-add1*)
  done

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

the ⊑ relation between two chains is preserved by their lubs

```
lemma lub-mono: ⟦chain X; chain Y; ⋀i. X i ⊑ Y i⟧ ⟹ (⨆i. X i) ⊑ (⨆i. Y i)
  by (fast elim: lub-below below-lub)
```

the = relation between two chains is preserved by their lubs

```
lemma lub-eq: (⋀i. X i = Y i) ⟹ (⨆i. X i) = (⨆i. Y i)
  by simp

lemma ch2ch-lub:
  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:
  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 below-antisym)
  have 3: chain (λi. Y i i)
    apply (rule chainI)
    apply (rule below-trans)
     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 (⨆i. ⨆j. Y i j) ⊑ (⨆i. Y i i)
    apply (rule lub-below [OF 4])
    apply (rule lub-below [OF 2])
    apply (rule below-lub [OF 3])
    apply (rule below-trans)
     apply (rule chain-mono [OF 1 max.cobounded1])
    apply (rule chain-mono [OF 2 max.cobounded2])
    done
```

```
  show (⨆i. Y i i) ⊑ (⨆i. ⨆j. Y i j)
    apply (rule lub-mono [OF 3 4])
    apply (rule is-ub-thelub [OF 2])
    done
qed

lemma ex-lub:
  assumes 1: ⋀j. chain (λi. Y i j)
  assumes 2: ⋀i. chain (λj. Y i j)
  shows (⨆i. ⨆j. Y i j) = (⨆j. ⨆i. Y i j)
  by (simp add: diag-lub 1 2)

end
```

2.2 Pointed cpos

The class pcpo of pointed cpos

```
class pcpo = cpo +
  assumes least: ∃x. ∀y. x ⊑ y
begin

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

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

end
```

Old "UU" syntax:

```
abbreviation (input) UU ≡ bottom
```

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

```
setup ‹Reorient-Proc.add (fn Const- ‹bottom -› => true | - => false)›
simproc-setup reorient-bottom (⊥ = x) = ‹K Reorient-Proc.proc›
```

useful lemmas about $\bot$

```
lemma below-bottom-iff [simp]: x ⊑ ⊥ ⟷ x = ⊥
  by (simp add: po-eq-conv)

lemma eq-bottom-iff: x = ⊥ ⟷ x ⊑ ⊥
  by simp
```

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

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

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

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

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

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

end

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

subclass *chfin*
proof
  fix Y
  assume $*$: *chain* Y
  show $\exists n.$ *max-in-chain* $n\ Y$
    apply (*unfold max-in-chain-def*)
    apply (*cases* $\forall i.\ Y\ i = \bot$)
     apply *simp*
    apply *simp*
    apply (*erule exE*)
    apply (*rule-tac x=i* **in** *exI*)
    apply *clarify*
    using $*$ **apply** (*blast dest*: *chain-mono ax-flat*)
    done
qed

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

lemma *flat-eq*: $a \neq \bot \Longrightarrow a \sqsubseteq b = (a = b)$

by (*safe dest*!: *ax-flat*)

end

2.4 Discrete cpos

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

subclass *po*
by *standard simp-all*

In a discrete cpo, every chain is constant

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

subclass *chfin*
proof
fix *S* :: *nat* ⇒ *'a*
assume *S*: *chain S*
then have $\exists x.\ S = (\lambda i.\ x)$
by (*rule discrete-chain-const*)
then have *max-in-chain 0 S*
by (*auto simp*: *max-in-chain-def*)
then show $\exists i.$ *max-in-chain i S* **..**
qed

end

3 Continuity and monotonicity

3.1 Definitions

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

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

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

lemma *contE*: $cont\ f \Longrightarrow chain\ Y \Longrightarrow range\ (\lambda i.\ f\ (Y\ i)) <<|\ f\ (\bigsqcup i.\ Y\ i)$
  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*: $monofun\ f \Longrightarrow x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
  by (*simp add*: *monofun-def*)

3.2 Equivalence of alternate definition

monotone functions map chains to chains

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

monotone functions map upper bound to upper bounds

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

a lemma about binary chains

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

continuity implies monotonicity

lemma *cont2mono*: $cont\ f \Longrightarrow monofun\ f$
  apply (*rule monofunI*)
  apply (*drule* (*1*) *binchain-cont*)
  apply (*drule-tac i=0* **in** *is-lub-rangeD1*)
  apply *simp*
  done

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

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

continuity implies preservation of lubs

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

lemma *contI2*:
  fixes f :: $'a \Rightarrow 'b$
  assumes *mono*: *monofun f*
  assumes *below*: $\bigwedge Y.\ [\![chain\ Y;\ chain\ (\lambda i.\ f\ (Y\ i))]\!] \Longrightarrow f\ (\bigsqcup i.\ Y\ i) \sqsubseteq (\bigsqcup i.\ f$
$(Y\ i))$
  shows *cont f*
proof (*rule contI*)
  fix Y :: $nat \Rightarrow 'a$
  assume Y: *chain Y*
  with *mono* **have** *fY*: *chain* ($\lambda i.\ f\ (Y\ i)$)
    by (*rule ch2ch-monofun*)
  have ($\bigsqcup i.\ f\ (Y\ i)$) = f ($\bigsqcup i.\ Y\ i$)
    apply (*rule below-antisym*)
     apply (*rule lub-below* [*OF fY*])
     apply (*rule monofunE* [*OF mono*])
     apply (*rule is-ub-thelub* [*OF Y*])
    apply (*rule below* [*OF Y fY*])
    done
  with *fY* **show** *range* ($\lambda i.\ f\ (Y\ i)$) <<| f ($\bigsqcup i.\ Y\ i$)
    by (*rule thelubE*)
qed

3.3 Collection of continuity rules

named-theorems *cont2cont continuity intro rule*

3.4 Continuity of basic functions

The identity function is continuous

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

constant functions are continuous

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

application of functions is continuous

lemma *cont-apply*:

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

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

Least upper bounds preserve continuity

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

if-then-else is continuous

lemma *cont-if* [*simp*, *cont2cont*]: *cont f* $\Longrightarrow$ *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* $\Longrightarrow$ *finite-chain Y* $\Longrightarrow$ *finite-chain* $(\lambda n.\ f\ (Y\ n))$
by (*force simp add*: *finite-chain-def ch2ch-monofun max-in-chain-def*)

The same holds for continuous functions.

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

All monotone functions with chain-finite domain are continuous.

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

All strict functions with flat domain are continuous.

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

All functions with discrete domain are continuous.

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

4 Admissibility and compactness

4.1 Definitions

context *cpo*
begin

definition *adm* :: (*'a* $\Rightarrow$ *bool*) $\Rightarrow$ *bool*
where *adm P* $\longleftrightarrow$ ($\forall Y.$ *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*: $adm\ P \Longrightarrow chain\ Y \Longrightarrow (\bigwedge i.\ P\ (Y\ i)) \Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
unfolding *adm-def* **by** *fast*

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

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

end

4.2 Admissibility on chain-finite types

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

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

4.3 Admissibility of special formulae and propagation

context *cpo*
begin

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

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

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

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

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

lemma *adm-disj-lemma1*:
assumes *adm*: *adm P*
assumes *chain*: *chain Y*
assumes *P*: $\forall\, i.\ \exists\, j{\geq}i.\ P\ (Y\ j)$
shows $P\ (\bigsqcup i.\ Y\ i)$
proof −
define *f* **where** $f\ i = (LEAST\ j.\ i \leq j \wedge P\ (Y\ j))$ **for** *i*
have *chain'*: $chain\ (\lambda i.\ Y\ (f\ i))$
unfolding *f-def*
apply (*rule chainI*)
apply (*rule chain-mono* [*OF chain*])
apply (*rule Least-le*)

apply (*rule LeastI2-ex*)
apply (*simp-all add*: *P*)
done
have *f1*: $\bigwedge i.\ i \leq f\ i$ **and** *f2*: $\bigwedge i.\ P\ (Y\ (f\ i))$
using *LeastI-ex* [*OF P* [*rule-format*]] **by** (*simp-all add*: *f-def*)
have *lub-eq*: $(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ Y\ (f\ i))$
apply (*rule below-antisym*)
apply (*rule lub-mono* [*OF chain chain'*])
apply (*rule chain-mono* [*OF chain f1*])
apply (*rule lub-range-mono* [*OF - chain chain'*])
apply *clarsimp*
done
show $P\ (\bigsqcup i.\ Y\ i)$
unfolding *lub-eq* **using** *adm chain' f2* **by** (*rule admD*)
qed

lemma *adm-disj-lemma2*: $\forall n::nat.\ P\ n \vee Q\ n \Longrightarrow (\forall i.\ \exists j \geq i.\ P\ j) \vee (\forall i.\ \exists j \geq i.\ Q\ j)$
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* [*simp*]: $adm\ (\lambda x.\ P\ x) \Longrightarrow adm\ (\lambda x.\ Q\ x) \Longrightarrow adm\ (\lambda x.\ P\ x \vee Q\ x)$
apply (*rule admI*)
apply (*erule adm-disj-lemma2* [*THEN disjE*])
apply (*erule* (*2*) *adm-disj-lemma1* [*THEN disjI1*])
apply (*erule* (*2*) *adm-disj-lemma1* [*THEN disjI2*])
done

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

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

end

admissibility and continuity

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

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

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

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

4.4 Compactness

context *cpo*
begin

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

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

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

lemma *compactI2*: $(\bigwedge Y.\ [\![chain\ Y;\ x \sqsubseteq (\bigsqcup i.\ Y\ i)]\!] \Longrightarrow \exists i.\ x \sqsubseteq Y\ i) \Longrightarrow$ *compact* x
unfolding *compact-def adm-def* **by** *fast*

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

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

end

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

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

admissibility and compactness

lemma *adm-compact-not-below* [*simp*]:

compact k $\Longrightarrow$ *cont* ($\lambda x.$ *t x*) $\Longrightarrow$ *adm* ($\lambda x.$ *k* $\not\sqsubseteq$ *t x*)
unfolding *compact-def* **by** (*rule adm-subst*)

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

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

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

Any upward-closed predicate is admissible.

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

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

5 Class instances for the full function space

5.1 Full function space is a partial order

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

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

instance ..
end

instance *fun* :: (*type*, *po*) *po*
proof
fix $f\ g\ h$ :: $'a \Rightarrow\ 'b$
show $f \sqsubseteq f$
by (*simp add*: *below-fun-def*)
show $f \sqsubseteq g \Longrightarrow g \sqsubseteq f \Longrightarrow f = g$
by (*simp add*: *below-fun-def fun-eq-iff below-antisym*)
show $f \sqsubseteq g \Longrightarrow g \sqsubseteq h \Longrightarrow f \sqsubseteq h$
unfolding *below-fun-def* **by** (*fast elim*: *below-trans*)
qed

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

by (*simp add*: *below-fun-def*)

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

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

5.2 Full function space is chain complete

Properties of chains of functions.

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

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

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

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

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

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

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

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

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

5.3 Full function space is pointed

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

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

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

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

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

5.4 Propagation of monotonicity and continuity

The lub of a chain of monotone functions is monotone.

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

The lub of a chain of continuous functions is continuous.

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

Function application preserves monotonicity and continuity.

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

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

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

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

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

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

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

lemma *contlub-lambda*: $(\bigwedge x.\ chain\ (\lambda i.\ S\ i\ x)) \Longrightarrow (\lambda x.\ \bigsqcup i.\ S\ i\ x) = (\bigsqcup i.\ (\lambda x.\ S\ i\ x))$
  for $S :: nat \Rightarrow 'a::type \Rightarrow 'b$
  by (*simp add*: *lub-fun ch2ch-lambda*)

6 The cpo of cartesian products

6.1 Unit type is a pcpo

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

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

instance
  by *standard simp*

end

instance *unit* :: *pcpo*
  by *standard simp*

6.2 Product type is a partial order

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

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

instance ..

end

instance *prod* :: (*po*, *po*) *po*
proof
  fix $x\ y\ z :: 'a \times 'b$
  show $x \sqsubseteq x$
    by (*simp add*: *below-prod-def*)
  show $x \sqsubseteq y \Longrightarrow y \sqsubseteq x \Longrightarrow x = y$
    unfolding *below-prod-def prod-eq-iff*
    by (*fast intro*: *below-antisym*)
  show $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
    unfolding *below-prod-def*
    by (*fast intro*: *below-trans*)
qed

6.3 Monotonicity of *Pair*, *fst*, *snd*

lemma *prod-belowI*: $fst\ p \sqsubseteq fst\ q \Longrightarrow snd\ p \sqsubseteq snd\ q \Longrightarrow p \sqsubseteq q$

by (*simp add*: *below-prod-def*)

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

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*: $x1 \sqsubseteq x2 \Longrightarrow y1 \sqsubseteq y2 \Longrightarrow (x1, y1) \sqsubseteq (x2, y2)$
by *simp*

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

fst and *snd* are monotone

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

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

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

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

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

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

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

6.4 Product type is a cpo

lemma *is-lub-Pair*: $\mathit{range}\ A <<|\ x \Longrightarrow \mathit{range}\ B <<|\ y \Longrightarrow \mathit{range}\ (\lambda i.\ (A\ i,\ B\ i))$ $<<|\ (x,\ y)$
  by (*simp add*: *is-lub-def is-ub-def below-prod-def*)

lemma *lub-Pair*: $\mathit{chain}\ A \Longrightarrow \mathit{chain}\ B \Longrightarrow (\bigsqcup i.\ (A\ i,\ B\ i)) = (\bigsqcup i.\ A\ i,\ \bigsqcup i.\ B\ i)$
  for $A :: \mathit{nat} \Rightarrow {}'a$ **and** $B :: \mathit{nat} \Rightarrow {}'b$
  by (*fast intro*: *lub-eqI is-lub-Pair elim*: *thelubE*)

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

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

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

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

6.5 Product type is pointed

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

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

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

lemma *Pair-bottom-iff* [*simp*]: $(x,\ y) = \bot \longleftrightarrow x = \bot \wedge y = \bot$
  by (*simp add*: *inst-prod-pcpo*)

lemma *fst-strict* [*simp*]: $\mathit{fst}\ \bot = \bot$

unfolding *inst-prod-pcpo* **by** (*rule fst-conv*)

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

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

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

6.6 Continuity of *Pair*, *fst*, *snd*

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

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

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

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

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

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

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

lemma cont2cont-case-prod:
  assumes f1: ⋀a b. cont (λx. f x a b)
  assumes f2: ⋀x b. cont (λa. f x a b)
  assumes f3: ⋀x a. cont (λb. f x a b)
  assumes g: cont (λx. g x)
  shows cont (λx. case g x of (a, b) ⇒ f x a b)
  unfolding split-def
  apply (rule cont-apply [OF g])
   apply (rule cont-apply [OF cont-fst f2])
   apply (rule cont-apply [OF cont-snd f3])
   apply (rule cont-const)
  apply (rule f1)
  done

lemma prod-contI:
  assumes f1: ⋀y. cont (λx. f (x, y))
  assumes f2: ⋀x. cont (λy. f (x, y))
  shows cont f
proof –
  have cont (λ(x, y). f (x, y))
    by (intro cont2cont-case-prod f1 f2 cont2cont)
  then show cont f
    by (simp only: case-prod-eta)
qed

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

lemma cont2cont-case-prod′ [simp, cont2cont]:
  assumes f: cont (λp. f (fst p) (fst (snd p)) (snd (snd p)))
  assumes g: cont (λx. g x)
  shows cont (λx. case-prod (f x) (g x))
  using assms by (simp add: cont2cont-case-prod prod-cont-iff)
```

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

```
lemma cont2cont-split-simple [simp, cont2cont]:
  assumes ⋀a b. cont (λx. f x a b)
  shows cont (λx. case p of (a, b) ⇒ f x a b)
  using assms by (cases p) auto
```

Admissibility of predicates on product types.

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

6.7 Compactness and chain-finiteness

lemma *fst-below-iff*: *fst x* ⊑ *y* ⟷ *x* ⊑ (*y*, *snd x*) **for** *x* :: *'a* × *'b*
by (*simp add*: *below-prod-def*)

lemma *snd-below-iff*: *snd x* ⊑ *y* ⟷ *x* ⊑ (*fst x*, *y*) **for** *x* :: *'a* × *'b*
by (*simp add*: *below-prod-def*)

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

lemma *compact-snd*: *compact x* ⟹ *compact* (*snd x*)
by (*rule compactI*) (*simp add*: *snd-below-iff*)

lemma *compact-Pair*: *compact x* ⟹ *compact y* ⟹ *compact* (*x*, *y*)
by (*rule compactI*) (*simp add*: *below-prod-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 *prod* :: (*chfin*, *chfin*) *chfin*
apply *intro-classes*
apply (*erule compact-imp-max-in-chain*)
apply (*case-tac* ⨆*i*. *Y i*, *simp*)
done

7 Discrete cpo types

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

7.1 Discrete cpo class instance

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

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

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

end

7.2 *undiscr*

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

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

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

end

8 Subtypes of pcpos

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

8.1 Proving a subtype is a partial order

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

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

lemmas *typedef-po-class* = *below.typedef-class-po* [*THEN po.intro-of-class*]

8.2 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
  then have finite (Abs ‘ A)
    by (rule finite-imageI)
  then show finite (UNIV :: 'b set)
    by (simp only: type-definition.Abs-image [OF type])
qed
```

8.3 Proving a subtype is chain-finite

```
lemma ch2ch-Rep:
  assumes below: (⊑) ≡ λx y. Rep x ⊑ Rep y
  shows chain S ⟹ chain (λi. Rep (S i))
  unfolding chain-def below .

theorem typedef-chfin:
  fixes Abs :: 'a::chfin ⇒ 'b::po
  assumes type: type-definition Rep Abs A
    and below: (⊑) ≡ λx y. Rep x ⊑ Rep y
  shows OFCLASS('b, chfin-class)
  apply intro-classes
  apply (drule ch2ch-Rep [OF below])
  apply (drule chfin)
  apply (unfold max-in-chain-def)
  apply (simp add: type-definition.Rep-inject [OF type])
  done
```

8.4 Proving a subtype is complete

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

```
lemma typedef-is-lubI:
  assumes below: (⊑) ≡ λx y. Rep x ⊑ Rep y
  shows range (λi. Rep (S i)) <<| Rep x ⟹ range S <<| x
  by (simp add: is-lub-def is-ub-def below)

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

theorem typedef-is-lub:
  fixes Abs :: 'a::cpo ⇒ 'b::po
```

assumes *type*: *type-definition Rep Abs A*
and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
assumes *S*: *chain S*
shows $range\ S <<|\ Abs\ (\bigsqcup i.\ Rep\ (S\ i))$
proof −
from *S* **have** $chain\ (\lambda i.\ Rep\ (S\ i))$
by (*rule ch2ch-Rep* [*OF below*])
then have $range\ (\lambda i.\ Rep\ (S\ i)) <<|\ (\bigsqcup i.\ Rep\ (S\ i))$
by (*rule cpo-lubI*)
then have $range\ (\lambda i.\ Rep\ (S\ i)) <<|\ Rep\ (Abs\ (\bigsqcup i.\ Rep\ (S\ i)))$
by (*simp only*: *Abs-inverse-lub-Rep* [*OF type below adm S*])
then show $range\ S <<|\ Abs\ (\bigsqcup i.\ Rep\ (S\ i))$
by (*rule typedef-is-lubI* [*OF below*])
qed

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

theorem *typedef-cpo*:
fixes *Abs* :: *'a::cpo* ⇒ *'b::po*
assumes *type*: *type-definition Rep Abs A*
and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
shows *OFCLASS*(*'b*, *cpo-class*)
proof
fix *S* :: *nat* ⇒ *'b*
assume *chain S*
then have $range\ S <<|\ Abs\ (\bigsqcup i.\ Rep\ (S\ i))$
by (*rule typedef-is-lub* [*OF type below adm*])
then show $\exists x.\ range\ S <<|\ x$ **..**
qed

8.4.1 Continuity of *Rep* and *Abs*

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

theorem *typedef-cont-Rep*:
fixes *Abs* :: *'a::cpo* ⇒ *'b::cpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *adm*: $adm\ (\lambda x.\ x \in A)$
shows $cont\ (\lambda x.\ f\ x) \Longrightarrow cont\ (\lambda x.\ Rep\ (f\ x))$
apply (*erule cont-apply* [*OF - - cont-const*])
apply (*rule contI*)
apply (*simp only*: *typedef-lub* [*OF type below adm*])
apply (*simp only*: *Abs-inverse-lub-Rep* [*OF type below adm*])
apply (*rule cpo-lubI*)
apply (*erule ch2ch-Rep* [*OF below*])
done

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

theorem *typedef-cont-Abs*:
  fixes *Abs* :: *'a::cpo* ⇒ *'b::cpo*
  fixes *f* :: *'c::cpo* ⇒ *'a::cpo*
  assumes *type*: *type-definition Rep Abs A*
    and *below*: $(\sqsubseteq) \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
    and *adm*: *adm* $(\lambda x.\ x \in A)$
    and *f-in-A*: $\bigwedge x.\ f\ x \in A$
  shows *cont f* $\Longrightarrow$ *cont* $(\lambda x.\ Abs\ (f\ x))$
  unfolding *cont-def is-lub-def is-ub-def ball-simps below*
  by (*simp add*: *type-definition.Abs-inverse* [*OF type f-in-A*])

8.5 Proving subtype elements are compact

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

8.6 Proving a subtype is pointed

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

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

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

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

8.6.1 Strictness of *Rep* and *Abs*

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

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

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

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

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

done

8.7 Proving a subtype is flat

theorem *typedef-flat*:
fixes *Abs* :: *'a::flat ⇒ 'b::pcpo*
assumes *type*: *type-definition Rep Abs A*
and *below*: (⊑) ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *bottom-in-A*: ⊥ ∈ *A*
shows *OFCLASS*(*'b*, *flat-class*)
apply (*intro-classes*)
apply (*unfold below*)
apply (*simp add*: *type-definition.Rep-inject* [*OF type*, *symmetric*])
apply (*simp add*: *typedef-Rep-strict* [*OF type below bottom-in-A*])
apply (*simp add*: *ax-flat*)
done

8.8 HOLCF type definition package

ML-file ‹*Tools/cpodef.ML*›

end

9 The type of continuous functions

theory *Cfun*
imports *Cpodef*
begin

9.1 Definition of continuous function type

definition *cfun* = {*f*::*'a* ⇒ *'b*. *cont f*}

cpodef (*'a*, *'b*) *cfun* (‹(‹*notation*=‹*infix* →››- →/ -)› [*1*, *0*] *0*) = *cfun* :: (*'a* ⇒ *'b*) *set*
by (*auto simp*: *cfun-def intro*: *cont-const adm-cont*)

type-notation (*ASCII*)
cfun (**infixr** ‹−>› *0*)

notation (*ASCII*)
Rep-cfun (‹(‹*notation*=‹*infix* $››-$/-)› [*999*,*1000*] *999*)

notation
Rep-cfun (‹(‹*notation*=‹*infix* ·››-·/-)› [*999*,*1000*] *999*)

9.2 Syntax for continuous lambda abstraction

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

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

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

Syntax for nested abstractions

```
syntax (ASCII)
  -Lambda :: [cargs, logic] ⇒ logic  (‹(‹indent=3 notation=‹binder LAM››LAM -./
-)› [1000, 10] 10)

syntax
  -Lambda :: [cargs, logic] ⇒ logic (‹(‹indent=3 notation=‹binder Λ››Λ -./ -)›
[1000, 10] 10)

syntax-consts
  -Lambda ⇌ Abs-cfun

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

print-ast-translation ‹
(* rewrite (-cabs x (-cabs y (-cabs z t))) => (LAM x y z. t) *)
(* cf. Syntax.abs-ast-tr′ from src/Pure/Syntax/syn-trans.ML *)
  let
    fun cabs-ast-tr′ asts =
      (case Ast.unfold-ast-p syntax-const ‹-cabs›
          (Ast.Appl (Ast.Constant syntax-const ‹-cabs› :: asts)) of
        ([], -) => raise Ast.AST (cabs-ast-tr′, asts)
      | (xs, body) => Ast.Appl
          [Ast.Constant syntax-const ‹-Lambda›,
           Ast.fold-ast syntax-const ‹-cargs› xs, body]);
  in [(syntax-const ‹-cabs›, K cabs-ast-tr′)] end
›
```

Dummy patterns for continuous abstraction

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

9.3 Continuous function space is pointed

lemma *bottom-cfun*: $\bot \in$ *cfun*
by (*simp add*: *cfun-def inst-fun-pcpo*)

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

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

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

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

function application is strict in its first argument

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

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

for compatibility with old HOLCF-Version

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

9.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

9.4.1 Beta-reduction simproc

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

The simproc does not solve any more goals that would be solved by using *beta-cfun* as a simp rule. The advantage of the simproc is that it can avoid

deeply-nested calls to the simplifier that would otherwise be caused by large continuity side conditions.

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

simproc-setup *beta-cfun-proc* (*Rep-cfun* (*Abs-cfun f*)) = ‹
  K (*fn ctxt* => *fn ct* =>
    let
      val f = *Thm.dest-arg* (*Thm.dest-arg ct*);
      val [*T*, *U*] = *Thm.dest-ctyp* (*Thm.ctyp-of-cterm f*);
     val tr = *Thm.instantiate′* [*SOME T*, *SOME U*] [*SOME f*] (*mk-meta-eq* @{*thm Abs-cfun-inverse2*});
      val rules = *Named-Theorems.get ctxt* ***named-theorems*** ‹*cont2cont*›;
      val tac = *SOLVED′* (*REPEAT-ALL-NEW* (*match-tac ctxt* (*rev rules*)));
    in SOME (*perhaps* (*SINGLE* (*tac 1*)) *tr*) *end*)
›

Eta-equality for continuous functions

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

Extensionality for continuous functions

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

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

Extensionality wrt. ordering for continuous functions

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

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

Congruence for continuous function application

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

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

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

9.5 Continuity of application

lemma *cont-Rep-cfun1*: *cont* $(\lambda f.\ f\cdot x)$

by (*rule cont-Rep-cfun* [*OF cont-id*, *THEN cont2cont-fun*])

lemma *cont-Rep-cfun2*: *cont* ($\lambda x.\ f\cdot x$)
using *Rep-cfun* [**where** $x = f$] **by** (*simp add*: *cfun-def*)

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

lemmas *monofun-Rep-cfun1* = *cont-Rep-cfun1* [*THEN cont2mono*]
lemmas *monofun-Rep-cfun2* = *cont-Rep-cfun2* [*THEN cont2mono*]

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

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

lemma *contlub-cfun-fun*: *chain* $F \Longrightarrow (\bigsqcup i.\ F\ i)\cdot x = (\bigsqcup i.\ F\ i\cdot x)$
by (*rule cont-Rep-cfun1* [*THEN cont2contlubE*])

monotonicity of application

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

lemma *monofun-cfun-arg*: $x \sqsubseteq y \Longrightarrow f\cdot x \sqsubseteq f\cdot y$
by (*rule monofun-Rep-cfun2* [*THEN monofunE*])

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

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

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

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

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

lemma *ch2ch-Rep-cfun* [*simp*]: *chain* $F \Longrightarrow$ *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* ($\lambda i.\ S\ i\ x$)) $\Longrightarrow$ ($\bigwedge i.$ *cont* ($\lambda x.\ S\ i\ x$)) $\Longrightarrow$ *chain* ($\lambda i.\ \Lambda\ x.\ S\ i\ x$)
by (*simp add*: *chain-def cfun-below-iff*)

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

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

lemma *lub-LAM*:
  assumes $\bigwedge x.\ chain\ (\lambda i.\ F\ i\ x)$
    and $\bigwedge i.\ cont\ (\lambda x.\ F\ i\ x)$
  shows $(\bigsqcup i.\ \Lambda\ x.\ F\ i\ x) = (\Lambda\ x.\ \bigsqcup i.\ F\ i\ x)$
  using *assms* **by** (*simp add*: *lub-cfun lub-fun ch2ch-lambda*)

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

strictness

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

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

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

9.6 Continuity simplification procedure

cont2cont lemma for *Rep-cfun*

lemma *cont2cont-APP* [*simp*, *cont2cont*]:
  assumes *f*: $cont\ (\lambda x.\ f\ x)$
  assumes *t*: $cont\ (\lambda x.\ t\ x)$
  shows $cont\ (\lambda x.\ (f\ x) \cdot (t\ x))$
proof −
  from *cont-Rep-cfun1 f* **have** $cont\ (\lambda x.\ (f\ x) \cdot y)$ **for** *y*
    by (*rule cont-compose*)
  with *t cont-Rep-cfun2* **show** $cont\ (\lambda x.\ (f\ x) \cdot (t\ x))$
    by (*rule cont-apply*)
qed

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

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

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

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

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

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: ⋀x. cont (λy. f x y)
  assumes f2: ⋀y. cont (λx. f x y)
  shows cont (λx. Λ y. f x y)
proof (rule cont-Abs-cfun)
  from f1 show f x ∈ cfun for x
    by (simp add: cfun-def)
  from f2 show cont f
    by (rule cont2cont-lambda)
qed
```

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

```
lemma cont2cont-LAM' [simp, cont2cont]:
  fixes f :: 'a::cpo ⇒ 'b::cpo ⇒ 'c::cpo
  assumes f: cont (λp. f (fst p) (snd p))
  shows cont (λx. Λ y. f x y)
  using assms by (simp add: cont2cont-LAM prod-cont-iff)

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

9.7 Miscellaneous

Monotonicity of *Abs-cfun*

```
lemma monofun-LAM: cont f ⟹ cont g ⟹ (⋀x. f x ⊑ g x) ⟹ (Λ x. f x) ⊑
(Λ x. g x)
  by (simp add: cfun-below-iff)
```

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

```
lemma chfin-Rep-cfunR: chain Y ⟹ ∀ s. ∃ n. (LUB i. Y i)·s = Y n·s
  for Y :: nat ⇒ 'a::cpo → 'b::chfin
  apply (rule allI)
  apply (subst contlub-cfun-fun)
   apply assumption
  apply (fast intro!: lub-eqI chfin lub-finch2 chfin2finch ch2ch-Rep-cfunL)
  done

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

9.8 Continuous injection-retraction pairs

Continuous retractions are strict.

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

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

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

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

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

a result about functions with flat codomain

lemma *flat-eqI*: $x \sqsubseteq y \Longrightarrow x \neq \bot \Longrightarrow x = y$
  for *x y* :: *'a*::*flat*
  by (*drule ax-flat*) *simp*

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

9.9 Identity and composition

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

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

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

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

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

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

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

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

Show that interpretation of (pcpo, -→-) is a category.

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

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

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

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

9.10 Strictified functions

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

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

lemma *seq-conv-if*: *seq·x* = (*if x* = ⊥ *then* ⊥ *else ID*)
  by (*simp add*: *seq-def*)

lemma *seq-simps* [*simp*]:
  seq·⊥ = ⊥
  seq·x·⊥ = ⊥
  x ≠ ⊥ ⟹ *seq·x* = *ID*
  by (*simp-all add*: *seq-conv-if*)

definition *strictify* :: (*'a*::*pcpo* → *'b*::*pcpo*) → *'a* → *'b*
  where *strictify* = (Λ *f x*. *seq·x·*(*f·x*))

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

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

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

9.11 Continuity of let-bindings

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

lemma *cont2cont-Let'* [*simp*, *cont2cont*]:
  assumes *f*: *cont* ($\lambda x.$ *f x*)
  assumes *g*: *cont* ($\lambda p.$ *g* (*fst p*) (*snd p*))
  shows *cont* ($\lambda x.$ *let y* = *f x in g x y*)
  using *f*
proof (*rule cont2cont-Let*)
  from *g* **show** *cont* ($\lambda y.$ *g x y*) **for** *x*

```
    by (simp add: prod-cont-iff)
  from g show cont (λx. g x y) for y
    by (simp add: prod-cont-iff)
qed
```

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

```
lemma cont2cont-Let-simple [simp, cont2cont]:
  assumes ⋀y. cont (λx. g x y)
  shows cont (λx. let y = t in g x y)
  unfolding Let-def using assms .

end
```

10 Continuous deflations and ep-pairs

```
theory Deflation
  imports Cfun
begin
```

10.1 Continuous deflations

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

lemma below-ID: d ⊑ ID
  by (rule cfun-belowI) (simp add: below)
```

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

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

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

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

```
lemma belowI:
  assumes f: ⋀x. d·x = x ⟹ f·x = x
  shows d ⊑ f
proof (rule cfun-belowI)
  fix x
  from below have f·(d·x) ⊑ f·x
    by (rule monofun-cfun-arg)
  also from idem have f·(d·x) = d·x
```

```
    by (rule f)
  finally show d·x ⊑ f·x .
qed

lemma belowD: ⟦f ⊑ d; f·x = x⟧ ⟹ d·x = x
proof (rule below-antisym)
  from below show d·x ⊑ x .
  assume f ⊑ d
  then have f·x ⊑ d·x by (rule monofun-cfun-fun)
  also assume f·x = x
  finally show x ⊑ d·x .
qed

end

lemma deflation-strict: deflation d ⟹ d·⊥ = ⊥
  by (rule deflation.below [THEN bottomI])

lemma adm-deflation: adm (λd. deflation d)
  by (simp add: deflation-def)

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

lemma deflation-bottom: deflation ⊥
  by (simp add: deflation.intro)

lemma deflation-below-iff: deflation p ⟹ deflation q ⟹ p ⊑ q ⟷ (∀x. p·x =
x ⟶ q·x = x)
  apply safe
   apply (simp add: deflation.belowD)
  apply (simp add: deflation.belowI)
  done
```

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

```
lemma deflation-below-comp1:
  assumes deflation f
  assumes deflation g
  shows f ⊑ g ⟹ f·(g·x) = f·x
proof (rule below-antisym)
  interpret g: deflation g by fact
  from g.below show f·(g·x) ⊑ f·x by (rule monofun-cfun-arg)
next
  interpret f: deflation f by fact
  assume f ⊑ g
  then have f·x ⊑ g·x by (rule monofun-cfun-fun)
  then have f·(f·x) ⊑ f·(g·x) by (rule monofun-cfun-arg)
  also have f·(f·x) = f·x by (rule f.idem)
```

finally show $f{\cdot}x \sqsubseteq f{\cdot}(g{\cdot}x)$ **.**
qed

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

10.2 Deflations with finite range

lemma *finite-range-imp-finite-fixes*:
assumes *finite* (*range f*)
shows *finite* $\{x.\ f\ x = x\}$
proof −
have $\{x.\ f\ x = x\} \subseteq$ *range f*
by (*clarify*, *erule subst*, *rule rangeI*)
from *this assms* **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*)
from Y **show** *chain* $(\lambda i.\ d{\cdot}(Y\ i))$ **by** *simp*
have *range* $(\lambda i.\ d{\cdot}(Y\ i)) \subseteq$ *range* $(\lambda x.\ d{\cdot}x)$ **by** *auto*
then show *finite* (*range* $(\lambda i.\ d{\cdot}(Y\ i))$)
using *finite-range* **by** (*rule finite-subset*)
qed
then have $\exists j.\ (\bigsqcup i.\ d{\cdot}(Y\ i)) = d{\cdot}(Y\ j)$
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 $d{\cdot}x \sqsubseteq (\bigsqcup i.\ Y\ i)$
then have $d{\cdot}(d{\cdot}x) \sqsubseteq d{\cdot}(\bigsqcup i.\ Y\ i)$
by (*rule monofun-cfun-arg*)
then have $d{\cdot}x \sqsubseteq (\bigsqcup i.\ d{\cdot}(Y\ i))$
by (*simp add*: *contlub-cfun-arg* Y *idem*)

with j **have** $d{\cdot}x \sqsubseteq d{\cdot}(Y\ j)$ **by** *simp*
then have $d{\cdot}x \sqsubseteq Y\ j$
using *below* **by** (*rule below-trans*)
then show $\exists j.\ d{\cdot}x \sqsubseteq Y\ j$ **..**
qed

end

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

lemma *finite-deflation-imp-deflation*: *finite-deflation d* $\Longrightarrow$ *deflation d*
by (*simp add*: *finite-deflation-def*)

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

10.3 Continuous embedding-projection pairs

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

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

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

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

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

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

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

lemma *compact-e-rev*: *compact* $(e{\cdot}x) \Longrightarrow$ *compact* x
proof –
assume *compact* $(e{\cdot}x)$
then have *adm* $(\lambda y.\ e{\cdot}x \not\sqsubseteq y)$ **by** (*rule compactD*)
then have *adm* $(\lambda y.\ e{\cdot}x \not\sqsubseteq e{\cdot}y)$ **by** (*rule adm-subst* [*OF cont-Rep-cfun2*])
then have *adm* $(\lambda y.\ x \not\sqsubseteq y)$ **by** *simp*
then show *compact* x **by** (*rule compactI*)
qed

lemma *compact-e*:
assumes *compact* x
shows *compact* $(e{\cdot}x)$
proof –
from *assms* **have** *adm* $(\lambda y.\ x \not\sqsubseteq y)$ **by** (*rule compactD*)
then have *adm* $(\lambda y.\ x \not\sqsubseteq p{\cdot}y)$ **by** (*rule adm-subst* [*OF cont-Rep-cfun2*])
then have *adm* $(\lambda y.\ e{\cdot}x \not\sqsubseteq y)$ **by** (*simp add*: *e-below-iff-below-p*)
then show *compact* $(e{\cdot}x)$ **by** (*rule compactI*)
qed

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

Deflations from ep-pairs

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

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

lemma *finite-deflation-e-d-p*:
assumes *finite-deflation d*
shows *finite-deflation* (*e oo d oo p*)
proof

```
  interpret finite-deflation d by fact
  fix x :: 'b
  show (e oo d oo p)·((e oo d oo p)·x) = (e oo d oo p)·x
    by (simp add: idem)
  show (e oo d oo p)·x ⊑ x
    by (simp add: e-below-iff-below-p below)
  have finite ((λx. e·x) ‘ (λx. d·x) ‘ range (λx. p·x))
    by (simp add: finite-image)
  then have finite (range (λx. (e oo d oo p)·x))
    by (simp add: image-image)
  then show 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
  have p-d-e-below: (p oo d oo e)·x ⊑ x for x
  proof −
    have d·(e·x) ⊑ e·x
      by (rule d.below)
    then have p·(d·(e·x)) ⊑ p·(e·x)
      by (rule monofun-cfun-arg)
    then show ?thesis by simp
  qed
  show ?thesis
  proof
    show (p oo d oo e)·x ⊑ x for x
      by (rule p-d-e-below)
    show (p oo d oo e)·((p oo d oo e)·x) = (p oo d oo e)·x for x
    proof (rule below-antisym)
      show (p oo d oo e)·((p oo d oo e)·x) ⊑ (p oo d oo e)·x
        by (rule p-d-e-below)
      have p·(d·(d·(d·(e·x)))) ⊑ p·(d·(e·(p·(d·(e·x)))))
        by (intro monofun-cfun-arg d)
      then have p·(d·(e·x)) ⊑ p·(d·(e·(p·(d·(e·x)))))
        by (simp only: d.idem)
      then show (p oo d oo e)·x ⊑ (p oo d oo e)·((p oo d oo e)·x)
        by simp
    qed
  qed
qed

lemma finite-deflation-p-d-e:
  assumes finite-deflation d
  assumes d: ⋀x. d·x ⊑ e·(p·x)
```

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

end
```

10.4 Uniqueness of ep-pairs

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

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

lemma ep-pair-unique-p-lemma:
  assumes 1: ep-pair e p1
    and 2: ep-pair e p2
  shows p1 ⊑ p2
proof (rule cfun-belowI)
  fix x
  have e·(p1·x) ⊑ x
    by (rule ep-pair.e-p-below [OF 1])
  then have p2·(e·(p1·x)) ⊑ p2·x
    by (rule monofun-cfun-arg)
```

```
  then show p1·x ⊑ p2·x
    by (simp only: ep-pair.e-inverse [OF 2])
qed

lemma ep-pair-unique-p: ep-pair e p1 ⟹ ep-pair e p2 ⟹ p1 = p2
  by (fast intro: below-antisym elim: ep-pair-unique-p-lemma)
```

10.5 Composing ep-pairs

```
lemma ep-pair-ID-ID: ep-pair ID ID
  by standard simp-all

lemma ep-pair-comp:
  assumes ep-pair e1 p1 and ep-pair e2 p2
  shows ep-pair (e2 oo e1) (p1 oo p2)
proof
  interpret ep1: ep-pair e1 p1 by fact
  interpret ep2: ep-pair e2 p2 by fact
  fix x y
  show (p1 oo p2)·((e2 oo e1)·x) = x
    by simp
  have e1·(p1·(p2·y)) ⊑ p2·y
    by (rule ep1.e-p-below)
  then have e2·(e1·(p1·(p2·y))) ⊑ e2·(p2·y)
    by (rule monofun-cfun-arg)
  also have e2·(p2·y) ⊑ y
    by (rule ep2.e-p-below)
  finally show (e2 oo e1)·((p1 oo p2)·y) ⊑ y
    by simp
qed

locale pcpo-ep-pair = ep-pair e p
  for e :: 'a::pcpo → 'b::pcpo
  and p :: 'b::pcpo → 'a::pcpo
begin

lemma e-strict [simp]: e·⊥ = ⊥
proof -
  have ⊥ ⊑ p·⊥ by (rule minimal)
  then have e·⊥ ⊑ e·(p·⊥) by (rule monofun-cfun-arg)
  also have e·(p·⊥) ⊑ ⊥ by (rule e-p-below)
  finally show e·⊥ = ⊥ by simp
qed

lemma e-bottom-iff [simp]: e·x = ⊥ ⟷ x = ⊥
  by (rule e-eq-iff [where y=⊥, unfolded e-strict])

lemma e-defined: x ≠ ⊥ ⟹ e·x ≠ ⊥
  by simp
```

lemma *p-strict* [*simp*]: *p*·⊥ = ⊥
  by (*rule e-inverse* [**where** *x*=⊥, *unfolded e-strict*])

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

end

end

11 The type of strict products

theory *Sprod*
  imports *Cfun*
begin

11.1 Definition of strict product type

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

pcpodef (′*a*::*pcpo*, ′*b*::*pcpo*) *sprod* (‹(‹*notation*=‹*infix strict product*››- ⊗/ -)›
[*21*,*20*] *20*) =
  sprod :: (′*a* × ′*b*) *set*
  by (*simp-all add*: *sprod-def*)

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

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

11.2 Definitions of constants

definition *sfst* :: (′*a*::*pcpo* ∗∗ ′*b*::*pcpo*) → ′*a*
  where *sfst* = (Λ *p*. *fst* (*Rep-sprod p*))

definition *ssnd* :: (′*a*::*pcpo* ∗∗ ′*b*::*pcpo*) → ′*b*
  where *ssnd* = (Λ *p*. *snd* (*Rep-sprod p*))

definition *spair* :: ′*a*::*pcpo* → ′*b*::*pcpo* → (′*a* ∗∗ ′*b*)
  where *spair* = (Λ *a b*. *Abs-sprod* (*seq*·*b*·*a*, *seq*·*a*·*b*))

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

syntax
  -stuple :: [*logic*, *args*] ⇒ *logic* (‹(‹*indent*=*1 notation*=‹*mixfix strict tuple*››′(:-,/
-:′))›)
syntax-consts

-stuple ⇌ *spair*
translations
(:x, y, z:) ⇌ (:x, (:y, z:):)
(:x, y:) ⇌ *CONST spair*·x·y

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

11.3 Case analysis

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

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

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

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

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

11.4 Properties of *spair*

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

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

lemma *spair-bottom-iff* [*simp*]: (:x, y:) = ⊥ ⟷ $x = \bot \vee y = \bot$
by (*simp add*: *Rep-sprod-simps seq-conv-if*)

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

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

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

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

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

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

lemma *spair-below*: $x \neq \bot \Longrightarrow y \neq \bot \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) \longleftrightarrow x \sqsubseteq a \wedge y \sqsubseteq b$
by (*simp add*: *spair-below-iff*)

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

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

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

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

11.5 Properties of *sfst* and *ssnd*

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

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

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

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

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

lemma *ssnd-bottom-iff* [*simp*]: $ssnd\cdot p = \bot \longleftrightarrow p = \bot$
by (*cases p*) *simp-all*

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

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

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

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

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

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

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

11.6 Compactness

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

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

lemma *compact-spair*: *compact* $x \Longrightarrow$ *compact* $y \Longrightarrow$ *compact* $(:x,\ y:)$
by (*rule compact-sprod*) (*simp add*: *Rep-sprod-spair seq-conv-if*)

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

11.7 Properties of *ssplit*

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

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

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

11.8 Strict product preserves flatness

instance *sprod* :: (*flat*, *flat*) *flat*

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

end
```

12 The type of lifted values

```
theory Up
  imports Cfun
begin
```

12.1 Definition of new type for lifting

```
datatype 'a u (‹(‹notation=‹postfix lifting››-⊥)› [1000] 999) = Ibottom | Iup 'a

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

12.2 Ordering on lifted cpo

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

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

instance ..

end

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

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

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

12.3 Lifted cpo is a partial order

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

12.4 Lifted cpo is a cpo

lemma *is-lub-Iup*: $range\ S <<|\ x \Longrightarrow range\ (\lambda i.\ Iup\ (S\ i)) <<|\ Iup\ x$
  by (*auto simp*: *is-lub-def is-ub-def ball-simps below-up-def split*: *u.split*)

lemma *up-chain-lemma*:
  assumes Y: *chain* Y
  obtains $\forall i.\ Y\ i = Ibottom$
  | $A\ k$ **where** $\forall i.\ Iup\ (A\ i) = Y\ (i + k)$ **and** *chain* A **and** $range\ Y <<|\ Iup$
$(\bigsqcup i.\ A\ i)$
proof (*cases* $\exists k.\ Y\ k \neq Ibottom$)
  case *True*
  then obtain k **where** k: $Y\ k \neq Ibottom$ **..**
  define A **where** $A\ i = (THE\ a.\ Iup\ a = Y\ (i + k))$ **for** i
  have *Iup-A*: $\forall i.\ Iup\ (A\ i) = Y\ (i + k)$
  proof
    fix i :: *nat*
    from Y *le-add2* **have** $Y\ k \sqsubseteq Y\ (i + k)$ **by** (*rule chain-mono*)
    with k **have** $Y\ (i + k) \neq Ibottom$ **by** (*cases* $Y\ k$) *auto*
    then show $Iup\ (A\ i) = Y\ (i + k)$
      by (*cases* $Y\ (i + k)$, *simp-all add*: *A-def*)
  qed
  from Y **have** *chain-A*: *chain* A
    by (*simp add*: *chain-def Iup-below* [*symmetric*] *Iup-A*)
  then have $range\ A <<|\ (\bigsqcup i.\ A\ i)$
    by (*rule cpo-lubI*)
  then have $range\ (\lambda i.\ Iup\ (A\ i)) <<|\ Iup\ (\bigsqcup i.\ A\ i)$
    by (*rule is-lub-Iup*)

```
  then have range (λi. Y (i + k)) <<| Iup (⨆i. A i)
    by (simp only: Iup-A)
  then have range (λi. Y i) <<| Iup (⨆i. A i)
    by (simp only: is-lub-range-shift [OF Y])
  with Iup-A chain-A show ?thesis ..
next
  case False
  then have ∀ i. Y i = Ibottom by simp
  then show ?thesis ..
qed

instance u :: (cpo) cpo
proof
  fix S :: nat ⇒ 'a u
  assume S: chain S
  then show ∃ x. range (λi. S i) <<| x
  proof (rule up-chain-lemma)
    assume ∀ i. S i = Ibottom
    then have range (λi. S i) <<| Ibottom
      by (simp add: is-lub-const)
    then show ?thesis ..
  next
    fix A :: nat ⇒ 'a
    assume range S <<| Iup (⨆i. A i)
    then show ?thesis ..
  qed
qed
```

12.5 Lifted cpo is pointed

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

for compatibility with old HOLCF-Version

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

12.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

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

continuity for *Ifup*

```
lemma cont-Ifup1: cont (λf. Ifup f x)
```

```
  by (induct x) simp-all

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

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

12.7 Continuous versions of constants

```
definition up :: 'a → 'a u
  where up = (Λ x. Iup x)

definition fup :: ('a → 'b::pcpo) → 'a u → 'b
  where fup = (Λ f p. Ifup f p)

translations
  case l of XCONST up·x ⇒ t ⇌ CONST fup·(Λ x. t)·l
  case l of (XCONST up :: 'a)·x ⇒ t ↽ CONST fup·(Λ x. t)·l
  Λ(XCONST up·x). t ⇌ CONST fup·(Λ x. t)
```

continuous versions of lemmas for $'a_\bot$

```
lemma Exh-Up: z = ⊥ ∨ (∃ x. z = up·x)
  by (induct z) (simp add: inst-up-pcpo, simp add: up-def cont-Iup)

lemma up-eq [simp]: (up·x = up·y) = (x = y)
  by (simp add: up-def cont-Iup)

lemma up-inject: up·x = up·y ⟹ x = y
```

by *simp*

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

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

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

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

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

lifting preserves chain-finiteness

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

lemma *compact-up*: $compact\ x \Longrightarrow compact\ (up{\cdot}x)$
apply (*rule compactI2*)
apply (*erule up-chain-cases*)
apply *simp*
apply (*drule* (*1*) *compactD2*, *simp*)
apply (*erule exE*)
apply (*drule-tac f=up* **and** *x=x* **in** *monofun-cfun-arg*)
apply (*simp*, *erule exI*)
done

lemma *compact-upD*: $compact\ (up{\cdot}x) \Longrightarrow compact\ x$
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*]: $fup \cdot f \cdot \bot = \bot$
by (*simp add*: *fup-def cont-Ifup1 cont-Ifup2 inst-up-pcpo cont2cont-LAM*)

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

lemma *fup3* [*simp*]: $fup \cdot up \cdot x = x$
by (*cases x*) *simp-all*

end

13 Lifting types of class type to flat pcpo's

theory *Lift*
imports *Up*
begin

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

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

definition
Def :: *'a::type* ⇒ *'a lift* **where**
Def x = *Abs-lift* ($up \cdot$(*Discr x*))

13.1 Lift as a datatype

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

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

⊥ and *Def*

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

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

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

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

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

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

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

lemma *Def-below-iff* [*simp*]: *Def x* ⊑ *y* ⟷ *Def x* = *y*
by (*induct y*, *simp*, *simp add*: *Def-below-Def*)

13.2 Lift is flat

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

13.3 Continuity of *case-lift*

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

lemma *cont2cont-case-lift* [*simp*]:
  ⟦⋀*y*. *cont* (λ*x*. *f x y*); *cont g*⟧ ⟹ *cont* (λ*x*. *case-lift* ⊥ (*f x*) (*g x*))
unfolding *case-lift-eq* **by** (*simp add*: *cont-Rep-lift*)

13.4 Further operations

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

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

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

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

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*)

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

lemma *FLIFT-mono*:
(⋀*x*. *f x* ⊑ *g x*) ⟹ (*FLIFT x*. *f x*) ⊑ (*FLIFT x*. *g x*)
by (*rule cfun-belowI*, *case-tac x*, *simp-all*)

lemma *cont2cont-flift1* [*simp*, *cont2cont*]:
⟦⋀*y*. *cont* (λ*x*. *f x y*)⟧ ⟹ *cont* (λ*x*. *FLIFT y*. *f x y*)
by (*simp add*: *flift1-def cont2cont-LAM*)

end

14 The type of lifted booleans

theory *Tr*
imports *Lift*
begin

14.1 Type definition and constructors

type-synonym *tr* = *bool lift*

translations
(*type*) *tr* ↽ (*type*) *bool lift*

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

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

Exhaustion and Elimination for type *tr*

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

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

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

distinctness for type *tr*

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

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

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

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

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

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

14.2 Case analysis

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

abbreviation *cifte-syn* :: [*tr*, *'c::pcpo*, *'c*] ⇒ *'c* (‹(‹notation=‹mixfix If expression››If (-)/ then (-)/ else (-))› [*0*, *0*, *60*] *60*)
  where *If b then e1 else e2* ≡ *tr-case·e1·e2·b*

translations
  Λ (*XCONST TT*). *t* ⇌ *CONST tr-case·t·*⊥
  Λ (*XCONST FF*). *t* ⇌ *CONST tr-case·*⊥*·t*

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

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

14.3 Boolean connectives

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

abbreviation *andalso-syn* :: *tr* ⇒ *tr* ⇒ *tr* (‹- *andalso* -› [*36*,*35*] *35*)
where *x andalso y* ≡ *trand*·*x*·*y*

definition *tror* :: *tr* → *tr* → *tr*
where *orelse-def*: *tror* = (Λ *x y*. *If x then TT else y*)

abbreviation *orelse-syn* :: *tr* ⇒ *tr* ⇒ *tr* (‹- *orelse* -› [*31*,*30*] *30*)
where *x orelse y* ≡ *tror*·*x*·*y*

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

definition *If2* :: *tr* ⇒ ′*c*::*pcpo* ⇒ ′*c* ⇒ ′*c*
where *If2 Q x y* = (*If Q then x else y*)

tactic for tr-thms with case split

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

lemmas about andalso, orelse, neg and if

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

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

done

lemma *neg-thms* [*simp*]:
$neg{\cdot}TT = FF$
$neg{\cdot}FF = TT$
$neg{\cdot}\bot = \bot$
by (*simp-all add*: *neg-def TT-def FF-def*)

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

lemma *split-If2*: $P\ (If2\ Q\ x\ y) \longleftrightarrow ((Q = \bot \longrightarrow P\ \bot) \wedge (Q = TT \longrightarrow P\ x) \wedge (Q = FF \longrightarrow P\ y))$
by (*cases Q*) (*simp-all add*: *If2-def*)

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

14.4 Rewriting of HOLCF operations to HOL functions

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

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

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

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

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

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

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

14.5 Compactness

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

lemma *compact-FF*: *compact FF*

by (*rule compact-chfin*)

end

15 The type of strict sums

theory *Ssum*
imports *Tr*
begin

15.1 Definition of strict sum type

definition *ssum* =
$\{p :: tr \times ('a::pcpo \times 'b::pcpo).\ p = \bot \lor$
$(fst\ p = TT \land fst\ (snd\ p) \neq \bot \land snd\ (snd\ p) = \bot) \lor$
$(fst\ p = FF \land fst\ (snd\ p) = \bot \land snd\ (snd\ p) \neq \bot)\}$

pcpodef (*'a::pcpo*, *'b::pcpo*) *ssum* (‹(‹*notation*=‹*infix strict sum*››- ⊕/ -)› [*21*, *20*] *20*) =
ssum :: (*tr* × *'a* × *'b*) *set*
by (*simp-all add*: *ssum-def*)

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

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

15.2 Definitions of constructors

definition *sinl* :: *'a::pcpo* → (*'a* ++ *'b::pcpo*)
where $sinl = (\Lambda\ a.\ Abs\text{-}ssum\ (seq{\cdot}a{\cdot}TT,\ a,\ \bot))$

definition *sinr* :: *'b::pcpo* → (*'a::pcpo* ++ *'b*)
where $sinr = (\Lambda\ b.\ Abs\text{-}ssum\ (seq{\cdot}b{\cdot}FF,\ \bot,\ b))$

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

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

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

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

lemmas *Rep-ssum-simps* =

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

15.3 Properties of *sinl* and *sinr*

Ordering

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

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

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

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

Equality

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

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

lemma *sinl-eq-sinr* [*simp*]: $sinl \cdot x = sinr \cdot y \longleftrightarrow x = \bot \land y = \bot$
by (*subst po-eq-conv*) *simp*

lemma *sinr-eq-sinl* [*simp*]: $sinr \cdot x = sinl \cdot y \longleftrightarrow x = \bot \land y = \bot$
by (*subst po-eq-conv*) *simp*

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

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

Strictness

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

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

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

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

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

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

Compactness

lemma *compact-sinl*: *compact x* ⟹ *compact* (*sinl·x*)
by (*rule compact-ssum*) (*simp add*: *Rep-ssum-sinl*)

lemma *compact-sinr*: *compact x* ⟹ *compact* (*sinr·x*)
by (*rule compact-ssum*) (*simp add*: *Rep-ssum-sinr*)

lemma *compact-sinlD*: *compact* (*sinl·x*) ⟹ *compact x*
unfolding *compact-def*
by (*drule adm-subst* [*OF cont-Rep-cfun2* [**where** *f*=*sinl*]], *simp*)

lemma *compact-sinrD*: *compact* (*sinr·x*) ⟹ *compact x*
unfolding *compact-def*
by (*drule adm-subst* [*OF cont-Rep-cfun2* [**where** *f*=*sinr*]], *simp*)

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

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

15.4 Case analysis

lemma *ssumE* [*case-names bottom sinl sinr*, *cases type*: *ssum*]:
obtains $p = \bot$
| *x* **where** $p = sinl \cdot x$ **and** $x \neq \bot$
| *y* **where** $p = sinr \cdot y$ **and** $y \neq \bot$
using *Rep-ssum* [*of p*] **by** (*auto simp add*: *ssum-def Rep-ssum-simps*)

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

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

lemma *below-sinlD*: $p \sqsubseteq sinl \cdot x \Longrightarrow \exists\, y.\ p = sinl \cdot y \wedge y \sqsubseteq x$

by (*cases p*, *rule-tac x*=$\bot$ **in** *exI*, *simp-all*)

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

15.5 Case analysis combinator

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

translations
case s of XCONST sinl·x ⇒ *t1* | *XCONST sinr·y* ⇒ *t2* ⇌ *CONST sscase*·(Λ *x*. *t1*)·(Λ *y*. *t2*)·*s*
case s of (*XCONST sinl* :: *'a*)·*x* ⇒ *t1* | *XCONST sinr·y* ⇒ *t2* ⇀ *CONST sscase*·(Λ *x*. *t1*)·(Λ *y*. *t2*)·*s*

translations
Λ(*XCONST sinl·x*). *t* ⇌ *CONST sscase*·(Λ *x*. *t*)·⊥
Λ(*XCONST sinr·y*). *t* ⇌ *CONST sscase*·⊥·(Λ *y*. *t*)

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

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

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

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

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

15.6 Strict sum preserves flatness

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

end

16 The Strict Function Type

theory *Sfun*

imports *Cfun*
begin

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

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

TODO: Define nice syntax for abstraction, application.

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

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

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

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

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

lemma *strictify-cancel*: $f\cdot\bot = \bot \Longrightarrow$ *strictify*·*f* = *f*
by (*simp add*: *cfun-eq-iff strictify-conv-if*)

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

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

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

lemma *sfun-below-iff*: $f \sqsubseteq g \longleftrightarrow$ *sfun-rep*·*f* $\sqsubseteq$ *sfun-rep*·*g*
by (*simp add*: *sfun-rep-def cont-Rep-sfun below-sfun-def*)

end

17 Map functions for various types

theory *Map-Functions*
imports *Deflation Sprod Ssum Sfun Up*
begin

17.1 Map operator for continuous function space

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

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

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

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

lemma *ep-pair-cfun-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*cfun-map*$\cdot p1\cdot e2$) (*cfun-map*$\cdot e1\cdot p2$)
proof
interpret *e1p1*: *ep-pair e1 p1* **by** *fact*
interpret *e2p2*: *ep-pair e2 p2* **by** *fact*
show *cfun-map*$\cdot e1\cdot p2\cdot($*cfun-map*$\cdot p1\cdot e2\cdot f) = f$ **for** *f*
by (*simp add*: *cfun-eq-iff*)
show *cfun-map*$\cdot p1\cdot e2\cdot($*cfun-map*$\cdot e1\cdot p2\cdot g) \sqsubseteq g$ **for** *g*
apply (*rule cfun-belowI*, *simp*)
apply (*rule below-trans* [*OF e2p2.e-p-below*])
apply (*rule monofun-cfun-arg*)
apply (*rule e1p1.e-p-below*)
done
qed

lemma *deflation-cfun-map*:
assumes *deflation d1* **and** *deflation d2*
shows *deflation* (*cfun-map*$\cdot d1\cdot d2$)
proof
interpret *d1*: *deflation d1* **by** *fact*
interpret *d2*: *deflation d2* **by** *fact*
fix *f*
show *cfun-map*$\cdot d1\cdot d2\cdot($*cfun-map*$\cdot d1\cdot d2\cdot f) =$ *cfun-map*$\cdot d1\cdot d2\cdot f$
by (*simp add*: *cfun-eq-iff d1.idem d2.idem*)
show *cfun-map*$\cdot d1\cdot d2\cdot f \sqsubseteq f$
apply (*rule cfun-belowI*, *simp*)
apply (*rule below-trans* [*OF d2.below*])
apply (*rule monofun-cfun-arg*)

```
    apply (rule d1.below)
    done
qed

lemma finite-range-cfun-map:
  assumes a: finite (range (λx. a·x))
  assumes b: finite (range (λy. b·y))
  shows finite (range (λf. cfun-map·a·b·f))  (is finite (range ?h))
proof (rule finite-imageD)
  let ?f = λg. range (λx. (a·x, g·x))
  show finite (?f ` range ?h)
  proof (rule finite-subset)
    let ?B = Pow (range (λx. a·x) × range (λy. b·y))
    show ?f ` range ?h ⊆ ?B
      by clarsimp
    show finite ?B
      by (simp add: a b)
  qed
  show inj-on ?f (range ?h)
  proof (rule inj-onI, rule cfun-eqI, clarsimp)
    fix x f g
    assume range (λx. (a·x, b·(f·(a·x)))) = range (λx. (a·x, b·(g·(a·x))))
    then have range (λx. (a·x, b·(f·(a·x)))) ⊆ range (λx. (a·x, b·(g·(a·x))))
      by (rule equalityD1)
    then have (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)
    then show b·(f·(a·x)) = b·(g·(a·x))
      by clarsimp
  qed
qed

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

Finite deflations are compact elements of the function space

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

17.2 Map operator for product type

definition *prod-map* :: $('a \to 'b) \to ('c \to 'd) \to 'a \times 'c \to 'b \times 'd$
  where *prod-map* = (Λ *f g p*. (*f*·(*fst p*), *g*·(*snd p*)))

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

lemma *prod-map-ID*: *prod-map*·*ID*·*ID* = *ID*
  by (*auto simp*: *cfun-eq-iff*)

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

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

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

lemma *finite-deflation-prod-map*:

assumes *finite-deflation d1* **and** *finite-deflation d2*
shows *finite-deflation* (*prod-map·d1·d2*)
proof (*rule finite-deflation-intro*)
interpret *d1*: *finite-deflation d1* **by** *fact*
interpret *d2*: *finite-deflation d2* **by** *fact*
from *d1.deflation-axioms d2.deflation-axioms* **show** *deflation* (*prod-map·d1·d2*)
by (*rule deflation-prod-map*)
have {*p. prod-map·d1·d2·p* = *p*} ⊆ {*x. d1·x* = *x*} × {*y. d2·y* = *y*}
by *auto*
then show *finite* {*p. prod-map·d1·d2·p* = *p*}
by (*rule finite-subset*, *simp add*: *d1.finite-fixes d2.finite-fixes*)
qed

17.3 Map function for lifted cpo

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

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

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

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

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

lemma *u-map-oo*: *u-map·*(*f oo g*) = *u-map·f oo u-map·g*
by (*simp add*: *cfcomp1 u-map-map eta-cfun*)

lemma *ep-pair-u-map*: *ep-pair e p* ⟹ *ep-pair* (*u-map·e*) (*u-map·p*)
apply *standard*
subgoal for *x* **by** (*cases x*) (*simp-all add*: *ep-pair.e-inverse*)
subgoal for *y* **by** (*cases y*) (*simp-all add*: *ep-pair.e-p-below*)
done

lemma *deflation-u-map*: *deflation d* ⟹ *deflation* (*u-map·d*)
apply *standard*
subgoal for *x* **by** (*cases x*) (*simp-all add*: *deflation.idem*)
subgoal for *x* **by** (*cases x*) (*simp-all add*: *deflation.below*)
done

lemma *finite-deflation-u-map*:
assumes *finite-deflation d*
shows *finite-deflation* (*u-map·d*)
proof (*rule finite-deflation-intro*)

interpret *d*: *finite-deflation d* **by** *fact*
from *d.deflation-axioms* **show** *deflation* (*u-map*·*d*)
by (*rule deflation-u-map*)
have $\{x.\ u\text{-}map \cdot d \cdot x = x\} \subseteq insert\ \bot\ ((\lambda x.\ up \cdot x)\ `\ \{x.\ d \cdot x = x\})$
by (*rule subsetI*, *case-tac x*, *simp-all*)
then show *finite* $\{x.\ u\text{-}map \cdot d \cdot x = x\}$
by (*rule finite-subset*) (*simp add*: *d.finite-fixes*)
qed

17.4 Map function for strict products

definition *sprod-map* :: $('a::pcpo \to 'b::pcpo) \to ('c::pcpo \to 'd::pcpo) \to 'a \otimes 'c \to 'b \otimes 'd$
where $sprod\text{-}map = (\Lambda\ f\ g.\ ssplit \cdot (\Lambda\ x\ y.\ (:f \cdot x,\ g \cdot y:)))$

lemma *sprod-map-strict* [*simp*]: $sprod\text{-}map \cdot a \cdot b \cdot \bot = \bot$
by (*simp add*: *sprod-map-def*)

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

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

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

lemma *sprod-map-map*:
$[\![f1 \cdot \bot = \bot;\ g1 \cdot \bot = \bot]\!] \Longrightarrow$
$sprod\text{-}map \cdot f1 \cdot g1 \cdot (sprod\text{-}map \cdot f2 \cdot g2 \cdot p) =$
$sprod\text{-}map \cdot (\Lambda\ x.\ f1 \cdot (f2 \cdot x)) \cdot (\Lambda\ x.\ g1 \cdot (g2 \cdot x)) \cdot p$
proof (*induct p*)
case *bottom*
then show *?case* **by** *simp*
next
case (*spair x y*)
then show *?case*
apply (*cases* $f2 \cdot x = \bot$, *simp*)
apply (*cases* $g2 \cdot y = \bot$, *simp*)
apply *simp*
done
qed

lemma *ep-pair-sprod-map*:
assumes *ep-pair e1 p1* **and** *ep-pair e2 p2*
shows *ep-pair* (*sprod-map*·*e1*·*e2*) (*sprod-map*·*p1*·*p2*)
proof

```
  interpret e1p1: pcpo-ep-pair e1 p1 unfolding pcpo-ep-pair-def by fact
  interpret e2p2: pcpo-ep-pair e2 p2 unfolding pcpo-ep-pair-def by fact
  show sprod-map·p1·p2·(sprod-map·e1·e2·x) = x for x
    by (induct x) simp-all
  show sprod-map·e1·e2·(sprod-map·p1·p2·y) ⊑ y for y
  proof (induct y)
    case bottom
    then show ?case by simp
  next
    case (spair x y)
    then show ?case
      apply simp
      apply (cases p1·x = ⊥, simp, cases p2·y = ⊥, simp)
      apply (simp add: monofun-cfun e1p1.e-p-below e2p2.e-p-below)
      done
  qed
qed

lemma deflation-sprod-map:
  assumes deflation d1 and deflation d2
  shows deflation (sprod-map·d1·d2)
proof
  interpret d1: deflation d1 by fact
  interpret d2: deflation d2 by fact
  fix x
  show sprod-map·d1·d2·(sprod-map·d1·d2·x) = sprod-map·d1·d2·x
  proof (induct x)
    case bottom
    then show ?case by simp
  next
    case (spair x y)
    then show ?case
      apply (cases d1·x = ⊥, simp, cases d2·y = ⊥, simp)
      apply (simp add: d1.idem d2.idem)
      done
  qed
  show sprod-map·d1·d2·x ⊑ x
  proof (induct x)
    case bottom
    then show ?case by simp
  next
    case spair
    then show ?case by (simp add: monofun-cfun d1.below d2.below)
  qed
qed

lemma finite-deflation-sprod-map:
  assumes finite-deflation d1 and finite-deflation d2
  shows finite-deflation (sprod-map·d1·d2)
```

proof (*rule finite-deflation-intro*)
  interpret *d1*: *finite-deflation d1* **by** *fact*
  interpret *d2*: *finite-deflation d2* **by** *fact*
  from *d1.deflation-axioms d2.deflation-axioms* **show** *deflation* (*sprod-map·d1·d2*)
    by (*rule deflation-sprod-map*)
  have $\{x.\ sprod\text{-}map \cdot d1 \cdot d2 \cdot x = x\} \subseteq$
      $insert\ \bot\ ((\lambda(x, y).\ (:x, y:))\ `\ (\{x.\ d1 \cdot x = x\} \times \{y.\ d2 \cdot y = y\}))$
    by (*rule subsetI*, *case-tac x*, *auto simp add*: *spair-eq-iff*)
  then show *finite* $\{x.\ sprod\text{-}map \cdot d1 \cdot d2 \cdot x = x\}$
    by (*rule finite-subset*) (*simp add*: *d1.finite-fixes d2.finite-fixes*)
qed

17.5 Map function for strict sums

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

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

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

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

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

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

lemma *ssum-map-ID*: $ssum\text{-}map \cdot ID \cdot ID = ID$
  by (*simp add*: *ssum-map-def cfun-eq-iff eta-cfun*)

lemma *ssum-map-map*:
  $[\![f1 \cdot \bot = \bot;\ g1 \cdot \bot = \bot]\!] \Longrightarrow$
    $ssum\text{-}map \cdot f1 \cdot g1 \cdot (ssum\text{-}map \cdot f2 \cdot g2 \cdot p) =$
     $ssum\text{-}map \cdot (\Lambda\ x.\ f1 \cdot (f2 \cdot x)) \cdot (\Lambda\ x.\ g1 \cdot (g2 \cdot x)) \cdot p$
proof (*induct p*)
  case *bottom*
  then show *?case* **by** *simp*
next
  case (*sinl x*)
  then show *?case* **by** (*cases* $f2 \cdot x = \bot$) *simp-all*
next
  case (*sinr y*)
  then show *?case* **by** (*cases* $g2 \cdot y = \bot$) *simp-all*

```
qed

lemma ep-pair-ssum-map:
  assumes ep-pair e1 p1 and ep-pair e2 p2
  shows ep-pair (ssum-map·e1·e2) (ssum-map·p1·p2)
proof
  interpret e1p1: pcpo-ep-pair e1 p1 unfolding pcpo-ep-pair-def by fact
  interpret e2p2: pcpo-ep-pair e2 p2 unfolding pcpo-ep-pair-def by fact
  show ssum-map·p1·p2·(ssum-map·e1·e2·x) = x for x
    by (induct x) simp-all
  show ssum-map·e1·e2·(ssum-map·p1·p2·y) ⊑ y for y
  proof (induct y)
    case bottom
    then show ?case by simp
  next
    case (sinl x)
    then show ?case by (cases p1·x = ⊥) (simp-all add: e1p1.e-p-below)
  next
    case (sinr y)
    then show ?case by (cases p2·y = ⊥) (simp-all add: e2p2.e-p-below)
  qed
qed

lemma deflation-ssum-map:
  assumes deflation d1 and deflation d2
  shows deflation (ssum-map·d1·d2)
proof
  interpret d1: deflation d1 by fact
  interpret d2: deflation d2 by fact
  fix x
  show ssum-map·d1·d2·(ssum-map·d1·d2·x) = ssum-map·d1·d2·x
  proof (induct x)
    case bottom
    then show ?case by simp
  next
    case (sinl x)
    then show ?case by (cases d1·x = ⊥) (simp-all add: d1.idem)
  next
    case (sinr y)
    then show ?case by (cases d2·y = ⊥) (simp-all add: d2.idem)
  qed
  show ssum-map·d1·d2·x ⊑ x
  proof (induct x)
    case bottom
    then show ?case by simp
  next
    case (sinl x)
    then show ?case by (cases d1·x = ⊥) (simp-all add: d1.below)
  next
```

```
    case (sinr y)
    then show ?case by (cases d2·y = ⊥) (simp-all add: d2.below)
  qed
qed

lemma finite-deflation-ssum-map:
  assumes finite-deflation d1 and finite-deflation d2
  shows finite-deflation (ssum-map·d1·d2)
proof (rule finite-deflation-intro)
  interpret d1: finite-deflation d1 by fact
  interpret d2: finite-deflation d2 by fact
  from d1.deflation-axioms d2.deflation-axioms show deflation (ssum-map·d1·d2)
    by (rule deflation-ssum-map)
  have {x. ssum-map·d1·d2·x = x} ⊆
      (λx. sinl·x) ` {x. d1·x = x} ∪
      (λx. sinr·x) ` {x. d2·x = x} ∪ {⊥}
    by (rule subsetI, case-tac x, simp-all)
  then show finite {x. ssum-map·d1·d2·x = x}
    by (rule finite-subset, simp add: d1.finite-fixes d2.finite-fixes)
qed
```

17.6 Map operator for strict function space

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

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

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

lemma ep-pair-sfun-map:
  assumes 1: ep-pair e1 p1
  assumes 2: ep-pair e2 p2
  shows ep-pair (sfun-map·p1·e2) (sfun-map·e1·p2)
proof
  interpret e1p1: pcpo-ep-pair e1 p1
    unfolding pcpo-ep-pair-def by fact
  interpret e2p2: pcpo-ep-pair e2 p2
    unfolding pcpo-ep-pair-def by fact
  show sfun-map·e1·p2·(sfun-map·p1·e2·f) = f for f
    unfolding sfun-map-def
    apply (simp add: sfun-eq-iff strictify-cancel)
    apply (rule ep-pair.e-inverse)
```

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

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

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

end
```

18 The cpo of cartesian products

theory *Cprod*
imports *Cfun*
begin

18.1 Continuous case function for unit type

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

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

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

18.2 Continuous version of split function

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

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

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

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

18.3 Convert all lemmas to the continuous versions

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

lemma *csplit-Pair* [*simp*]: *csplit*$\cdot f \cdot (x, y) = f \cdot x \cdot y$
by (*simp add*: *csplit-def*)

end

19 Profinite and bifinite cpos

theory *Bifinite*
imports *Map-Functions Cprod Sprod Sfun Up HOL−Library.Countable*
begin

19.1 Chains of finite deflations

locale *approx-chain* =

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

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

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

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

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

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

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

end

19.2 Omega-profinite and bifinite domains

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

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

19.3 Building approx chains

lemma *approx-chain-iso*:
assumes *a*: *approx-chain a*
assumes [*simp*]: ⋀*x. f*·(*g*·*x*) = *x*
assumes [*simp*]: ⋀*y. g*·(*f*·*y*) = *y*
shows *approx-chain* (*λi. f oo a i oo g*)
proof −
have *1*: *f oo g* = *ID* **by** (*simp add*: *cfun-eqI*)
have *2*: *ep-pair f g* **by** (*simp add*: *ep-pair-def*)
from *1 2* **show** *?thesis*
using *a* **unfolding** *approx-chain-def*

by (*simp add*: *lub-APP ep-pair.finite-deflation-e-d-p*)
qed

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

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

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

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

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

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

Approx chains for countable discrete types.

definition *discr-approx* :: *nat* ⇒ *'a::countable discr u* → *'a discr u*
where *discr-approx* = ($\lambda i.$ Λ(*up*·*x*). *if to-nat* (*undiscr x*) < *i then up*·*x else* ⊥)

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

lemma *lub-discr-approx* [*simp*]: ($\bigsqcup i.$ *discr-approx i*) = *ID*
apply (*rule cfun-eqI*)

```
  apply (simp add: contlub-cfun-fun)
  apply (simp add: discr-approx-def)
  subgoal for x
    apply (cases x)
     apply simp
    apply (rule lub-eqI)
    apply (rule is-lubI)
     apply (rule ub-rangeI, simp)
    apply (drule ub-rangeD)
    apply (erule rev-below-trans)
    apply simp
    apply (rule lessI)
    done
  done

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

lemma finite-deflation-discr-approx: finite-deflation (discr-approx i)
proof
  fix x :: 'a discr u
  show discr-approx i·x ⊑ x
    unfolding discr-approx-def
    by (cases x, simp, simp)
  show discr-approx i·(discr-approx i·x) = discr-approx i·x
    unfolding discr-approx-def
    by (cases x, simp, simp)
  show finite {x::'a discr u. discr-approx i·x = x}
  proof (rule finite-subset)
    let ?S = insert (⊥::'a discr u) ((λx. up·x) ‘ undiscr −‘ to-nat −‘ {..<i})
    show {x::'a discr u. discr-approx i·x = x} ⊆ ?S
      unfolding discr-approx-def
      by (rule subsetI, case-tac x, simp, simp split: if-split-asm)
    show finite ?S
      by (simp add: finite-vimageI)
  qed
qed

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

19.4 Class instance proofs

```
instance bifinite ⊆ profinite
proof
  show ∃(a::nat ⇒ 'a⊥ → 'a⊥). approx-chain a
    using bifinite [where 'a='a]
    by (fast intro!: approx-chain-u-map)
```

qed

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

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

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

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

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

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

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

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

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

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

lemma *decode-encode-prod-u* [*simp*]: *decode-prod-u*·(*encode-prod-u*·*x*) = *x*
  unfolding *encode-prod-u-def decode-prod-u-def*
  apply (*cases x*)
   apply *simp*
  subgoal for *y* **by** (*cases y*) *simp*
  done

lemma *encode-decode-prod-u* [*simp*]: *encode-prod-u*·(*decode-prod-u*·*y*) = *y*
  unfolding *encode-prod-u-def decode-prod-u-def*
  apply (*cases y*)
   apply *simp*
  subgoal for *a b*

apply (*cases a*, *simp*)
apply (*cases b*, *simp*, *simp*)
done
done

instance *prod* :: (*profinite*, *profinite*) *profinite*
proof
obtain $a :: nat \Rightarrow 'a_\bot \to 'a_\bot$ **where** *a*: *approx-chain a*
using *profinite* **..**
obtain $b :: nat \Rightarrow 'b_\bot \to 'b_\bot$ **where** *b*: *approx-chain b*
using *profinite* **..**
have *approx-chain* ($\lambda i.$ *decode-prod-u oo sprod-map*$\cdot$(*a i*)$\cdot$(*b i*) *oo encode-prod-u*)
using *a b* **by** (*simp add*: *approx-chain-iso approx-chain-sprod-map*)
thus $\exists (a::nat \Rightarrow ('a \times 'b)_\bot \to ('a \times 'b)_\bot).$ *approx-chain a*
by − (*rule exI*)
qed

instance *prod* :: (*bifinite*, *bifinite*) *bifinite*
proof
show $\exists (a::nat \Rightarrow ('a \times 'b) \to ('a \times 'b)).$ *approx-chain a*
using *bifinite* [**where** $'a='a$] **and** *bifinite* [**where** $'a='b$]
by (*fast intro*!: *approx-chain-prod-map*)
qed

instance *sfun* :: (*bifinite*, *bifinite*) *bifinite*
proof
show $\exists (a::nat \Rightarrow ('a \to! 'b) \to ('a \to! 'b)).$ *approx-chain a*
using *bifinite* [**where** $'a='a$] **and** *bifinite* [**where** $'a='b$]
by (*fast intro*!: *approx-chain-sfun-map*)
qed

instance *sprod* :: (*bifinite*, *bifinite*) *bifinite*
proof
show $\exists (a::nat \Rightarrow ('a \otimes 'b) \to ('a \otimes 'b)).$ *approx-chain a*
using *bifinite* [**where** $'a='a$] **and** *bifinite* [**where** $'a='b$]
by (*fast intro*!: *approx-chain-sprod-map*)
qed

instance *ssum* :: (*bifinite*, *bifinite*) *bifinite*
proof
show $\exists (a::nat \Rightarrow ('a \oplus 'b) \to ('a \oplus 'b)).$ *approx-chain a*
using *bifinite* [**where** $'a='a$] **and** *bifinite* [**where** $'a='b$]
by (*fast intro*!: *approx-chain-ssum-map*)
qed

lemma *approx-chain-unit*: *approx-chain* ($\bot :: nat \Rightarrow unit \to unit$)
by (*simp add*: *approx-chain-def cfun-eq-iff finite-deflation-bottom*)

instance *unit* :: *bifinite*

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

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

instance *lift* :: (*countable*) *bifinite*
proof
note [*simp*] = *cont-Abs-lift cont-Rep-lift Rep-lift-inverse Abs-lift-inverse*
obtain $a :: nat \Rightarrow ('a\ discr)_\bot \rightarrow ('a\ discr)_\bot$ **where** *a*: *approx-chain a*
using *profinite* **..**
hence *approx-chain* ($\lambda i.$ (Λ *y*. *Abs-lift y*) *oo a i oo* (Λ *x*. *Rep-lift x*))
by (*rule approx-chain-iso*) *simp-all*
thus $\exists (a::nat \Rightarrow 'a\ lift \rightarrow 'a\ lift).$ *approx-chain a*
by − (*rule exI*)
qed

end

20 Defining algebraic domains by ideal completion

theory *Completion*
imports *Cfun*
begin

20.1 Ideals over a preorder

locale *preorder* =
fixes *r* :: *'a*::*type* ⇒ *'a* ⇒ *bool* (**infix** ‹⪯› *50*)
assumes *r-refl*: $x \preceq x$
assumes *r-trans*: $[\![x \preceq y;\ y \preceq z]\!] \Longrightarrow x \preceq z$
begin

definition
ideal :: *'a set* ⇒ *bool* **where**
$ideal\ A = ((\exists x.\ x \in A) \land (\forall x{\in}A.\ \forall y{\in}A.\ \exists z{\in}A.\ x \preceq z \land y \preceq z) \land$
$(\forall x\ y.\ x \preceq y \longrightarrow y \in A \longrightarrow x \in A))$

lemma *idealI*:
assumes $\exists x.\ x \in A$
assumes $\bigwedge x\ y.\ [\![x \in A;\ y \in A]\!] \Longrightarrow \exists z{\in}A.\ x \preceq z \land y \preceq z$
assumes $\bigwedge x\ y.\ [\![x \preceq y;\ y \in A]\!] \Longrightarrow x \in A$
shows *ideal A*
unfolding *ideal-def* **using** *assms* **by** *fast*

lemma *idealD1*:
$ideal\ A \Longrightarrow \exists x.\ x \in A$
unfolding *ideal-def* **by** *fast*

lemma *idealD2*:

```
  ⟦ideal A; x ∈ A; y ∈ A⟧ ⟹ ∃z∈A. x ≼ z ∧ y ≼ z
unfolding ideal-def by fast

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

lemma ideal-principal: ideal {x. x ≼ z}
  apply (rule idealI)
    apply (rule exI [where x = z])
    apply (fast intro: r-refl)
   apply (rule bexI [where x = z], fast)
   apply (fast intro: r-refl)
  apply (fast intro: r-trans)
  done

lemma ex-ideal: ∃A. A ∈ {A. ideal A}
by (fast intro: ideal-principal)
```

The set of ideals is a cpo

```
lemma ideal-UN:
  fixes A :: nat ⇒ 'a set
  assumes ideal-A: ⋀i. ideal (A i)
  assumes chain-A: ⋀i j. i ≤ j ⟹ A i ⊆ A j
  shows ideal (⋃i. A i)
  apply (rule idealI)
  using idealD1 [OF ideal-A] apply fast
   apply (clarify)
  subgoal for i j
    apply (drule subsetD [OF chain-A [OF max.cobounded1]])
    apply (drule subsetD [OF chain-A [OF max.cobounded2]])
    apply (drule (1) idealD2 [OF ideal-A])
    apply blast
    done
  apply clarify
  apply (drule (1) idealD3 [OF ideal-A])
  apply fast
  done

lemma typedef-ideal-po:
  fixes Abs :: 'a set ⇒ 'b::below
  assumes type: type-definition Rep Abs {S. ideal S}
  assumes below: ⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y
  shows OFCLASS('b, po-class)
 apply (intro-classes, unfold below)
   apply (rule subset-refl)
  apply (erule (1) subset-trans)
 apply (rule type-definition.Rep-inject [OF type, THEN iffD1])
 apply (erule (1) subset-antisym)
```

```
done

lemma
  fixes Abs :: 'a set ⇒ 'b::po
  assumes type: type-definition Rep Abs {S. ideal S}
  assumes below: ⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y
  assumes S: chain S
  shows typedef-ideal-lub: range S <<| Abs (⋃ i. Rep (S i))
    and typedef-ideal-rep-lub: Rep (⨆ i. S i) = (⋃ i. Rep (S i))
proof −
  have 1: ideal (⋃ i. Rep (S i))
    apply (rule ideal-UN)
     apply (rule type-definition.Rep [OF type, unfolded mem-Collect-eq])
    apply (subst below [symmetric])
    apply (erule chain-mono [OF S])
    done
  hence 2: Rep (Abs (⋃ i. Rep (S i))) = (⋃ i. Rep (S i))
    by (simp add: type-definition.Abs-inverse [OF type])
  show 3: range S <<| Abs (⋃ i. Rep (S i))
    apply (rule is-lubI)
     apply (rule is-ubI)
     apply (simp add: below 2, fast)
    apply (simp add: below 2 is-ub-def, fast)
    done
  hence 4: (⨆ i. S i) = Abs (⋃ i. Rep (S i))
    by (rule lub-eqI)
  show 5: Rep (⨆ i. S i) = (⋃ i. Rep (S i))
    by (simp add: 4 2)
qed

lemma typedef-ideal-cpo:
  fixes Abs :: 'a set ⇒ 'b::po
  assumes type: type-definition Rep Abs {S. ideal S}
  assumes below: ⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y
  shows OFCLASS('b, cpo-class)
  by standard (rule exI, erule typedef-ideal-lub [OF type below])

end

interpretation below: preorder below :: 'a::po ⇒ 'a ⇒ bool
apply unfold-locales
apply (rule below-refl)
apply (erule (1) below-trans)
done
```

20.2 Lemmas about least upper bounds

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

apply (*drule is-lubD1*)
apply (*erule* (*1*) *is-ubD*)
done

lemma *is-lub-thelub-ex*: $[\![\exists u.\ S <<|\ u;\ S <|\ x]\!] \Longrightarrow lub\ S \sqsubseteq x$
by (*erule exE*, *drule is-lub-lub*, *erule is-lubD2*)

20.3 Locale for ideal completion

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

locale *ideal-completion* = *preorder* +
  fixes *principal* :: $'a::type \Rightarrow 'b$
  fixes *rep* :: $'b \Rightarrow 'a::type\ set$
  assumes *ideal-rep*: $\bigwedge x.\ ideal\ (rep\ x)$
  assumes *rep-lub*: $\bigwedge Y.\ chain\ Y \Longrightarrow rep\ (\bigsqcup i.\ Y\ i) = (\bigcup i.\ rep\ (Y\ i))$
  assumes *rep-principal*: $\bigwedge a.\ rep\ (principal\ a) = \{b.\ b \preceq a\}$
  assumes *belowI*: $\bigwedge x\ y.\ rep\ x \subseteq rep\ y \Longrightarrow x \sqsubseteq y$
  assumes *countable*: $\exists f::'a \Rightarrow nat.\ inj\ f$
begin

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

lemma *below-def*: $x \sqsubseteq y \longleftrightarrow rep\ x \subseteq rep\ y$
by (*rule iffI* [*OF rep-mono belowI*])

lemma *principal-below-iff-mem-rep*: $principal\ a \sqsubseteq x \longleftrightarrow a \in rep\ x$
unfolding *below-def rep-principal*
by (*auto intro*: *r-refl elim*: *idealD3* [*OF ideal-rep*])

lemma *principal-below-iff* [*simp*]: $principal\ a \sqsubseteq principal\ b \longleftrightarrow a \preceq b$
by (*simp add*: *principal-below-iff-mem-rep rep-principal*)

lemma *principal-eq-iff*: $principal\ a = principal\ b \longleftrightarrow a \preceq b \wedge b \preceq a$
unfolding *po-eq-conv* [**where** *'a='b*] *principal-below-iff* **..**

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

lemma *principal-mono*: $a \preceq b \Longrightarrow principal\ a \sqsubseteq principal\ b$
by (*simp only*: *principal-below-iff*)

lemma *ch2ch-principal* [*simp*]:
  $\forall i.\ Y\ i \preceq Y\ (Suc\ i) \Longrightarrow chain\ (\lambda i.\ principal\ (Y\ i))$

by (*simp add*: *chainI principal-mono*)

20.3.1 Principal ideals approximate all elements

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

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

```
lemma obtain-principal-chain:
  obtains Y where ∀ i. Y i ⪯ Y (Suc i) and x = (⨆ i. principal (Y i))
proof −
  obtain count :: 'a ⇒ nat where inj: inj count
    using countable ..
  define enum where enum i = (THE a. count a = i) for i
  have enum-count [simp]: ⋀x. enum (count x) = x
    unfolding enum-def by (simp add: inj-eq [OF inj])
  define a where a = (LEAST i. enum i ∈ rep x)
  define b where b i = (LEAST j. enum j ∈ rep x ∧ ¬ enum j ⪯ enum i) for i
  define c where c i j = (LEAST k. enum k ∈ rep x ∧ enum i ⪯ enum k ∧ enum
j ⪯ enum k) for i j
  define P where P i ⟷ (∃ j. enum j ∈ rep x ∧ ¬ enum j ⪯ enum i) for i
  define X where X = rec-nat a (λn i. if P i then c i (b i) else i)
  have X-0: X 0 = a unfolding X-def by simp
  have X-Suc: ⋀n. X (Suc n) = (if P (X n) then c (X n) (b (X n)) else X n)
    unfolding X-def by simp
  have a-mem: enum a ∈ rep x
    unfolding a-def
    apply (rule LeastI-ex)
    apply (insert ideal-rep [of x])
    apply (drule idealD1)
    apply (clarify)
    subgoal for a by (rule exI [where x=count a]) simp
    done
  have b: ⋀i. P i ⟹ enum i ∈ rep x
    ⟹ enum (b i) ∈ rep x ∧ ¬ enum (b i) ⪯ enum i
    unfolding P-def b-def by (erule LeastI2-ex, simp)
  have c: ⋀i j. enum i ∈ rep x ⟹ enum j ∈ rep x
    ⟹ enum (c i j) ∈ rep x ∧ enum i ⪯ enum (c i j) ∧ enum j ⪯ enum (c i j)
    unfolding c-def
    apply (drule (1) idealD2 [OF ideal-rep], clarify)
    subgoal for ... z by (rule LeastI2 [where a=count z], simp, simp)
    done
  have X-mem: enum (X n) ∈ rep x for n
  proof (induct n)
    case 0
    then show ?case by (simp add: X-0 a-mem)
  next
    case (Suc n)
    with b c show ?case by (auto simp: X-Suc)
```

```
  qed
  have X-chain: ⋀n. enum (X n) ≼ enum (X (Suc n))
    apply (clarsimp simp add: X-Suc r-refl)
    apply (simp add: b c X-mem)
    done
  have less-b: ⋀n i. n < b i ⟹ enum n ∈ rep x ⟹ enum n ≼ enum i
    unfolding b-def by (drule not-less-Least, simp)
  have X-covers: ∀k≤n. enum k ∈ rep x ⟶ enum k ≼ enum (X n) for n
  proof (induct n)
    case 0
    then show ?case
      apply (clarsimp simp add: X-0 a-def)
      apply (drule Least-le [where k=0], simp add: r-refl)
      done
  next
    case (Suc n)
    then show ?case
      apply clarsimp
      apply (erule le-SucE)
       apply (rule r-trans [OF - X-chain], simp)
      apply (cases P (X n), simp add: X-Suc)
       apply (rule linorder-cases [where x=b (X n) and y=Suc n])
         apply (simp only: less-Suc-eq-le)
         apply (drule spec, drule (1) mp, simp add: b X-mem)
        apply (simp add: c X-mem)
       apply (drule (1) less-b)
       apply (erule r-trans)
       apply (simp add: b c X-mem)
      apply (simp add: X-Suc)
      apply (simp add: P-def)
      done
  qed
  have 1: ∀i. enum (X i) ≼ enum (X (Suc i))
    by (simp add: X-chain)
  have x = (⨆n. principal (enum (X n)))
    apply (simp add: eq-iff rep-lub 1 rep-principal)
    apply auto
    subgoal for a
      apply (subgoal-tac ∃i. a = enum i, erule exE)
       apply (rule-tac x=i in exI, simp add: X-covers)
      apply (rule-tac x=count a in exI, simp)
      done
    subgoal
      apply (erule idealD3 [OF ideal-rep])
      apply (rule X-mem)
      done
    done
  with 1 show ?thesis ..
qed
```

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

lemma compact-imp-principal: compact x ⟹ ∃ a. x = principal a
apply (rule obtain-principal-chain [of x])
apply (drule adm-compact-neq [OF - cont-id])
apply (subgoal-tac chain (λi. principal (Y i)))
apply (drule (2) admD2, fast, simp)
done
```

20.4 Defining functions in terms of basis elements

```
definition
  extension :: ('a::type ⇒ 'c) ⇒ 'b → 'c where
  extension = (λf. (Λ x. lub (f ` rep x)))

lemma extension-lemma:
  fixes f :: 'a::type ⇒ 'c
  assumes f-mono: ⋀a b. a ≼ b ⟹ f a ⊑ f b
  shows ∃ u. f ` rep x <<| u
proof −
  obtain Y where Y: ∀ i. Y i ≼ Y (Suc i)
  and x: x = (⨆ i. principal (Y i))
    by (rule obtain-principal-chain [of x])
  have chain: chain (λi. f (Y i))
    by (rule chainI, simp add: f-mono Y)
  have rep-x: rep x = (⋃ n. {a. a ≼ Y n})
    by (simp add: x rep-lub Y rep-principal)
  have f ` rep x <<| (⨆ n. f (Y n))
    apply (rule is-lubI)
     apply (rule ub-imageI)
    subgoal for a
      apply (clarsimp simp add: rep-x)
      apply (drule f-mono)
      apply (erule below-lub [OF chain])
      done
    apply (rule lub-below [OF chain])
    subgoal for ... n
      apply (drule ub-imageD [where x=Y n])
       apply (simp add: rep-x, fast intro: r-refl)
      apply assumption
      done
    done
```

```
  then show ?thesis ..
qed

lemma extension-beta:
  fixes f :: 'a::type ⇒ 'c
  assumes f-mono: ⋀a b. a ≼ b ⟹ f a ⊑ f b
  shows extension f·x = lub (f ` rep x)
unfolding extension-def
proof (rule beta-cfun)
  have lub: ⋀x. ∃u. f ` rep x <<| u
    using f-mono by (rule extension-lemma)
  show cont: cont (λx. lub (f ` rep x))
    apply (rule contI2)
     apply (rule monofunI)
     apply (rule is-lub-thelub-ex [OF lub ub-imageI])
     apply (rule is-ub-thelub-ex [OF lub imageI])
     apply (erule (1) subsetD [OF rep-mono])
    apply (rule is-lub-thelub-ex [OF lub ub-imageI])
    apply (simp add: rep-lub, clarify)
    apply (erule rev-below-trans [OF is-ub-thelub])
    apply (erule is-ub-thelub-ex [OF lub imageI])
    done
qed

lemma extension-principal:
  fixes f :: 'a::type ⇒ 'c
  assumes f-mono: ⋀a b. a ≼ b ⟹ f a ⊑ f b
  shows extension f·(principal a) = f a
apply (subst extension-beta, erule f-mono)
apply (subst rep-principal)
apply (rule lub-eqI)
apply (rule is-lub-maximal)
apply (rule ub-imageI)
apply (simp add: f-mono)
apply (rule imageI)
apply (simp add: r-refl)
done

lemma extension-mono:
  assumes f-mono: ⋀a b. a ≼ b ⟹ f a ⊑ f b
  assumes g-mono: ⋀a b. a ≼ b ⟹ g a ⊑ g b
  assumes below: ⋀a. f a ⊑ g a
  shows extension f ⊑ extension g
  apply (rule cfun-belowI)
  apply (simp only: extension-beta f-mono g-mono)
  apply (rule is-lub-thelub-ex)
   apply (rule extension-lemma, erule f-mono)
  apply (rule ub-imageI)
  subgoal for x a
```

```
    apply (rule below-trans [OF below])
    apply (rule is-ub-thelub-ex)
     apply (rule extension-lemma, erule g-mono)
    apply (erule imageI)
    done
  done

lemma cont-extension:
  assumes f-mono: ⋀a b x. a ≼ b ⟹ f x a ⊑ f x b
  assumes f-cont: ⋀a. cont (λx. f x a)
  shows cont (λx. extension (λa. f x a))
 apply (rule contI2)
  apply (rule monofunI)
  apply (rule extension-mono, erule f-mono, erule f-mono)
  apply (erule cont2monofunE [OF f-cont])
 apply (rule cfun-belowI)
 apply (rule principal-induct, simp)
 apply (simp only: contlub-cfun-fun)
 apply (simp only: extension-principal f-mono)
 apply (simp add: cont2contlubE [OF f-cont])
done

end

lemma (in preorder) typedef-ideal-completion:
  fixes Abs :: 'a set ⇒ 'b
  assumes type: type-definition Rep Abs {S. ideal S}
  assumes below: ⋀x y. x ⊑ y ⟷ Rep x ⊆ Rep y
  assumes principal: ⋀a. principal a = Abs {b. b ≼ a}
  assumes countable: ∃f::'a ⇒ nat. inj f
  shows ideal-completion r principal Rep
proof
  interpret type-definition Rep Abs {S. ideal S} by fact
  fix a b :: 'a and x y :: 'b and Y :: nat ⇒ 'b
  show ideal (Rep x)
    using Rep [of x] by simp
  show chain Y ⟹ Rep (⨆i. Y i) = (⋃i. Rep (Y i))
    using type below by (rule typedef-ideal-rep-lub)
  show Rep (principal a) = {b. b ≼ a}
    by (simp add: principal Abs-inverse ideal-principal)
  show Rep x ⊆ Rep y ⟹ x ⊑ y
    by (simp only: below)
  show ∃f::'a ⇒ nat. inj f
    by (rule countable)
qed

end
```

21 A universal bifinite domain

theory *Universal*
imports *Bifinite Completion HOL−Library.Nat-Bijection*
begin

unbundle *no binomial-syntax*

21.1 Basis for universal domain

21.1.1 Basis datatype

type-synonym *ubasis* = *nat*

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

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

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

lemma *node-inject* [*simp*]:
⟦*finite S*; *finite T*⟧
⟹ *node i a S* = *node j b T* ⟷ *i* = *j* ∧ *a* = *b* ∧ *S* = *T*
unfolding *node-def* **by** (*simp add*: *prod-encode-eq set-encode-eq*)

lemma *node-gt0*: *i* < *node i a S*
unfolding *node-def less-Suc-eq-le*
by (*rule le-prod-encode-1*)

lemma *node-gt1*: *a* < *node i a S*
unfolding *node-def less-Suc-eq-le*
by (*rule order-trans* [*OF le-prod-encode-1 le-prod-encode-2*])

lemma *nat-less-power2*: *n* < *2^n*
by (*fact less-exp*)

lemma *node-gt2*: ⟦*finite S*; *b* ∈ *S*⟧ ⟹ *b* < *node i a S*
unfolding *node-def less-Suc-eq-le set-encode-def*
apply (*rule order-trans* [*OF - le-prod-encode-2*])
apply (*rule order-trans* [*OF - le-prod-encode-2*])
apply (*rule order-trans* [**where** *y*=*sum* ((^) *2*) {*b*}])
apply (*simp add*: *nat-less-power2* [*THEN order-less-imp-le*])
apply (*erule sum-mono2*, *simp*, *simp*)
done

lemma *eq-prod-encode-pairI*:
  $[\![fst\ (prod\text{-}decode\ x) = a;\ snd\ (prod\text{-}decode\ x) = b]\!] \Longrightarrow x = prod\text{-}encode\ (a,\ b)$
  by *auto*

lemma *node-cases*:
  assumes *1*: $x = 0 \Longrightarrow P$
  assumes *2*: $\bigwedge i\ a\ S.\ [\![finite\ S;\ x = node\ i\ a\ S]\!] \Longrightarrow P$
  shows *P*
 apply (*cases x*)
  apply (*erule 1*)
 apply (*rule 2*)
  apply (*rule finite-set-decode*)
 apply (*simp add*: *node-def*)
 apply (*rule eq-prod-encode-pairI* [*OF refl*])
 apply (*rule eq-prod-encode-pairI* [*OF refl refl*])
done

lemma *node-induct*:
  assumes *1*: *P 0*
  assumes *2*: $\bigwedge i\ a\ S.\ [\![P\ a;\ finite\ S;\ \forall\, b{\in}S.\ P\ b]\!] \Longrightarrow P\ (node\ i\ a\ S)$
  shows *P x*
 apply (*induct x rule*: *nat-less-induct*)
 apply (*case-tac n rule*: *node-cases*)
  apply (*simp add*: *1*)
 apply (*simp add*: *2 node-gt1 node-gt2*)
done

21.1.2 Basis ordering

inductive
  ubasis-le :: $nat \Rightarrow nat \Rightarrow bool$
where
  ubasis-le-refl: *ubasis-le a a*
| *ubasis-le-trans*:
    $[\![ubasis\text{-}le\ a\ b;\ ubasis\text{-}le\ b\ c]\!] \Longrightarrow ubasis\text{-}le\ a\ c$
| *ubasis-le-lower*:
    $finite\ S \Longrightarrow ubasis\text{-}le\ a\ (node\ i\ a\ S)$
| *ubasis-le-upper*:
    $[\![finite\ S;\ b \in S;\ ubasis\text{-}le\ a\ b]\!] \Longrightarrow ubasis\text{-}le\ (node\ i\ a\ S)\ b$

lemma *ubasis-le-minimal*: *ubasis-le 0 x*
apply (*induct x rule*: *node-induct*)
apply (*rule ubasis-le-refl*)
apply (*erule ubasis-le-trans*)
apply (*erule ubasis-le-lower*)
done

interpretation *udom*: *preorder ubasis-le*
apply *standard*

```
apply (rule ubasis-le-refl)
apply (erule (1) ubasis-le-trans)
done
```

21.1.3 Generic take function

```
function
  ubasis-until :: (ubasis ⇒ bool) ⇒ ubasis ⇒ ubasis
where
  ubasis-until P 0 = 0
| finite S ⟹ ubasis-until P (node i a S) =
    (if P (node i a S) then node i a S else ubasis-until P a)
   apply clarify
   apply (rule-tac x=b in node-cases)
    apply simp-all
done

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

lemma ubasis-until: P 0 ⟹ P (ubasis-until P x)
by (induct x rule: node-induct) simp-all

lemma ubasis-until': 0 < ubasis-until P x ⟹ P (ubasis-until P x)
by (induct x rule: node-induct) auto

lemma ubasis-until-same: P x ⟹ ubasis-until P x = x
by (induct x rule: node-induct) simp-all

lemma ubasis-until-idem:
  P 0 ⟹ ubasis-until P (ubasis-until P x) = ubasis-until P x
by (rule ubasis-until-same [OF ubasis-until])

lemma ubasis-until-0:
  ∀ x. x ≠ 0 ⟶ ¬ P x ⟹ ubasis-until P x = 0
by (induct x rule: node-induct) simp-all

lemma ubasis-until-less: ubasis-le (ubasis-until P x) x
apply (induct x rule: node-induct)
apply (simp add: ubasis-le-refl)
 by (metis ubasis-le.simps ubasis-until.simps(2))

lemma ubasis-until-chain:
  assumes PQ: ⋀x. P x ⟹ 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*)
  by (*metis assms ubasis-until.simps*(*2*) *ubasis-until-less*)

lemma *ubasis-until-mono*:
  assumes $\bigwedge i\ a\ S\ b.\ [\![finite\ S;\ P\ (node\ i\ a\ S);\ b \in S;\ ubasis\text{-}le\ a\ b]\!] \Longrightarrow P\ b$
  shows $ubasis\text{-}le\ a\ b \Longrightarrow ubasis\text{-}le\ (ubasis\text{-}until\ P\ a)\ (ubasis\text{-}until\ P\ b)$
proof (*induct set*: *ubasis-le*)
  case (*ubasis-le-refl a*) **show** *?case* **by** (*rule ubasis-le.ubasis-le-refl*)
next
  case (*ubasis-le-trans a b c*) **thus** *?case* **by** − (*rule ubasis-le.ubasis-le-trans*)
next
  case (*ubasis-le-lower S a i*) **thus** *?case*
    by (*metis ubasis-le.simps ubasis-until.simps*(*2*) *ubasis-until-less*)
next
  case (*ubasis-le-upper S b a i*) **thus** *?case*
    by (*metis assms ubasis-le.simps ubasis-until.simps*(*2*) *ubasis-until-same*)
qed

lemma *finite-range-ubasis-until*:
  $finite\ \{x.\ P\ x\} \Longrightarrow finite\ (range\ (ubasis\text{-}until\ P))$
apply (*rule finite-subset* [**where** $B{=}insert\ 0\ \{x.\ P\ x\}$])
apply (*clarsimp simp add*: *ubasis-until′*)
apply *simp*
done

21.2 Defining the universal domain by ideal completion

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

instantiation *udom* :: *below*
begin

definition
  $x \sqsubseteq y \longleftrightarrow Rep\text{-}udom\ x \subseteq Rep\text{-}udom\ y$

instance ..
end

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

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

definition
  udom-principal :: $nat \Rightarrow udom$ **where**

udom-principal t = *Abs-udom* $\{u.$ *ubasis-le* $u\ t\}$

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

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

Universal domain is pointed

lemma *udom-minimal*: *udom-principal 0* $\sqsubseteq x$
apply (*induct x rule*: *udom.principal-induct*)
apply (*simp*, *simp add*: *ubasis-le-minimal*)
done

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

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

21.3 Compact bases of domains

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

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

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

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

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

instance ..
end

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

definition
  approximants :: *'a::pcpo* ⇒ *'a compact-basis set* **where**
  approximants = (λx. {*a*. *Rep-compact-basis a* ⊑ *x*})

definition
  compact-bot :: *'a::pcpo compact-basis* **where**
  compact-bot = *Abs-compact-basis* ⊥

lemma *Rep-compact-bot* [*simp*]: *Rep-compact-basis compact-bot* = ⊥
unfolding *compact-bot-def* **by** *simp*

lemma *compact-bot-minimal* [*simp*]: *compact-bot* ⊑ *a*
unfolding *compact-le-def Rep-compact-bot* **by** *simp*

21.4 Universality of *udom*

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

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

21.4.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
  fixes *A* :: *'a compact-basis set*
  shows ⟦*finite A*; *A* ≠ {}⟧ ⟹ ∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*
proof (*induct rule*: *finite-ne-induct*)
  case (*singleton x*)
    show *?case* **by** *simp*
next
  case (*insert a A*)
  from ‹∃ *x*∈*A*. ∀ *y*∈*A*. *x* ⊑ *y* ⟶ *x* = *y*›
  obtain *x* **where** *x*: *x* ∈ *A*
          and *x-eq*: ⋀*y*. ⟦*y* ∈ *A*; *x* ⊑ *y*⟧ ⟹ *x* = *y* **by** *fast*
  show *?case*
  proof (*intro bexI ballI impI*)
    fix *y*
    assume *y* ∈ *insert a A* **and** (*if x* ⊑ *a then a else x*) ⊑ *y*
    thus (*if x* ⊑ *a then a else x*) = *y*
      apply *auto*
      apply (*frule* (*1*) *below-trans*)
      apply (*frule* (*1*) *x-eq*)
      apply (*rule below-antisym*, *assumption*)
      apply *simp*
      apply (*erule* (*1*) *x-eq*)
      done
  next
    show (*if x* ⊑ *a then a else x*) ∈ *insert a A*

```
    by (simp add: x)
  qed
qed

definition
  choose :: 'a compact-basis set ⇒ 'a compact-basis
where
  choose A = (SOME x. x ∈ {x∈A. ∀ y∈A. x ⊑ y ⟶ x = y})

lemma choose-lemma:
  ⟦finite A; A ≠ {}⟧ ⟹ choose A ∈ {x∈A. ∀ y∈A. x ⊑ y ⟶ x = y}
unfolding choose-def
apply (rule someI-ex)
apply (frule (1) finite-has-maximal, fast)
done

lemma maximal-choose:
  ⟦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: if-split-asm)
 apply (erule (1) inj-onD, simp, simp)
done

lemma choose-pos-bounded [OF refl]:
  ⟦card A = n; finite A; x ∈ A⟧ ⟹ choose-pos A x < n
apply (induct n arbitrary: A)
apply simp
 apply (case-tac A = {}, simp)
 apply (frule (1) choose-in)
apply (subst choose-pos.simps)
apply simp
done

lemma choose-pos-lessD:
  ⟦choose-pos A x < choose-pos A y; finite A; x ∈ A; y ∈ A⟧ ⟹ x ⋢ y
 apply (induct A x arbitrary: y rule: choose-pos.induct)
 apply simp
 apply (case-tac x = choose A)
  apply simp
  apply (rule notI)
  apply (frule (2) maximal-choose)
  apply simp
 apply (case-tac y = choose A)
  apply (simp add: choose-pos-choose)
 apply (drule-tac x=y in meta-spec)
 apply simp
 apply (erule meta-mp)
 apply (simp add: choose-pos.simps)
done
```

21.4.2 Compact basis take function

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

declare cb-take.simps [simp del]

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

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

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

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

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

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

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

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

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

21.4.3 Rank of basis elements

definition
  rank :: ′*a compact-basis* ⇒ *nat*
where
  rank x = (*LEAST n*. *cb-take n x* = *x*)

lemma *compact-approx-rank*: *cb-take* (*rank x*) *x* = *x*
unfolding *rank-def*
apply (*rule LeastI-ex*)
apply (*rule cb-take-covers*)
done

lemma *rank-leD*: *rank x* ≤ *n* ⟹ *cb-take n x* = *x*
apply (*rule below-antisym* [*OF cb-take-less*])
apply (*subst compact-approx-rank* [*symmetric*])
apply (*erule cb-take-chain-le*)
done

lemma *rank-leI*: *cb-take n x* = *x* ⟹ *rank x* ≤ *n*
unfolding *rank-def* **by** (*rule Least-le*)

lemma *rank-le-iff*: *rank x* ≤ *n* ⟷ *cb-take n x* = *x*
by (*rule iffI* [*OF rank-leD rank-leI*])

lemma *rank-compact-bot* [*simp*]: *rank compact-bot* = *0*
using *rank-leI* [*of 0 compact-bot*] **by** *simp*

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

definition
  rank-le :: *'a compact-basis* ⇒ *'a compact-basis set*
where
  rank-le x = {*y*. *rank y* ≤ *rank x*}

definition
  rank-lt :: *'a compact-basis* ⇒ *'a compact-basis set*
where
  rank-lt x = {*y*. *rank y* < *rank x*}

definition
  rank-eq :: *'a compact-basis* ⇒ *'a compact-basis set*
where
  rank-eq x = {*y*. *rank y* = *rank x*}

lemma *rank-eq-cong*: *rank x* = *rank y* ⟹ *rank-eq x* = *rank-eq y*
unfolding *rank-eq-def* **by** *simp*

lemma *rank-lt-cong*: *rank x* = *rank y* ⟹ *rank-lt x* = *rank-lt y*
unfolding *rank-lt-def* **by** *simp*

lemma *rank-eq-subset*: *rank-eq x* ⊆ *rank-le x*
unfolding *rank-eq-def rank-le-def* **by** *auto*

lemma *rank-lt-subset*: *rank-lt x* ⊆ *rank-le x*
unfolding *rank-lt-def rank-le-def* **by** *auto*

lemma *finite-rank-le*: *finite* (*rank-le x*)
unfolding *rank-le-def*
apply (*rule finite-subset* [**where** *B*=*range* (*cb-take* (*rank x*))])
apply *clarify*
apply (*rule range-eqI*)
apply (*erule rank-leD* [*symmetric*])
apply (*rule finite-range-cb-take*)
done

lemma *finite-rank-eq*: *finite* (*rank-eq x*)
by (*rule finite-subset* [*OF rank-eq-subset finite-rank-le*])

lemma *finite-rank-lt*: *finite* (*rank-lt x*)
by (*rule finite-subset* [*OF rank-lt-subset finite-rank-le*])

lemma *rank-lt-Int-rank-eq*: *rank-lt x* ∩ *rank-eq x* = {}
unfolding *rank-lt-def rank-eq-def rank-le-def* **by** *auto*

lemma *rank-lt-Un-rank-eq*: *rank-lt x* ∪ *rank-eq x* = *rank-le x*
unfolding *rank-lt-def rank-eq-def rank-le-def* **by** *auto*

21.4.4 Sequencing basis elements

definition
  place :: *'a compact-basis* ⇒ *nat*
where
  place x = *card* (*rank-lt x*) + *choose-pos* (*rank-eq x*) *x*

lemma *place-bounded*: *place x* < *card* (*rank-le x*)
unfolding *place-def*
 apply (*rule ord-less-eq-trans*)
  apply (*rule add-strict-left-mono*)
  apply (*rule choose-pos-bounded*)
   apply (*rule finite-rank-eq*)
  apply (*simp add*: *rank-eq-def*)
 apply (*subst card-Un-disjoint* [*symmetric*])
    apply (*rule finite-rank-lt*)
   apply (*rule finite-rank-eq*)
  apply (*rule rank-lt-Int-rank-eq*)
 apply (*simp add*: *rank-lt-Un-rank-eq*)
done

lemma *place-ge*: *card* (*rank-lt x*) ≤ *place x*
unfolding *place-def* **by** *simp*

lemma *place-rank-mono*:

fixes *x y* :: *'a compact-basis*
shows *rank x* < *rank y* ⟹ *place x* < *place y*
apply (*rule less-le-trans* [*OF place-bounded*])
apply (*rule order-trans* [*OF - place-ge*])
apply (*rule card-mono*)
apply (*rule finite-rank-lt*)
apply (*simp add*: *rank-le-def rank-lt-def subset-eq*)
done

lemma *place-eqD*: *place x* = *place y* ⟹ *x* = *y*
apply (*rule linorder-cases* [**where** *x*=*rank x* **and** *y*=*rank y*])
apply (*drule place-rank-mono*, *simp*)
apply (*simp add*: *place-def*)
apply (*rule inj-on-choose-pos* [**where** *A*=*rank-eq x*, *THEN inj-onD*])
apply (*rule finite-rank-eq*)
apply (*simp cong*: *rank-lt-cong rank-eq-cong*)
apply (*simp add*: *rank-eq-def*)
apply (*simp add*: *rank-eq-def*)
apply (*drule place-rank-mono*, *simp*)
done

lemma *inj-place*: *inj place*
by (*rule inj-onI*, *erule place-eqD*)

21.4.5 Embedding and projection on basis elements

definition
sub :: *'a compact-basis* ⇒ *'a compact-basis*
where
sub x = (*case rank x of 0* ⇒ *compact-bot* | *Suc k* ⇒ *cb-take k x*)

lemma *rank-sub-less*: *x* ≠ *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 simp-all

termination basis-emb
  by (relation measure place) (simp-all add: place-sub-less)

declare basis-emb.simps [simp del]

lemma basis-emb-compact-bot [simp]:
  basis-emb compact-bot = 0
  using basis-emb.simps [of compact-bot] by simp

lemma basis-emb-rec:
  basis-emb x = node (place x) (basis-emb (sub x)) (basis-emb ‘ {y. place y < place
x ∧ x ⊑ y})
  if x ≠ compact-bot
  using that basis-emb.simps [of x] by simp

lemma basis-emb-eq-0-iff [simp]:
  basis-emb x = 0 ⟷ x = compact-bot
  by (cases x = compact-bot) (simp-all add: basis-emb-rec)

lemma fin1: finite {y. place y < place x ∧ x ⊑ y}
apply (subst Collect-conj-eq)
apply (rule finite-Int)
apply (rule disjI1)
apply (subgoal-tac finite (place −‘ {n. n < place x}), simp)
apply (rule finite-vimageI [OF - inj-place])
apply (simp add: lessThan-def [symmetric])
done

lemma fin2: finite (basis-emb ‘ {y. place y < place x ∧ x ⊑ y})
by (rule finite-imageI [OF fin1])

lemma rank-place-mono:
  ⟦place x < place y; x ⊑ y⟧ ⟹ rank x < rank y
apply (rule linorder-cases, assumption)
apply (simp add: place-def cong: rank-lt-cong rank-eq-cong)
apply (drule choose-pos-lessD)
```

```
apply (rule finite-rank-eq)
apply (simp add: rank-eq-def)
apply (simp add: rank-eq-def)
apply simp
apply (drule place-rank-mono, simp)
done

lemma basis-emb-mono:
  x ⊑ y ⟹ ubasis-le (basis-emb x) (basis-emb y)
proof (induct max (place x) (place y) arbitrary: x y rule: less-induct)
  case less
  show ?case proof (rule linorder-cases)
    assume place x < place y
    then have rank x < rank y
      using ‹x ⊑ y› by (rule rank-place-mono)
    with ‹place x < place y› show ?case
      apply (case-tac y = compact-bot, simp)
      apply (simp add: basis-emb.simps [of y])
      apply (rule ubasis-le-trans [OF - ubasis-le-lower [OF fin2]])
      apply (rule less)
       apply (simp add: less-max-iff-disj)
       apply (erule place-sub-less)
      apply (erule rank-less-imp-below-sub [OF ‹x ⊑ y›])
      done
  next
    assume place x = place y
    hence x = y by (rule place-eqD)
    thus ?case by (simp add: ubasis-le-refl)
  next
    assume place x > place y
    with ‹x ⊑ y› show ?case
      apply (case-tac x = compact-bot, simp add: ubasis-le-minimal)
      apply (simp add: basis-emb.simps [of x])
      apply (rule ubasis-le-upper [OF fin2], simp)
      apply (rule less)
       apply (simp add: less-max-iff-disj)
       apply (erule place-sub-less)
      apply (erule rev-below-trans)
      apply (rule sub-below)
      done
  qed
qed

lemma inj-basis-emb: inj basis-emb
proof (rule injI)
  fix x y
  assume basis-emb x = basis-emb y
  then show x = y
    by (cases x = compact-bot ∨ y = compact-bot) (auto simp add: basis-emb-rec
```

```
fin2 place-eqD)
qed

definition
  basis-prj :: ubasis ⇒ 'a compact-basis
where
  basis-prj x = inv basis-emb
    (ubasis-until (λx. x ∈ range (basis-emb :: 'a compact-basis ⇒ ubasis)) x)

lemma basis-prj-basis-emb: ⋀x. basis-prj (basis-emb x) = x
unfolding basis-prj-def
 apply (subst ubasis-until-same)
  apply (rule rangeI)
 apply (rule inv-f-f)
 apply (rule inj-basis-emb)
done

lemma basis-prj-node:
  ⟦finite S; node i a S ∉ range (basis-emb :: 'a compact-basis ⇒ nat)⟧
    ⟹ basis-prj (node i a S) = (basis-prj a :: 'a compact-basis)
unfolding basis-prj-def by simp

lemma basis-prj-0: basis-prj 0 = compact-bot
apply (subst basis-emb-compact-bot [symmetric])
apply (rule basis-prj-basis-emb)
done

lemma node-eq-basis-emb-iff:
  finite S ⟹ node i a S = basis-emb x ⟷
    x ≠ compact-bot ∧ i = place x ∧ a = basis-emb (sub x) ∧
        S = basis-emb ` {y. place y < place x ∧ x ⊑ y}
apply (cases x = compact-bot, simp)
apply (simp add: basis-emb.simps [of x])
apply (simp add: fin2)
done

lemma basis-prj-mono: ubasis-le a b ⟹ basis-prj a ⊑ basis-prj b
proof (induct a b rule: ubasis-le.induct)
  case (ubasis-le-refl a) show ?case by (rule below-refl)
next
  case (ubasis-le-trans a b c) thus ?case by − (rule below-trans)
next
  case (ubasis-le-lower S a i) thus ?case
    apply (cases node i a S ∈ range (basis-emb :: 'a compact-basis ⇒ nat))
     apply (erule rangeE, rename-tac x)
     apply (simp add: basis-prj-basis-emb)
     apply (simp add: node-eq-basis-emb-iff)
     apply (simp add: basis-prj-basis-emb)
     apply (rule sub-below)
```

```
    apply (simp add: basis-prj-node)
    done
next
  case (ubasis-le-upper S b a i) thus ?case
    apply (cases node i a S ∈ range (basis-emb :: 'a compact-basis ⇒ nat))
     apply (erule rangeE, rename-tac x)
     apply (simp add: basis-prj-basis-emb)
     apply (clarsimp simp add: node-eq-basis-emb-iff)
     apply (simp add: basis-prj-basis-emb)
    apply (simp add: basis-prj-node)
    done
qed

lemma basis-emb-prj-less: ubasis-le (basis-emb (basis-prj x)) x
unfolding basis-prj-def
 apply (subst f-inv-into-f [where f=basis-emb])
  apply (rule ubasis-until)
  apply (rule range-eqI [where x=compact-bot])
  apply simp
 apply (rule ubasis-until-less)
done

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

```
    by (auto elim: below-trans)
  qed
next
  fix Y :: nat ⇒ 'a
  assume chain Y
  thus approximants (⨆ i. Y i) = (⋃ i. approximants (Y i))
    unfolding approximants-def
    by (auto simp add: compact-below-lub-iff)
next
  fix a :: 'a compact-basis
  show approximants (Rep-compact-basis a) = {b. b ⊑ a}
    unfolding approximants-def compact-le-def ..
next
  fix x y :: 'a
  assume approximants x ⊆ approximants y
  hence ∀ z. compact z ⟶ z ⊑ x ⟶ z ⊑ y
    by (simp add: approximants-def subset-eq)
      (metis Abs-compact-basis-inverse')
  hence (⨆ i. approx i·x) ⊑ y
    by (simp add: lub-below approx-below)
  thus x ⊑ y
    by (simp add: lub-distribs)
next
  show ∃f::'a compact-basis ⇒ nat. inj f
    by (rule exI, rule inj-place)
qed

end

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

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

```
context bifinite-approx-chain begin

definition
  udom-emb :: 'a → udom
where
  udom-emb = compact-basis.extension (λx. udom-principal (basis-emb x))
```

definition
udom-prj :: *udom* → *'a*
where
udom-prj = *udom.extension* (*λx. Rep-compact-basis* (*basis-prj x*))

lemma *udom-emb-principal*:
udom-emb·(*Rep-compact-basis x*) = *udom-principal* (*basis-emb x*)
unfolding *udom-emb-def*
apply (*rule compact-basis.extension-principal*)
apply (*rule udom.principal-mono*)
apply (*erule basis-emb-mono*)
done

lemma *udom-prj-principal*:
udom-prj·(*udom-principal x*) = *Rep-compact-basis* (*basis-prj x*)
unfolding *udom-prj-def*
apply (*rule udom.extension-principal*)
apply (*rule compact-basis.principal-mono*)
apply (*erule basis-prj-mono*)
done

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
apply *standard*
apply (*rule compact-basis.principal-induct*, *simp*)
apply (*simp add*: *udom-emb-principal udom-prj-principal*)
apply (*simp add*: *basis-prj-basis-emb*)
apply (*rule udom.principal-induct*, *simp*)
apply (*simp add*: *udom-emb-principal udom-prj-principal*)
apply (*rule basis-emb-prj-less*)
done

end

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

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

21.5 Chain of approx functions for type *udom*

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

lemma *udom-approx-mono*:

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

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

lemma udom-approx-principal:
  udom-approx i·(udom-principal x) =
    udom-principal (ubasis-until (λy. y ≤ i) x)
unfolding udom-approx-def
apply (rule udom.extension-principal)
apply (erule udom-approx-mono)
done

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

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

lemma chain-udom-approx [simp]: chain (λi. udom-approx i)
```

unfolding *udom-approx-def*
apply (*rule chainI*)
apply (*rule udom.extension-mono*)
apply (*erule udom-approx-mono*)
apply (*erule udom-approx-mono*)
apply (*rule udom.principal-mono*)
apply (*rule ubasis-until-chain*, *simp*)
done

lemma *lub-udom-approx* [*simp*]: ($\bigsqcup$ *i. udom-approx i*) = *ID*
apply (*rule cfun-eqI*, *simp add*: *contlub-cfun-fun*)
apply (*rule below-antisym*)
apply (*rule lub-below*)
apply (*simp*)
apply (*rule udom-approx.below*)
apply (*rule-tac x=x* **in** *udom.principal-induct*)
apply (*simp add*: *lub-distribs*)
apply (*rule-tac i=a* **in** *below-lub*)
apply *simp*
apply (*simp add*: *udom-approx-principal*)
apply (*simp add*: *ubasis-until-same ubasis-le-refl*)
done

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

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

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

unbundle *binomial-syntax*

end

22 Algebraic deflations

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

22.1 Type constructor for finite deflations

typedef *'a::bifinite fin-defl* = {*d::'a → 'a. finite-deflation d*}

by (*fast intro*: *finite-deflation-bottom*)

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

definition *below-fin-defl-def*:
  below $\equiv$ $\lambda x\ y$. *Rep-fin-defl* $x \sqsubseteq$ *Rep-fin-defl* y

instance ..
end

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

lemma *finite-deflation-Rep-fin-defl*: *finite-deflation* (*Rep-fin-defl d*)
using *Rep-fin-defl* **by** *simp*

lemma *deflation-Rep-fin-defl*: *deflation* (*Rep-fin-defl d*)
using *finite-deflation-Rep-fin-defl*
by (*rule finite-deflation-imp-deflation*)

interpretation *Rep-fin-defl*: *finite-deflation Rep-fin-defl d*
by (*rule finite-deflation-Rep-fin-defl*)

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

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

lemma *fin-defl-eqI*:
  $a = b$ **if** ($\bigwedge x$. *Rep-fin-defl* $a \cdot x = x \longleftrightarrow$ *Rep-fin-defl* $b \cdot x = x$)
proof (*rule below-antisym*)
  show $a \sqsubseteq b$ **by** (*rule fin-defl-belowI*) (*simp add*: *that*)
  show $b \sqsubseteq a$ **by** (*rule fin-defl-belowI*) (*simp add*: *that*)
qed

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

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 *below-fin-defl-def*

by (*simp add*: *Abs-fin-defl-inverse*)

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

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

22.2 Defining algebraic deflations by ideal completion

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

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

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

instance ..

end

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

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

definition *defl-principal* :: *'a::bifinite fin-defl* $\Rightarrow$ *'a defl*
  where *defl-principal t* = *Abs-defl* {*u. u* $\sqsubseteq$ *t*}

lemma *fin-defl-countable*: $\exists$*f::'a::bifinite fin-defl* $\Rightarrow$ *nat. inj f*
proof −
  obtain *f* :: *'a compact-basis* $\Rightarrow$ *nat* **where** *inj-f*: *inj f*
    using *compact-basis.countable* **..**
  have $*$: $\bigwedge$*d. finite* (*f* ' *Rep-compact-basis* −' {*x. Rep-fin-defl d*·*x* = *x*})
    apply (*rule finite-imageI*)
    apply (*rule finite-vimageI*)
    apply (*rule Rep-fin-defl.finite-fixes*)
    apply (*simp add*: *inj-on-def Rep-compact-basis-inject*)
    done
  have *range-eq*: *range Rep-compact-basis* = {*x. compact x*}
    using *type-definition-compact-basis* **by** (*rule type-definition.Rep-range*)
  have *inj* (λ*d. set-encode*
    (*f* ' *Rep-compact-basis* −' {*x. Rep-fin-defl d*·*x* = *x*}))

```
  apply (rule inj-onI)
  apply (simp only: set-encode-eq ∗)
  apply (simp only: inj-image-eq-iff inj-f)
  apply (drule-tac f=image Rep-compact-basis in arg-cong)
  apply (simp del: vimage-Collect-eq add: range-eq set-eq-iff)
  apply (rule Rep-fin-defl-inject [THEN iffD1])
  apply (rule below-antisym)
  apply (rule Rep-fin-defl.compact-belowI, rename-tac z)
  apply (drule-tac x=z in spec, simp)
  apply (rule Rep-fin-defl.compact-belowI, rename-tac z)
  apply (drule-tac x=z in spec, simp)
  done
 thus ?thesis by − (rule exI)
qed

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

Algebraic deflations are pointed

```
lemma defl-minimal: defl-principal (Abs-fin-defl ⊥) ⊑ x
proof (induct x rule: defl.principal-induct)
 fix a :: 'a fin-defl
 have Abs-fin-defl ⊥ ⊑ a
  by (simp add: below-fin-defl-def Abs-fin-defl-inverse finite-deflation-bottom)
 then show defl-principal (Abs-fin-defl ⊥) ⊑ defl-principal a
  by (rule defl.principal-mono)
qed simp

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

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

22.3 Applying algebraic deflations

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

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

lemma deflation-cast: deflation (cast·d)
apply (induct d rule: defl.principal-induct)
apply (rule adm-subst [OF - adm-deflation], simp)
apply (simp add: cast-defl-principal)
```

apply (*rule finite-deflation-imp-deflation*)
apply (*rule finite-deflation-Rep-fin-defl*)
done

lemma *finite-deflation-cast*: *compact d* $\Longrightarrow$ *finite-deflation* (*cast·d*)
  apply (*drule defl.compact-imp-principal*)
  apply *clarify*
  apply (*simp add*: *cast-defl-principal*)
  apply (*rule finite-deflation-Rep-fin-defl*)
  done

interpretation *cast*: *deflation cast·d*
by (*rule deflation-cast*)

declare *cast.idem* [*simp*]

lemma *compact-cast* [*simp*]: *compact* (*cast·d*) **if** *compact d*
  by (*rule finite-deflation-imp-compact*) (*use that* **in** ‹*rule finite-deflation-cast*›)

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

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

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

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

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

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

22.4 Deflation combinators

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

definition
  defl-fun2 e p f =
    defl.extension (λ*a*.
      defl.extension (λ*b*.
        defl-principal (*Abs-fin-defl*
          (*e oo f*·(*Rep-fin-defl a*)·(*Rep-fin-defl b*) *oo p*))))

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

lemma *cast-defl-fun2*:
  assumes *ep*: *ep-pair e p*
  assumes *f*: ⋀*a b*. *finite-deflation a* ⟹ *finite-deflation b* ⟹
              finite-deflation (*f*·*a*·*b*)
  shows *cast*·(*defl-fun2 e p f*·*A*·*B*) = *e oo f*·(*cast*·*A*)·(*cast*·*B*) *oo p*
proof −
  have *1*: *finite-deflation* (*e oo f*·(*Rep-fin-defl a*)·(*Rep-fin-defl b*) *oo p*) **for** *a b*
  proof −
    have *finite-deflation* (*f*·(*Rep-fin-defl a*)·(*Rep-fin-defl b*))
      using *finite-deflation-Rep-fin-defl finite-deflation-Rep-fin-defl* **by** (*rule f*)

```
  with ep show ?thesis
    by (rule ep-pair.finite-deflation-e-d-p)
 qed
 show ?thesis
  apply (induct A rule: defl.principal-induct, simp)
  apply (induct B rule: defl.principal-induct, simp)
  by (simp only: defl-fun2-def
               defl.extension-principal
               defl.extension-mono
               defl.principal-mono
               Abs-fin-defl-mono [OF 1 1]
               monofun-cfun below-refl
               Rep-fin-defl-mono
               cast-defl-principal
               Abs-fin-defl-inverse [unfolded mem-Collect-eq, OF 1])
qed

end
```

23 Representable domains

```
theory Representable
imports Algebraic Map-Functions HOL−Library.Countable
begin
```

23.1 Class of representable domains

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

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

```
class predomain-syn = cpo +
 fixes liftemb :: 'a⊥ → udom⊥
 fixes liftprj :: udom⊥ → 'a⊥
 fixes liftdefl :: 'a itself ⇒ udom u defl

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

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

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

lemma cast-liftdefl-of: cast·(liftdefl-of·t) = u-map·(cast·t)
```

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

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

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

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

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

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

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

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

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

23.2 Domains are bifinite

lemma *approx-chain-ep-cast*:

```
  assumes ep: ep-pair (e::'a::pcpo → 'b::bifinite) (p::'b → 'a)
  assumes cast-t: cast·t = e oo p
  shows ∃(a::nat ⇒ 'a::pcpo → 'a). approx-chain a
proof –
  interpret ep-pair e p by fact
  obtain Y where Y: ∀ i. Y i ⊑ Y (Suc i)
  and t: t = (⨆ i. defl-principal (Y i))
    by (rule defl.obtain-principal-chain)
  define approx where approx i = (p oo cast·(defl-principal (Y i)) oo e) for i
  have approx-chain approx
  proof (rule approx-chain.intro)
    show chain (λi. approx i)
      unfolding approx-def by (simp add: Y)
    show (⨆ i. approx i) = ID
      unfolding approx-def
      by (simp add: lub-distribs Y t [symmetric] cast-t cfun-eq-iff)
    show ⋀i. finite-deflation (approx i)
      unfolding approx-def
      apply (rule finite-deflation-p-d-e)
      apply (rule finite-deflation-cast)
      apply (rule defl.compact-principal)
      apply (rule below-trans [OF monofun-cfun-fun])
      apply (rule is-ub-thelub, simp add: Y)
      apply (simp add: lub-distribs Y t [symmetric] cast-t)
      done
  qed
  thus ∃(a::nat ⇒ 'a → 'a). approx-chain a by – (rule exI)
qed

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

instance predomain ⊆ profinite
by standard (rule approx-chain-ep-cast [OF predomain-ep cast-liftdefl])
```

23.3 Universal domain ep-pairs

```
definition u-emb = udom-emb (λi. u-map·(udom-approx i))
definition u-prj = udom-prj (λi. u-map·(udom-approx i))

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

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

definition ssum-emb = udom-emb (λi. ssum-map·(udom-approx i)·(udom-approx
```

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

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

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

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

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

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

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

23.4 Type combinators

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

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

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

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

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

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

lemma *cast-prod-defl*:

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

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

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

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

Special deflation combinator for unpointed types.

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

lemma *cast-u-liftdefl*:
cast·(*u-liftdefl*·*A*) = *u-emb oo cast*·*A oo u-prj*
unfolding *u-liftdefl-def* **by** (*simp add*: *cast-defl-fun1 ep-pair-u*)

lemma *u-liftdefl-liftdefl-of*:
u-liftdefl·(*liftdefl-of*·*A*) = *u-defl*·*A*
by (*rule cast-eq-imp-eq*)
(*simp add*: *cast-u-liftdefl cast-liftdefl-of cast-u-defl*)

23.5 Class instance proofs

23.5.1 Universal domain

instantiation *udom* :: *domain*
begin

definition [*simp*]:
emb = (*ID* :: *udom* → *udom*)

definition [*simp*]:
prj = (*ID* :: *udom* → *udom*)

definition

defl (*t*::*udom itself*) = ($\bigsqcup$ *i*. *defl-principal* (*Abs-fin-defl* (*udom-approx i*)))

definition
(*liftemb* :: *udom u* → *udom u*) = *u-map*·*emb*

definition
(*liftprj* :: *udom u* → *udom u*) = *u-map*·*prj*

definition
liftdefl (*t*::*udom itself*) = *liftdefl-of*·*DEFL*(*udom*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → *udom*)
by (*simp add*: *ep-pair.intro*)
show *cast*·*DEFL*(*udom*) = *emb oo* (*prj* :: *udom* → *udom*)
unfolding *defl-udom-def*
apply (*subst contlub-cfun-arg*)
apply (*rule chainI*)
apply (*rule defl.principal-mono*)
apply (*simp add*: *below-fin-defl-def*)
apply (*simp add*: *Abs-fin-defl-inverse finite-deflation-udom-approx*)
apply (*rule chainE*)
apply (*rule chain-udom-approx*)
apply (*subst cast-defl-principal*)
apply (*simp add*: *Abs-fin-defl-inverse finite-deflation-udom-approx*)
done
qed (*fact liftemb-udom-def liftprj-udom-def liftdefl-udom-def*)+

end

23.5.2 Lifted cpo

instantiation *u* :: (*predomain*) *domain*
begin

definition
emb = *u-emb oo liftemb*

definition
prj = *liftprj oo u-prj*

definition
defl (*t*::'*a u itself*) = *u-liftdefl*·*LIFTDEFL*('*a*)

definition
(*liftemb* :: '*a u u* → *udom u*) = *u-map*·*emb*

definition
(*liftprj* :: *udom u* → '*a u u*) = *u-map*·*prj*

definition
liftdefl (*t*::′*a u itself*) = *liftdefl-of*·*DEFL*(′*a u*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → ′*a u*)
unfolding *emb-u-def prj-u-def*
by (*intro ep-pair-comp ep-pair-u predomain-ep*)
show *cast*·*DEFL*(′*a u*) = *emb oo* (*prj* :: *udom* → ′*a u*)
unfolding *emb-u-def prj-u-def defl-u-def*
by (*simp add*: *cast-u-liftdefl cast-liftdefl assoc-oo*)
qed (*fact liftemb-u-def liftprj-u-def liftdefl-u-def*)+

end

lemma *DEFL-u*: *DEFL*(′*a*::*predomain u*) = *u-liftdefl*·*LIFTDEFL*(′*a*)
by (*rule defl-u-def*)

23.5.3 Strict function space

instantiation *sfun* :: (*domain*, *domain*) *domain*
begin

definition
emb = *sfun-emb oo sfun-map*·*prj*·*emb*

definition
prj = *sfun-map*·*emb*·*prj oo sfun-prj*

definition
defl (*t*::(′*a* →! ′*b*) *itself*) = *sfun-defl*·*DEFL*(′*a*)·*DEFL*(′*b*)

definition
(*liftemb* :: (′*a* →! ′*b*) *u* → *udom u*) = *u-map*·*emb*

definition
(*liftprj* :: *udom u* → (′*a* →! ′*b*) *u*) = *u-map*·*prj*

definition
liftdefl (*t*::(′*a* →! ′*b*) *itself*) = *liftdefl-of*·*DEFL*(′*a* →! ′*b*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → ′*a* →! ′*b*)
unfolding *emb-sfun-def prj-sfun-def*
by (*intro ep-pair-comp ep-pair-sfun ep-pair-sfun-map ep-pair-emb-prj*)
show *cast*·*DEFL*(′*a* →! ′*b*) = *emb oo* (*prj* :: *udom* → ′*a* →! ′*b*)
unfolding *emb-sfun-def prj-sfun-def defl-sfun-def cast-sfun-defl*
by (*simp add*: *cast-DEFL oo-def sfun-eq-iff sfun-map-map*)
qed (*fact liftemb-sfun-def liftprj-sfun-def liftdefl-sfun-def*)+

end

lemma *DEFL-sfun*:
  DEFL('a::domain →! 'b::domain) = sfun-defl·DEFL('a)·DEFL('b)
by (*rule defl-sfun-def*)

23.5.4 Continuous function space

instantiation *cfun* :: (*predomain*, *domain*) *domain*
begin

definition
  emb = emb oo encode-cfun

definition
  prj = decode-cfun oo prj

definition
  defl (t::('a → 'b) itself) = DEFL('a u →! 'b)

definition
  (*liftemb* :: (*'a → 'b*) *u → udom u*) = *u-map·emb*

definition
  (*liftprj* :: *udom u →* (*'a → 'b*) *u*) = *u-map·prj*

definition
  liftdefl (t::('a → 'b) itself) = liftdefl-of·DEFL('a → 'b)

instance proof
  have *ep-pair encode-cfun decode-cfun*
    by (*rule ep-pair.intro*, *simp-all*)
  thus *ep-pair emb* (*prj* :: *udom → 'a → 'b*)
    unfolding *emb-cfun-def prj-cfun-def*
    using *ep-pair-emb-prj* **by** (*rule ep-pair-comp*)
  show *cast·DEFL('a → 'b) = emb oo* (*prj* :: *udom → 'a → 'b*)
    unfolding *emb-cfun-def prj-cfun-def defl-cfun-def*
    by (*simp add*: *cast-DEFL cfcomp1*)
qed (*fact liftemb-cfun-def liftprj-cfun-def liftdefl-cfun-def*)+

end

lemma *DEFL-cfun*:
  DEFL('a::predomain → 'b::domain) = DEFL('a u →! 'b)
by (*rule defl-cfun-def*)

23.5.5 Strict product

instantiation *sprod* :: (*domain*, *domain*) *domain*

begin

definition
emb = sprod-emb oo sprod-map·emb·emb

definition
prj = sprod-map·prj·prj oo sprod-prj

definition
defl (t::('a ⊗ 'b) itself) = sprod-defl·DEFL('a)·DEFL('b)

definition
(liftemb :: ('a ⊗ 'b) u → udom u) = u-map·emb

definition
(liftprj :: udom u → ('a ⊗ 'b) u) = u-map·prj

definition
liftdefl (t::('a ⊗ 'b) itself) = liftdefl-of·DEFL('a ⊗ 'b)

instance proof
show *ep-pair emb (prj :: udom → 'a ⊗ 'b)*
unfolding *emb-sprod-def prj-sprod-def*
by *(intro ep-pair-comp ep-pair-sprod ep-pair-sprod-map ep-pair-emb-prj)*
show *cast·DEFL('a ⊗ 'b) = emb oo (prj :: udom → 'a ⊗ 'b)*
unfolding *emb-sprod-def prj-sprod-def defl-sprod-def cast-sprod-defl*
by *(simp add: cast-DEFL oo-def cfun-eq-iff sprod-map-map)*
qed *(fact liftemb-sprod-def liftprj-sprod-def liftdefl-sprod-def)+*

end

lemma *DEFL-sprod*:
DEFL('a::domain ⊗ 'b::domain) = sprod-defl·DEFL('a)·DEFL('b)
by *(rule defl-sprod-def)*

23.5.6 Cartesian product

definition *prod-liftdefl :: udom u defl → udom u defl → udom u defl*
where *prod-liftdefl = defl-fun2 (u-map·prod-emb oo decode-prod-u)*
(encode-prod-u oo u-map·prod-prj) sprod-map

lemma *cast-prod-liftdefl*:
cast·(prod-liftdefl·a·b) =
(u-map·prod-emb oo decode-prod-u) oo sprod-map·(cast·a)·(cast·b) oo
(encode-prod-u oo u-map·prod-prj)
unfolding *prod-liftdefl-def*
apply *(rule cast-defl-fun2)*
apply *(intro ep-pair-comp ep-pair-u-map ep-pair-prod)*
apply *(simp add: ep-pair.intro)*

apply (*erule* (*1*) *finite-deflation-sprod-map*)
done

instantiation *prod* :: (*predomain*, *predomain*) *predomain*
begin

definition
liftemb = (*u-map·prod-emb oo decode-prod-u*) *oo*
(*sprod-map·liftemb·liftemb oo encode-prod-u*)

definition
liftprj = (*decode-prod-u oo sprod-map·liftprj·liftprj*) *oo*
(*encode-prod-u oo u-map·prod-prj*)

definition
liftdefl (*t*::(*'a* × *'b*) *itself*) = *prod-liftdefl·LIFTDEFL*(*'a*)·*LIFTDEFL*(*'b*)

instance proof
show *ep-pair liftemb* (*liftprj* :: *udom u* → (*'a* × *'b*) *u*)
unfolding *liftemb-prod-def liftprj-prod-def*
by (*intro ep-pair-comp ep-pair-sprod-map ep-pair-u-map*
ep-pair-prod predomain-ep, *simp-all add*: *ep-pair.intro*)
show *cast·LIFTDEFL*(*'a* × *'b*) = *liftemb oo* (*liftprj* :: *udom u* → (*'a* × *'b*) *u*)
unfolding *liftemb-prod-def liftprj-prod-def liftdefl-prod-def*
by (*simp add*: *cast-prod-liftdefl cast-liftdefl cfcomp1 sprod-map-map*)
qed

end

instantiation *prod* :: (*domain*, *domain*) *domain*
begin

definition
emb = *prod-emb oo prod-map·emb·emb*

definition
prj = *prod-map·prj·prj oo prod-prj*

definition
defl (*t*::(*'a* × *'b*) *itself*) = *prod-defl·DEFL*(*'a*)·*DEFL*(*'b*)

instance proof
show *1*: *ep-pair emb* (*prj* :: *udom* → *'a* × *'b*)
unfolding *emb-prod-def prj-prod-def*
by (*intro ep-pair-comp ep-pair-prod ep-pair-prod-map ep-pair-emb-prj*)
show *2*: *cast·DEFL*(*'a* × *'b*) = *emb oo* (*prj* :: *udom* → *'a* × *'b*)
unfolding *emb-prod-def prj-prod-def defl-prod-def cast-prod-defl*
by (*simp add*: *cast-DEFL oo-def cfun-eq-iff prod-map-map*)
show *3*: *liftemb* = *u-map*·(*emb* :: *'a* × *'b* → *udom*)

unfolding *emb-prod-def liftemb-prod-def liftemb-eq*
unfolding *encode-prod-u-def decode-prod-u-def*
by (*rule cfun-eqI*, *case-tac x*, *simp*, *clarsimp*)
show *4*: *liftprj* = *u-map*·(*prj* :: *udom* → *'a* × *'b*)
unfolding *prj-prod-def liftprj-prod-def liftprj-eq*
unfolding *encode-prod-u-def decode-prod-u-def*
apply (*rule cfun-eqI*, *case-tac x*, *simp*)
apply (*rename-tac y*, *case-tac prod-prj*·*y*, *simp*)
done
show *5*: *LIFTDEFL*(*'a* × *'b*) = *liftdefl-of*·*DEFL*(*'a* × *'b*)
by (*rule cast-eq-imp-eq*)
(*simp add*: *cast-liftdefl cast-liftdefl-of cast-DEFL 2 3 4 u-map-oo*)
qed

end

lemma *DEFL-prod*:
DEFL(*'a*::*domain* × *'b*::*domain*) = *prod-defl*·*DEFL*(*'a*)·*DEFL*(*'b*)
by (*rule defl-prod-def*)

lemma *LIFTDEFL-prod*:
LIFTDEFL(*'a*::*predomain* × *'b*::*predomain*) =
prod-liftdefl·*LIFTDEFL*(*'a*)·*LIFTDEFL*(*'b*)
by (*rule liftdefl-prod-def*)

23.5.7 Unit type

instantiation *unit* :: *domain*
begin

definition
emb = (⊥ :: *unit* → *udom*)

definition
prj = (⊥ :: *udom* → *unit*)

definition
defl (*t*::*unit itself*) = ⊥

definition
(*liftemb* :: *unit u* → *udom u*) = *u-map*·*emb*

definition
(*liftprj* :: *udom u* → *unit u*) = *u-map*·*prj*

definition
liftdefl (*t*::*unit itself*) = *liftdefl-of*·*DEFL*(*unit*)

instance proof

```
  show ep-pair emb (prj :: udom → unit)
    unfolding emb-unit-def prj-unit-def
    by (simp add: ep-pair.intro)
  show cast·DEFL(unit) = emb oo (prj :: udom → unit)
    unfolding emb-unit-def prj-unit-def defl-unit-def by simp
qed (fact liftemb-unit-def liftprj-unit-def liftdefl-unit-def)+

end
```

23.5.8 Discrete cpo

```
instantiation discr :: (countable) predomain
begin

definition
  (liftemb :: 'a discr u → udom u) = strictify·up oo udom-emb discr-approx

definition
  (liftprj :: udom u → 'a discr u) = udom-prj discr-approx oo fup·ID

definition
  liftdefl (t::'a discr itself) =
    (⨆ i. defl-principal (Abs-fin-defl (liftemb oo discr-approx i oo (liftprj::udom u
→ 'a discr u))))

instance proof
  show 1: ep-pair liftemb (liftprj :: udom u → 'a discr u)
    unfolding liftemb-discr-def liftprj-discr-def
    apply (intro ep-pair-comp ep-pair-udom [OF discr-approx])
    apply (rule ep-pair.intro)
    apply (simp add: strictify-conv-if)
    apply (case-tac y, simp, simp add: strictify-conv-if)
    done
  show cast·LIFTDEFL('a discr) = liftemb oo (liftprj :: udom u → 'a discr u)
    unfolding liftdefl-discr-def
    apply (subst contlub-cfun-arg)
    apply (rule chainI)
    apply (rule defl.principal-mono)
    apply (simp add: below-fin-defl-def)
    apply (simp add: Abs-fin-defl-inverse
        ep-pair.finite-deflation-e-d-p [OF 1]
        approx-chain.finite-deflation-approx [OF discr-approx])
    apply (intro monofun-cfun below-refl)
    apply (rule chainE)
    apply (rule chain-discr-approx)
    apply (subst cast-defl-principal)
    apply (simp add: Abs-fin-defl-inverse
        ep-pair.finite-deflation-e-d-p [OF 1]
        approx-chain.finite-deflation-approx [OF discr-approx])
```

```
  apply (simp add: lub-distribs)
  done
qed

end
```

23.5.9 Strict sum

```
instantiation ssum :: (domain, domain) domain
begin

definition
  emb = ssum-emb oo ssum-map·emb·emb

definition
  prj = ssum-map·prj·prj oo ssum-prj

definition
  defl (t::('a ⊕ 'b) itself) = ssum-defl·DEFL('a)·DEFL('b)

definition
  (liftemb :: ('a ⊕ 'b) u → udom u) = u-map·emb

definition
  (liftprj :: udom u → ('a ⊕ 'b) u) = u-map·prj

definition
  liftdefl (t::('a ⊕ 'b) itself) = liftdefl-of·DEFL('a ⊕ 'b)

instance proof
  show ep-pair emb (prj :: udom → 'a ⊕ 'b)
    unfolding emb-ssum-def prj-ssum-def
    by (intro ep-pair-comp ep-pair-ssum ep-pair-ssum-map ep-pair-emb-prj)
  show cast·DEFL('a ⊕ 'b) = emb oo (prj :: udom → 'a ⊕ 'b)
    unfolding emb-ssum-def prj-ssum-def defl-ssum-def cast-ssum-defl
    by (simp add: cast-DEFL oo-def cfun-eq-iff ssum-map-map)
qed (fact liftemb-ssum-def liftprj-ssum-def liftdefl-ssum-def)+

end

lemma DEFL-ssum:
  DEFL('a::domain ⊕ 'b::domain) = ssum-defl·DEFL('a)·DEFL('b)
by (rule defl-ssum-def)
```

23.5.10 Lifted HOL type

```
instantiation lift :: (countable) domain
begin

definition
```

emb = *emb oo* (Λ *x*. *Rep-lift x*)

definition
prj = (Λ *y*. *Abs-lift y*) *oo prj*

definition
defl (*t*::'*a lift itself*) = *DEFL*('*a discr u*)

definition
(*liftemb* :: '*a lift u* → *udom u*) = *u-map*·*emb*

definition
(*liftprj* :: *udom u* → '*a lift u*) = *u-map*·*prj*

definition
liftdefl (*t*::'*a lift itself*) = *liftdefl-of*·*DEFL*('*a lift*)

instance proof
 note [*simp*] = *cont-Rep-lift cont-Abs-lift Rep-lift-inverse Abs-lift-inverse*
 have *ep-pair* (Λ(*x*::'*a lift*). *Rep-lift x*) (Λ *y*. *Abs-lift y*)
  by (*simp add*: *ep-pair-def*)
 thus *ep-pair emb* (*prj* :: *udom* → '*a lift*)
  unfolding *emb-lift-def prj-lift-def*
  using *ep-pair-emb-prj* **by** (*rule ep-pair-comp*)
 show *cast*·*DEFL*('*a lift*) = *emb oo* (*prj* :: *udom* → '*a lift*)
  unfolding *emb-lift-def prj-lift-def defl-lift-def cast-DEFL*
  by (*simp add*: *cfcomp1*)
qed (*fact liftemb-lift-def liftprj-lift-def liftdefl-lift-def*)+

end

end

24 The unit domain

theory *One*
 imports *Lift*
begin

type-synonym *one* = *unit lift*

translations
 (*type*) *one* ↽ (*type*) *unit lift*

definition *ONE* :: *one*
 where *ONE* ≡ *Def* ()

Exhaustion and Elimination for type *one*

lemma *Exh-one*: $t = \perp \lor t = ONE$

by (*induct t*) (*simp-all add*: *ONE-def*)

lemma *oneE* [*case-names bottom ONE*]: $[\![p = \bot \Longrightarrow Q;\ p = ONE \Longrightarrow Q]\!] \Longrightarrow Q$
by (*induct p*) (*simp-all add*: *ONE-def*)

lemma *one-induct* [*case-names bottom ONE*]: $P\ \bot \Longrightarrow P\ ONE \Longrightarrow P\ x$
by (*cases x rule*: *oneE*) *simp-all*

lemma *dist-below-one* [*simp*]: $ONE \not\sqsubseteq \bot$
by (*simp add*: *ONE-def*)

lemma *below-ONE* [*simp*]: $x \sqsubseteq ONE$
by (*induct x rule*: *one-induct*) *simp-all*

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

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

lemma *one-neq-iffs* [*simp*]:
$x \neq ONE \longleftrightarrow x = \bot$
$ONE \neq x \longleftrightarrow x = \bot$
$x \neq \bot \longleftrightarrow x = ONE$
$\bot \neq x \longleftrightarrow x = ONE$
by (*induct x rule*: *one-induct*) *simp-all*

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

Case analysis function for type *one*

definition *one-case* :: $'a::pcpo \rightarrow one \rightarrow 'a$
where *one-case* = $(\Lambda\ a\ x.\ seq\cdot x\cdot a)$

translations
$case\ x\ of\ XCONST\ ONE \Rightarrow t \rightleftharpoons CONST\ one\text{-}case\cdot t\cdot x$
$case\ x\ of\ XCONST\ ONE :: 'a \Rightarrow t \rightharpoonup CONST\ one\text{-}case\cdot t\cdot x$
$\Lambda\ (XCONST\ ONE).\ t \rightleftharpoons CONST\ one\text{-}case\cdot t$

lemma *one-case1* [*simp*]: $(case\ \bot\ of\ ONE \Rightarrow t) = \bot$
by (*simp add*: *one-case-def*)

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

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

end

theory *Fixrec*
imports *Cprod Sprod Ssum Up One Tr Cfun*
keywords *fixrec* :: *thy-defn*
begin

25 Fixed point operator and admissibility

25.1 Iteration

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

Derive inductive properties of iterate from primitive recursion

lemma *iterate-0* [*simp*]: *iterate 0*·*F*·*x* = *x*
  by *simp*

lemma *iterate-Suc* [*simp*]: *iterate* (*Suc n*)·*F*·*x* = *F*·(*iterate n*·*F*·*x*)
  by *simp*

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

lemma *iterate-Suc2*: *iterate* (*Suc n*)·*F*·*x* = *iterate n*·*F*·(*F*·*x*)
  by (*induct n*) *simp-all*

lemma *iterate-iterate*: *iterate m*·*F*·(*iterate n*·*F*·*x*) = *iterate* (*m* + *n*)·*F*·*x*
  by (*induct m*) *simp-all*

The sequence of function iterations is a chain.

lemma *chain-iterate* [*simp*]: *chain* (λ*i. iterate i*·*F*·⊥)
  by (*rule chainI*, *unfold iterate-Suc2*, *rule monofun-cfun-arg*, *rule minimal*)

25.2 Least fixed point operator

definition *fix* :: (*'a*::*pcpo* → *'a*) → *'a*
  where *fix* = (Λ *F*. ⨆*i. iterate i*·*F*·⊥)

Binder syntax for *fix*

abbreviation *fix-syn* :: (*'a*::*pcpo* ⇒ *'a*) ⇒ *'a* (**binder** ‹μ › *10*)
  where *fix-syn* (λ*x. f x*) ≡ *fix*·(Λ *x. f x*)

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

Properties of *fix*

direct connection between *fix* and iteration

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

lemma *iterate-below-fix*: $iterate\ n \cdot f \cdot \bot \sqsubseteq fix \cdot f$
  unfolding *fix-def2*
  using *chain-iterate* **by** (*rule is-ub-thelub*)

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

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

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

lemma *fix-least*: $F \cdot x = x \Longrightarrow fix \cdot F \sqsubseteq x$
  by (*rule fix-least-below*) *simp*

lemma *fix-eqI*:
  assumes *fixed*: $F \cdot x = x$
    and *least*: $\bigwedge z.\ F \cdot z = z \Longrightarrow x \sqsubseteq z$
  shows $fix \cdot F = x$
  apply (*rule below-antisym*)
   apply (*rule fix-least* [*OF fixed*])
  apply (*rule least* [*OF fix-eq* [*symmetric*]])
  done

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

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

lemma *fix-eq4*: $f = fix \cdot F \Longrightarrow f = F \cdot f$

by (*erule ssubst*) (*rule fix-eq*)

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

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

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

fix applied to identity and constant functions

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

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

25.3 Fixed point induction

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

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

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

lemma *fix-ind2*:
assumes *adm*: *adm P*
assumes *0*: $P\ \bot$ **and** *1*: $P\ (F{\cdot}\bot)$
assumes *step*: $\bigwedge x.\ [\![P\ x;\ P\ (F{\cdot}x)]\!] \Longrightarrow P\ (F{\cdot}(F{\cdot}x))$
shows $P\ (fix{\cdot}F)$
unfolding *fix-def2*
apply (*rule admD* [*OF adm chain-iterate*])

```
  apply (rule nat-less-induct)
  apply (case-tac n)
   apply (simp add: 0)
  apply (case-tac nat)
   apply (simp add: 1)
  apply (frule-tac x=nat in spec)
  apply (simp add: step)
  done

lemma parallel-fix-ind:
  assumes adm: adm (λx. P (fst x) (snd x))
  assumes base: P ⊥ ⊥
  assumes step: ⋀x y. P x y ⟹ P (F·x) (G·y)
  shows P (fix·F) (fix·G)
proof –
  from adm have adm': adm (case-prod P)
    unfolding split-def .
  have P (iterate i·F·⊥) (iterate i·G·⊥) for i
    by (induct i) (simp add: base, simp add: step)
  then have ⋀i. case-prod P (iterate i·F·⊥, iterate i·G·⊥)
    by simp
  then have case-prod P (⨆i. (iterate i·F·⊥, iterate i·G·⊥))
    by – (rule admD [OF adm'], simp, assumption)
  then have case-prod P (⨆i. iterate i·F·⊥, ⨆i. iterate i·G·⊥)
    by (simp add: lub-Pair)
  then have P (⨆i. iterate i·F·⊥) (⨆i. iterate i·G·⊥)
    by simp
  then show P (fix·F) (fix·G)
    by (simp add: fix-def2)
qed

lemma cont-parallel-fix-ind:
  assumes cont F and cont G
  assumes adm (λx. P (fst x) (snd x))
  assumes P ⊥ ⊥
  assumes ⋀x y. P x y ⟹ P (F x) (G y)
  shows P (fix·(Abs-cfun F)) (fix·(Abs-cfun G))
  by (rule parallel-fix-ind) (simp-all add: assms)
```

25.4 Fixed-points on product types

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

```
lemma fix-cprod:
  fixes F :: 'a::pcpo × 'b::pcpo → 'a × 'b
  shows
    fix·F =
     (μ x. fst (F·(x, μ y. snd (F·(x, y)))),
      μ y. snd (F·(μ x. fst (F·(x, μ y. snd (F·(x, y)))), y)))
```

(**is** $fix \cdot F = (?x,\ ?y)$)
proof (*rule fix-eqI*)
have ∗: $fst\ (F \cdot (?x,\ ?y)) = ?x$
by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
have $snd\ (F \cdot (?x,\ ?y)) = ?y$
by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
with ∗ **show** $F \cdot (?x,\ ?y) = (?x,\ ?y)$
by (*simp add*: *prod-eq-iff*)
next
fix z
assume *F-z*: $F \cdot z = z$
obtain $x\ y$ **where** z: $z = (x,\ y)$ **by** (*rule prod.exhaust*)
from *F-z z* **have** *F-x*: $fst\ (F \cdot (x,\ y)) = x$ **by** *simp*
from *F-z z* **have** *F-y*: $snd\ (F \cdot (x,\ y)) = y$ **by** *simp*
let $?y1 = \mu\ y.\ snd\ (F \cdot (x,\ y))$
have $?y1 \sqsubseteq y$
by (*rule fix-least*) (*simp add*: *F-y*)
then have $fst\ (F \cdot (x,\ ?y1)) \sqsubseteq fst\ (F \cdot (x,\ y))$
by (*simp add*: *fst-monofun monofun-cfun*)
with *F-x* **have** $fst\ (F \cdot (x,\ ?y1)) \sqsubseteq x$
by *simp*
then have ∗: $?x \sqsubseteq x$
by (*simp add*: *fix-least-below*)
then have $snd\ (F \cdot (?x,\ y)) \sqsubseteq snd\ (F \cdot (x,\ y))$
by (*simp add*: *snd-monofun monofun-cfun*)
with *F-y* **have** $snd\ (F \cdot (?x,\ y)) \sqsubseteq y$
by *simp*
then have $?y \sqsubseteq y$
by (*simp add*: *fix-least-below*)
with z ∗ **show** $(?x,\ ?y) \sqsubseteq z$
by *simp*
qed

26 Package for defining recursive functions in HOLCF

26.1 Pattern-match monad

pcpodef $'a\ match = UNIV::(one\ {+}{+}\ 'a\ u)\ set$
by *simp-all*

definition
fail :: $'a\ match$ **where**
$fail = Abs\text{-}match\ (sinl \cdot ONE)$

definition
succeed :: $'a \to 'a\ match$ **where**
$succeed = (\Lambda\ x.\ Abs\text{-}match\ (sinr \cdot (up \cdot x)))$

lemma *matchE* [*case-names bottom fail succeed*, *cases type*: *match*]:

$[\![p = \bot \Longrightarrow Q;\ p = fail \Longrightarrow Q;\ \bigwedge x.\ p = succeed \cdot x \Longrightarrow Q]\!] \Longrightarrow Q$
unfolding *fail-def succeed-def*
apply (*cases p*, *rename-tac r*)
apply (*rule-tac p=r* **in** *ssumE*, *simp add*: *Abs-match-strict*)
apply (*rule-tac p=x* **in** *oneE*, *simp*, *simp*)
apply (*rule-tac p=y* **in** *upE*, *simp*, *simp add*: *cont-Abs-match*)
done

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

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

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

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

26.1.1 Run operator

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

rewrite rules for run

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

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

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

26.1.2 Monad plus operator

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

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

rewrite rules for mplus

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

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

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

lemma *mplus-fail2* [*simp*]: *m* +++ *fail* = *m*
by (*cases m*, *simp-all*)

lemma *mplus-assoc*: (*x* +++ *y*) +++ *z* = *x* +++ (*y* +++ *z*)
by (*cases x*, *simp-all*)

26.2 Match functions for built-in types

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

definition
  match-Pair :: *'a* × *'b* → (*'a* → *'b* → *'c match*) → *'c match*
where
  match-Pair = (Λ *x k*. *csplit·k·x*)

definition
  match-spair :: *'a*::*pcpo* ⊗ *'b*::*pcpo* → (*'a* → *'b* → *'c match*) → *'c*::*pcpo match*
where
  match-spair = (Λ *x k*. *ssplit·k·x*)

definition
  match-sinl :: *'a*::*pcpo* ⊕ *'b*::*pcpo* → (*'a* → *'c*::*pcpo match*) → *'c match*
where
  match-sinl = (Λ *x k*. *sscase·k·*(Λ *b*. *fail*)*·x*)

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

definition
  match-up :: *'a u* → (*'a* → *'c*::*pcpo match*) → *'c match*
where

match-up = (Λ *x k*. *fup*·*k*·*x*)

definition
match-ONE :: *one* → ′*c*::*pcpo match* → ′*c match*
where
match-ONE = (Λ *ONE k*. *k*)

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

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

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

lemma *match-Pair-simps* [*simp*]:
match-Pair·(*x*, *y*)·*k* = *k*·*x*·*y*
by (*simp-all add*: *match-Pair-def*)

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

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

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

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

lemma *match-ONE-simps* [*simp*]:
match-ONE·*ONE*·*k* = *k*

$match\text{-}ONE \cdot \bot \cdot k = \bot$
by (*simp-all add*: *match-ONE-def*)

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

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

26.3 Mutual recursion

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

lemma *Pair-equalI*: $[\![x \equiv fst\ p;\ y \equiv snd\ p]\!] \Longrightarrow (x,\ y) \equiv p$
by *simp*

lemma *Pair-eqD1*: $(x,\ y) = (x',\ y') \Longrightarrow x = x'$
by *simp*

lemma *Pair-eqD2*: $(x,\ y) = (x',\ y') \Longrightarrow y = y'$
by *simp*

lemma *def-cont-fix-eq*:
$[\![f \equiv fix \cdot (Abs\text{-}cfun\ F);\ cont\ F]\!] \Longrightarrow f = F\ f$
by (*simp*, *subst fix-eq*, *simp*)

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

lemma for proving rewrite rules

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

26.4 Initializing the fixrec package

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

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

setup ‹
  Fixrec.add-matchers
    [(***const-name*** ‹*up*›, ***const-name*** ‹*match-up*›),
      (***const-name*** ‹*sinl*›, ***const-name*** ‹*match-sinl*›),
      (***const-name*** ‹*sinr*›, ***const-name*** ‹*match-sinr*›),
      (***const-name*** ‹*spair*›, ***const-name*** ‹*match-spair*›),
      (***const-name*** ‹*Pair*›, ***const-name*** ‹*match-Pair*›),
      (***const-name*** ‹*ONE*›, ***const-name*** ‹*match-ONE*›),
      (***const-name*** ‹*TT*›, ***const-name*** ‹*match-TT*›),
      (***const-name*** ‹*FF*›, ***const-name*** ‹*match-FF*›),
      (***const-name*** ‹*bottom*›, ***const-name*** ‹*match-bottom*›)]
›

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

end

27 Domain package

theory *Domain*
imports *Representable Map-Functions Fixrec*
keywords
  lazy unsafe **and**
  domaindef domain :: *thy-defn* **and**
  domain-isomorphism :: *thy-decl*
begin

27.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
  fixes *abs* :: *'a::pcpo → 'b::pcpo*
  fixes *rep* :: *'b → 'a*
  assumes *abs-iso* [*simp*]: *rep·(abs·x) = x*
  assumes *rep-iso* [*simp*]: *abs·(rep·y) = y*
begin

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

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

qed

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

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

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

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

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

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

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

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

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

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

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

lemma *casedist-rule*: $rep{\cdot}x = \bot \vee P \Longrightarrow x = \bot \vee P$
  by (*simp add*: *rep-bottom-iff*)

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

proof (*unfold compact-def*)
  assume *adm* ($\lambda y.$ *abs*$\cdot x \not\sqsubseteq y$)
  with *cont-Rep-cfun2*
  have *adm* ($\lambda y.$ *abs*$\cdot x \not\sqsubseteq$ *abs*$\cdot y$) **by** (*rule adm-subst*)
  then show *adm* ($\lambda y.\ x \not\sqsubseteq y$) **using** *abs-below* **by** *simp*
qed

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

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

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

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

end

27.2 Proofs about take functions

This section contains lemmas that are used in a module that supports the domain isomorphism package; the module contains proofs related to take functions and the finiteness predicate.

lemma *deflation-abs-rep*:
  fixes *abs* **and** *rep* **and** d
  assumes *abs-iso*: $\bigwedge x.$ *rep*$\cdot$(*abs*$\cdot x$) = x
  assumes *rep-iso*: $\bigwedge y.$ *abs*$\cdot$(*rep*$\cdot y$) = y
  shows *deflation* $d \Longrightarrow$ *deflation* (*abs oo d oo rep*)
by (*rule ep-pair.deflation-e-d-p*) (*simp add*: *ep-pair.intro assms*)

lemma *deflation-chain-min*:
  assumes *chain*: *chain* d
  assumes *defl*: $\bigwedge n.$ *deflation* ($d\ n$)
  shows $d\ m\cdot(d\ n\cdot x) = d$ (*min* $m\ n$)$\cdot x$
proof (*rule linorder-le-cases*)
  assume $m \leq n$
  with *chain* **have** $d\ m \sqsubseteq d\ n$ **by** (*rule chain-mono*)
  then have $d\ m\cdot(d\ n\cdot x) = d\ m\cdot x$

by (*rule deflation-below-comp1* [*OF defl defl*])
moreover from ‹$m \le n$› **have** $min\ m\ n = m$ **by** *simp*
ultimately show *?thesis* **by** *simp*
next
assume $n \le m$
with *chain* **have** $d\ n \sqsubseteq d\ m$ **by** (*rule chain-mono*)
then have $d\ m \cdot (d\ n \cdot x) = d\ n \cdot x$
by (*rule deflation-below-comp2* [*OF defl defl*])
moreover from ‹$n \le m$› **have** $min\ m\ n = n$ **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *lub-ID-take-lemma*:
assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
assumes $\bigwedge n.\ t\ n \cdot x = t\ n \cdot y$ **shows** $x = y$
proof −
have $(\bigsqcup n.\ t\ n \cdot x) = (\bigsqcup n.\ t\ n \cdot y)$
using *assms*(*3*) **by** *simp*
then have $(\bigsqcup n.\ t\ n) \cdot x = (\bigsqcup n.\ t\ n) \cdot y$
using *assms*(*1*) **by** (*simp add*: *lub-distribs*)
then show $x = y$
using *assms*(*2*) **by** *simp*
qed

lemma *lub-ID-reach*:
assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
shows $(\bigsqcup n.\ t\ n \cdot x) = x$
using *assms* **by** (*simp add*: *lub-distribs*)

lemma *lub-ID-take-induct*:
assumes *chain t* **and** $(\bigsqcup n.\ t\ n) = ID$
assumes *adm P* **and** $\bigwedge n.\ P\ (t\ n \cdot x)$ **shows** $P\ x$
proof −
from ‹*chain t*› **have** *chain* $(\lambda n.\ t\ n \cdot x)$ **by** *simp*
from ‹*adm P*› *this* ‹$\bigwedge n.\ P\ (t\ n \cdot x)$› **have** $P\ (\bigsqcup n.\ t\ n \cdot x)$ **by** (*rule admD*)
with ‹*chain t*› ‹$(\bigsqcup n.\ t\ n) = ID$› **show** $P\ x$ **by** (*simp add*: *lub-distribs*)
qed

27.3 Finiteness

Let a "decisive" function be a deflation that maps every input to either itself or bottom. Then if a domain's take functions are all decisive, then all values in the domain are finite.

definition
decisive :: $('a::pcpo \to 'a) \Rightarrow bool$
where
$decisive\ d \longleftrightarrow (\forall x.\ d \cdot x = x \lor d \cdot x = \bot)$

lemma *decisiveI*: $(\bigwedge x.\ d \cdot x = x \lor d \cdot x = \bot) \Longrightarrow decisive\ d$

```
  unfolding decisive-def by simp

lemma decisive-cases:
  assumes decisive d obtains d·x = x | d·x = ⊥
using assms unfolding decisive-def by auto

lemma decisive-bottom: decisive ⊥
  unfolding decisive-def by simp

lemma decisive-ID: decisive ID
  unfolding decisive-def by simp

lemma decisive-ssum-map:
  assumes f: decisive f
  assumes g: decisive g
  shows decisive (ssum-map·f·g)
  apply (rule decisiveI)
  subgoal for s
    apply (cases s, simp-all)
     apply (rule-tac x=x in decisive-cases [OF f], simp-all)
    apply (rule-tac x=y in decisive-cases [OF g], simp-all)
    done
  done

lemma decisive-sprod-map:
  assumes f: decisive f
  assumes g: decisive g
  shows decisive (sprod-map·f·g)
  apply (rule decisiveI)
  subgoal for s
    apply (cases s, simp)
    subgoal for x y
      apply (rule decisive-cases [OF f, where x = x], simp-all)
      apply (rule decisive-cases [OF g, where x = y], simp-all)
      done
    done
  done

lemma decisive-abs-rep:
  fixes abs rep
  assumes iso: iso abs rep
  assumes d: decisive d
  shows decisive (abs oo d oo rep)
  apply (rule decisiveI)
  subgoal for s
    apply (rule decisive-cases [OF d, where x=rep·s])
     apply (simp add: iso.rep-iso [OF iso])
    apply (simp add: iso.abs-strict [OF iso])
    done
```

done

lemma *lub-ID-finite*:
assumes *chain*: *chain d*
assumes *lub*: $(\bigsqcup n.\ d\ n) = ID$
assumes *decisive*: $\bigwedge n.\ decisive\ (d\ n)$
shows $\exists\, n.\ d\ n{\cdot}x = x$
proof −
have *1*: $chain\ (\lambda n.\ d\ n{\cdot}x)$ **using** *chain* **by** *simp*
have *2*: $(\bigsqcup n.\ d\ n{\cdot}x) = x$ **using** *chain lub* **by** (*rule lub-ID-reach*)
have $\forall\, n.\ d\ n{\cdot}x = x \lor d\ n{\cdot}x = \bot$
using *decisive* **unfolding** *decisive-def* **by** *simp*
hence $range\ (\lambda n.\ d\ n{\cdot}x) \subseteq \{x, \bot\}$
by *auto*
hence $finite\ (range\ (\lambda n.\ d\ n{\cdot}x))$
by (*rule finite-subset*, *simp*)
with *1* **have** $finite\text{-}chain\ (\lambda n.\ d\ n{\cdot}x)$
by (*rule finite-range-imp-finch*)
then have $\exists\, n.\ (\bigsqcup n.\ d\ n{\cdot}x) = d\ n{\cdot}x$
unfolding *finite-chain-def* **by** (*auto simp add*: *maxinch-is-thelub*)
with *2* **show** $\exists\, n.\ d\ n{\cdot}x = x$ **by** (*auto elim*: *sym*)
qed

lemma *lub-ID-finite-take-induct*:
assumes *chain d* **and** $(\bigsqcup n.\ d\ n) = ID$ **and** $\bigwedge n.\ decisive\ (d\ n)$
shows $(\bigwedge n.\ P\ (d\ n{\cdot}x)) \Longrightarrow P\ x$
using *lub-ID-finite* [*OF assms*] **by** *metis*

27.4 Proofs about constructor functions

Lemmas for proving nchotomy rule:

lemma *ex-one-bottom-iff*:
$(\exists\, x.\ P\ x \land x \neq \bot) = P\ ONE$
by *simp*

lemma *ex-up-bottom-iff*:
$(\exists\, x.\ P\ x \land x \neq \bot) = (\exists\, x.\ P\ (up{\cdot}x))$
by (*safe*, *case-tac x*, *auto*)

lemma *ex-sprod-bottom-iff*:
$(\exists\, y.\ P\ y \land y \neq \bot) =$
$(\exists\, x\ y.\ (P\ (:x,\ y:) \land x \neq \bot) \land y \neq \bot)$
by (*safe*, *case-tac y*, *auto*)

lemma *ex-sprod-up-bottom-iff*:
$(\exists\, y.\ P\ y \land y \neq \bot) =$
$(\exists\, x\ y.\ P\ (:up{\cdot}x,\ y:) \land y \neq \bot)$
by (*safe*, *case-tac y*, *simp*, *case-tac x*, *auto*)

lemma *ex-ssum-bottom-iff*:
$(\exists x.\ P\ x \wedge x \neq \bot) =$
$((\exists x.\ P\ (sinl \cdot x) \wedge x \neq \bot) \vee$
$(\exists x.\ P\ (sinr \cdot x) \wedge x \neq \bot))$
by (*safe*, *case-tac x*, *auto*)

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

lemmas *ex-bottom-iffs* =
ex-ssum-bottom-iff
ex-sprod-up-bottom-iff
ex-sprod-bottom-iff
ex-up-bottom-iff
ex-one-bottom-iff

Rules for turning nchotomy into exhaust:

lemma *exh-casedist0*: $[\![R;\ R \Longrightarrow P]\!] \Longrightarrow P$
by *auto*

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

Rules for proving constructor properties

lemmas *con-strict-rules* =
sinl-strict sinr-strict spair-strict1 spair-strict2

lemmas *con-bottom-iff-rules* =
sinl-bottom-iff sinr-bottom-iff spair-bottom-iff up-defined ONE-defined

lemmas *con-below-iff-rules* =
sinl-below sinr-below sinl-below-sinr sinr-below-sinl con-bottom-iff-rules

lemmas *con-eq-iff-rules* =
sinl-eq sinr-eq sinl-eq-sinr sinr-eq-sinl con-bottom-iff-rules

lemmas *sel-strict-rules* =
cfcomp2 sscase1 sfst-strict ssnd-strict fup1

lemma *sel-app-extra-rules*:
$sscase \cdot ID \cdot \bot \cdot (sinr \cdot x) = \bot$

sscase·ID·⊥·(sinl·x) = x
sscase·⊥·ID·(sinl·x) = ⊥
sscase·⊥·ID·(sinr·x) = x
fup·ID·(up·x) = x
by (*cases x* = ⊥, *simp*, *simp*)+

lemmas *sel-app-rules* =
sel-strict-rules sel-app-extra-rules
ssnd-spair sfst-spair up-defined spair-defined

lemmas *sel-bottom-iff-rules* =
cfcomp2 sfst-bottom-iff ssnd-bottom-iff

lemmas *take-con-rules* =
ssum-map-sinl′ ssum-map-sinr′ sprod-map-spair′ u-map-up
deflation-strict deflation-ID ID1 cfcomp2

27.5 ML setup

named-theorems *domain-deflation theorems like deflation a ==> deflation* (*foo-map*$*a*)
and *domain-map-ID theorems like foo-map*$*ID* = *ID*

ML-file ‹*Tools/Domain/domain-take-proofs.ML*›
ML-file ‹*Tools/cont-consts.ML*›
ML-file ‹*Tools/cont-proc.ML*›
simproc-setup *cont* (*cont f*) = ‹*K ContProc.cont-proc*›

ML-file ‹*Tools/Domain/domain-constructors.ML*›
ML-file ‹*Tools/Domain/domain-induction.ML*›

27.6 Representations of types

lemma *emb-prj*: *emb*·((*prj*·*x*)::′*a*::*domain*) = *cast*·*DEFL*(′*a*)·*x*
by (*simp add*: *cast-DEFL*)

lemma *emb-prj-emb*:
fixes *x* :: ′*a*::*domain*
assumes *DEFL*(′*a*) ⊑ *DEFL*(′*b*)
shows *emb*·(*prj*·(*emb*·*x*) :: ′*b*::*domain*) = *emb*·*x*
unfolding *emb-prj*
apply (*rule cast.belowD*)
apply (*rule monofun-cfun-arg* [*OF assms*])
apply (*simp add*: *cast-DEFL*)
done

lemma *prj-emb-prj*:
assumes *DEFL*(′*a*::*domain*) ⊑ *DEFL*(′*b*::*domain*)
shows *prj*·(*emb*·(*prj*·*x* :: ′*b*)) = (*prj*·*x* :: ′*a*)
apply (*rule emb-eq-iff* [*THEN iffD1*])
apply (*simp only*: *emb-prj*)

apply (*rule deflation-below-comp1*)
apply (*rule deflation-cast*)
apply (*rule deflation-cast*)
apply (*rule monofun-cfun-arg* [*OF assms*])
done

Isomorphism lemmas used internally by the domain package:

lemma *domain-abs-iso*:
fixes *abs* **and** *rep*
assumes *DEFL*: *DEFL*('*b*::*domain*) = *DEFL*('*a*::*domain*)
assumes *abs-def*: (*abs* :: '*a* → '*b*) ≡ *prj oo emb*
assumes *rep-def*: (*rep* :: '*b* → '*a*) ≡ *prj oo emb*
shows *rep*·(*abs*·*x*) = *x*
unfolding *abs-def rep-def*
by (*simp add*: *emb-prj-emb DEFL*)

lemma *domain-rep-iso*:
fixes *abs* **and** *rep*
assumes *DEFL*: *DEFL*('*b*::*domain*) = *DEFL*('*a*::*domain*)
assumes *abs-def*: (*abs* :: '*a* → '*b*) ≡ *prj oo emb*
assumes *rep-def*: (*rep* :: '*b* → '*a*) ≡ *prj oo emb*
shows *abs*·(*rep*·*x*) = *x*
unfolding *abs-def rep-def*
by (*simp add*: *emb-prj-emb DEFL*)

27.7 Deflations as sets

definition *defl-set* :: '*a*::*bifinite defl* ⇒ '*a set*
where *defl-set A* = {*x*. *cast*·*A*·*x* = *x*}

lemma *adm-defl-set*: *adm* (λ*x*. *x* ∈ *defl-set A*)
unfolding *defl-set-def* **by** *simp*

lemma *defl-set-bottom*: ⊥ ∈ *defl-set A*
unfolding *defl-set-def* **by** *simp*

lemma *defl-set-cast* [*simp*]: *cast*·*A*·*x* ∈ *defl-set A*
unfolding *defl-set-def* **by** *simp*

lemma *defl-set-subset-iff*: *defl-set A* ⊆ *defl-set B* ⟷ *A* ⊑ *B*
apply (*simp add*: *defl-set-def subset-eq cast-below-cast* [*symmetric*])
apply (*auto simp add*: *cast.belowI cast.belowD*)
done

27.8 Proving a subtype is representable

Temporarily relax type constraints.

setup ‹
fold Sign.add-const-constraint

```
  [ (const-name ‹defl›, SOME typ ‹'a::pcpo itself ⇒ udom defl›)
  , (const-name ‹emb›, SOME typ ‹'a::pcpo → udom›)
  , (const-name ‹prj›, SOME typ ‹udom → 'a::pcpo›)
  , (const-name ‹liftdefl›, SOME typ ‹'a::pcpo itself ⇒ udom u defl›)
  , (const-name ‹liftemb›, SOME typ ‹'a::pcpo u → udom u›)
  , (const-name ‹liftprj›, SOME typ ‹udom u → 'a::pcpo u›) ]
›

lemma typedef-domain-class:
  fixes Rep :: 'a::pcpo ⇒ udom
  fixes Abs :: udom ⇒ 'a::pcpo
  fixes t :: udom defl
  assumes type: type-definition Rep Abs (defl-set t)
  assumes below: (⊑) ≡ λx y. Rep x ⊑ Rep y
  assumes emb: emb ≡ (Λ x. Rep x)
  assumes prj: prj ≡ (Λ x. Abs (cast·t·x))
  assumes defl: defl ≡ (λ a::'a itself. t)
  assumes liftemb: (liftemb :: 'a u → udom u) ≡ u-map·emb
  assumes liftprj: (liftprj :: udom u → 'a u) ≡ u-map·prj
  assumes liftdefl: (liftdefl :: 'a itself ⇒ -) ≡ (λt. liftdefl-of·DEFL('a))
  shows OFCLASS('a, domain-class)
proof
  have emb-beta: ⋀x. emb·x = Rep x
    unfolding emb
    apply (rule beta-cfun)
    apply (rule typedef-cont-Rep [OF type below adm-defl-set cont-id])
    done
  have prj-beta: ⋀y. prj·y = Abs (cast·t·y)
    unfolding prj
    apply (rule beta-cfun)
    apply (rule typedef-cont-Abs [OF type below adm-defl-set])
    apply simp-all
    done
  have prj-emb: ⋀x::'a. prj·(emb·x) = x
    using type-definition.Rep [OF type]
    unfolding prj-beta emb-beta defl-set-def
    by (simp add: type-definition.Rep-inverse [OF type])
  have emb-prj: ⋀y. emb·(prj·y :: 'a) = cast·t·y
    unfolding prj-beta emb-beta
    by (simp add: type-definition.Abs-inverse [OF type])
  show ep-pair (emb :: 'a → udom) prj
    apply standard
    apply (simp add: prj-emb)
    apply (simp add: emb-prj cast.below)
    done
  show cast·DEFL('a) = emb oo (prj :: udom → 'a)
    by (rule cfun-eqI, simp add: defl emb-prj)
qed (simp-all only: liftemb liftprj liftdefl)
```

lemma *typedef-DEFL*:
  assumes *defl* ≡ (λ*a*::'*a*::*pcpo itself*. *t*)
  shows *DEFL*('*a*::*pcpo*) = *t*
unfolding *assms* **..**

Restore original typing constraints.

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

ML-file ‹*Tools/domaindef.ML*›

27.9 Isomorphic deflations

definition *isodefl* :: ('*a*::*domain* → '*a*) ⇒ *udom defl* ⇒ *bool*
  where *isodefl d t* ⟷ *cast*·*t* = *emb oo d oo prj*

definition *isodefl'* :: ('*a*::*predomain* → '*a*) ⇒ *udom u defl* ⇒ *bool*
  where *isodefl' d t* ⟷ *cast*·*t* = *liftemb oo u-map*·*d oo liftprj*

lemma *isodeflI*: (⋀*x*. *cast*·*t*·*x* = *emb*·(*d*·(*prj*·*x*))) ⟹ *isodefl d t*
unfolding *isodefl-def* **by** (*simp add*: *cfun-eqI*)

lemma *cast-isodefl*: *isodefl d t* ⟹ *cast*·*t* = (Λ *x*. *emb*·(*d*·(*prj*·*x*)))
unfolding *isodefl-def* **by** (*simp add*: *cfun-eqI*)

lemma *isodefl-strict*: *isodefl d t* ⟹ *d*·⊥ = ⊥
unfolding *isodefl-def*
by (*drule cfun-fun-cong* [**where** *x*=⊥], *simp*)

lemma *isodefl-imp-deflation*:
  fixes *d* :: '*a*::*domain* → '*a*
  assumes *isodefl d t* **shows** *deflation d*
proof
  note *assms* [*unfolded isodefl-def*, *simp*]
  fix *x* :: '*a*
  show *d*·(*d*·*x*) = *d*·*x*
    using *cast.idem* [*of t emb*·*x*] **by** *simp*
  show *d*·*x* ⊑ *x*
    using *cast.below* [*of t emb*·*x*] **by** *simp*
qed

lemma *isodefl-ID-DEFL*: *isodefl* (*ID* :: '*a* → '*a*) *DEFL*('*a*::*domain*)

unfolding *isodefl-def* **by** (*simp add*: *cast-DEFL*)

lemma *isodefl-LIFTDEFL*:
isodefl′ (*ID* :: *′a* → *′a*) *LIFTDEFL*(*′a*::*predomain*)
unfolding *isodefl′-def* **by** (*simp add*: *cast-liftdefl u-map-ID*)

lemma *isodefl-DEFL-imp-ID*: *isodefl* (*d* :: *′a* → *′a*) *DEFL*(*′a*::*domain*) ⟹ *d* = *ID*
unfolding *isodefl-def*
apply (*simp add*: *cast-DEFL*)
apply (*simp add*: *cfun-eq-iff*)
apply (*rule allI*)
apply (*drule-tac x*=*emb*·*x* **in** *spec*)
apply *simp*
done

lemma *isodefl-bottom*: *isodefl* ⊥ ⊥
unfolding *isodefl-def* **by** (*simp add*: *cfun-eq-iff*)

lemma *adm-isodefl*:
cont f ⟹ *cont g* ⟹ *adm* (*λx*. *isodefl* (*f x*) (*g x*))
unfolding *isodefl-def* **by** *simp*

lemma *isodefl-lub*:
assumes *chain d* **and** *chain t*
assumes ⋀*i*. *isodefl* (*d i*) (*t i*)
shows *isodefl* (⨆*i*. *d i*) (⨆*i*. *t i*)
using *assms* **unfolding** *isodefl-def*
by (*simp add*: *contlub-cfun-arg contlub-cfun-fun*)

lemma *isodefl-fix*:
assumes ⋀*d t*. *isodefl d t* ⟹ *isodefl* (*f*·*d*) (*g*·*t*)
shows *isodefl* (*fix*·*f*) (*fix*·*g*)
unfolding *fix-def2*
apply (*rule isodefl-lub*, *simp*, *simp*)
apply (*induct-tac i*)
apply (*simp add*: *isodefl-bottom*)
apply (*simp add*: *assms*)
done

lemma *isodefl-abs-rep*:
fixes *abs* **and** *rep* **and** *d*
assumes *DEFL*: *DEFL*(*′b*::*domain*) = *DEFL*(*′a*::*domain*)
assumes *abs-def*: (*abs* :: *′a* → *′b*) ≡ *prj oo emb*
assumes *rep-def*: (*rep* :: *′b* → *′a*) ≡ *prj oo emb*
shows *isodefl d t* ⟹ *isodefl* (*abs oo d oo rep*) *t*
unfolding *isodefl-def*
by (*simp add*: *cfun-eq-iff assms prj-emb-prj emb-prj-emb*)

lemma *isodefl′-liftdefl-of*: *isodefl d t* ⟹ *isodefl′ d* (*liftdefl-of*·*t*)
unfolding *isodefl-def isodefl′-def*
by (*simp add*: *cast-liftdefl-of u-map-oo liftemb-eq liftprj-eq*)

lemma *isodefl-sfun*:
  isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
    isodefl (*sfun-map*·*d1*·*d2*) (*sfun-defl*·*t1*·*t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-sfun-defl cast-isodefl*)
apply (*simp add*: *emb-sfun-def prj-sfun-def*)
apply (*simp add*: *sfun-map-map isodefl-strict*)
done

lemma *isodefl-ssum*:
  isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
    isodefl (*ssum-map*·*d1*·*d2*) (*ssum-defl*·*t1*·*t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-ssum-defl cast-isodefl*)
apply (*simp add*: *emb-ssum-def prj-ssum-def*)
apply (*simp add*: *ssum-map-map isodefl-strict*)
done

lemma *isodefl-sprod*:
  isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
    isodefl (*sprod-map*·*d1*·*d2*) (*sprod-defl*·*t1*·*t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-sprod-defl cast-isodefl*)
apply (*simp add*: *emb-sprod-def prj-sprod-def*)
apply (*simp add*: *sprod-map-map isodefl-strict*)
done

lemma *isodefl-prod*:
  isodefl d1 t1 ⟹ *isodefl d2 t2* ⟹
    isodefl (*prod-map*·*d1*·*d2*) (*prod-defl*·*t1*·*t2*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-prod-defl cast-isodefl*)
apply (*simp add*: *emb-prod-def prj-prod-def*)
apply (*simp add*: *prod-map-map cfcomp1*)
done

lemma *isodefl-u*:
  isodefl d t ⟹ *isodefl* (*u-map*·*d*) (*u-defl*·*t*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-u-defl cast-isodefl*)
apply (*simp add*: *emb-u-def prj-u-def liftemb-eq liftprj-eq u-map-map*)
done

lemma *isodefl-u-liftdefl*:
  isodefl′ d t ⟹ *isodefl* (*u-map*·*d*) (*u-liftdefl*·*t*)

apply (*rule isodeflI*)
apply (*simp add*: *cast-u-liftdefl isodefl′-def*)
apply (*simp add*: *emb-u-def prj-u-def liftemb-eq liftprj-eq*)
done

lemma *encode-prod-u-map*:
  encode-prod-u·(*u-map*·(*prod-map*·*f*·*g*)·(*decode-prod-u*·*x*))
    = *sprod-map*·(*u-map*·*f*)·(*u-map*·*g*)·*x*
unfolding *encode-prod-u-def decode-prod-u-def*
apply (*case-tac x*, *simp*, *rename-tac a b*)
apply (*case-tac a*, *simp*, *case-tac b*, *simp*, *simp*)
done

lemma *isodefl-prod-u*:
  assumes *isodefl′ d1 t1* **and** *isodefl′ d2 t2*
  shows *isodefl′* (*prod-map*·*d1*·*d2*) (*prod-liftdefl*·*t1*·*t2*)
using *assms* **unfolding** *isodefl′-def*
unfolding *liftemb-prod-def liftprj-prod-def*
by (*simp add*: *cast-prod-liftdefl cfcomp1 encode-prod-u-map sprod-map-map*)

lemma *encode-cfun-map*:
  encode-cfun·(*cfun-map*·*f*·*g*·(*decode-cfun*·*x*))
    = *sfun-map*·(*u-map*·*f*)·*g*·*x*
unfolding *encode-cfun-def decode-cfun-def*
apply (*simp add*: *sfun-eq-iff cfun-map-def sfun-map-def*)
apply (*rule cfun-eqI*, *rename-tac y*, *case-tac y*, *simp-all*)
done

lemma *isodefl-cfun*:
  assumes *isodefl* (*u-map*·*d1*) *t1* **and** *isodefl d2 t2*
  shows *isodefl* (*cfun-map*·*d1*·*d2*) (*sfun-defl*·*t1*·*t2*)
using *isodefl-sfun* [*OF assms*] **unfolding** *isodefl-def*
by (*simp add*: *emb-cfun-def prj-cfun-def cfcomp1 encode-cfun-map*)

27.10 Setting up the domain package

named-theorems *domain-defl-simps theorems like DEFL*(*′a t*) = *t-defl*$*DEFL*(*′a*)
 and *domain-isodefl theorems like isodefl d t ==> isodefl* (*foo-map*$*d*) (*foo-defl*$*t*)

ML-file ‹*Tools/Domain/domain-isomorphism.ML*›
ML-file ‹*Tools/Domain/domain-axioms.ML*›
ML-file ‹*Tools/Domain/domain.ML*›

lemmas [*domain-defl-simps*] =
  DEFL-cfun DEFL-sfun DEFL-ssum DEFL-sprod DEFL-prod DEFL-u
  liftdefl-eq LIFTDEFL-prod u-liftdefl-liftdefl-of

lemmas [*domain-map-ID*] =
  cfun-map-ID sfun-map-ID ssum-map-ID sprod-map-ID prod-map-ID u-map-ID

lemmas [*domain-isodefl*] =
isodefl-u isodefl-sfun isodefl-ssum isodefl-sprod
isodefl-cfun isodefl-prod isodefl-prod-u isodefl'-liftdefl-of
isodefl-u-liftdefl

lemmas [*domain-deflation*] =
deflation-cfun-map deflation-sfun-map deflation-ssum-map
deflation-sprod-map deflation-prod-map deflation-u-map

setup ‹
fold Domain-Take-Proofs.add-rec-type
[(***type-name*** ‹*cfun*›, [*true*, *true*]),
(***type-name*** ‹*sfun*›, [*true*, *true*]),
(***type-name*** ‹*ssum*›, [*true*, *true*]),
(***type-name*** ‹*sprod*›, [*true*, *true*]),
(***type-name*** ‹*prod*›, [*true*, *true*]),
(***type-name*** ‹*u*›, [*true*])]
›

end

28 A compact basis for powerdomains

theory *Compact-Basis*
imports *Universal*
begin

28.1 A compact basis for powerdomains

definition *pd-basis* = {*S*::'*a*::*bifinite compact-basis set*. *finite S* ∧ *S* ≠ {}}

typedef '*a*::*bifinite pd-basis* = *pd-basis* :: '*a compact-basis set set*
proof
show {*a*} ∈ *?pd-basis* **for** *a*
by (*simp add*: *pd-basis-def*)
qed

lemma *finite-Rep-pd-basis* [*simp*]: *finite* (*Rep-pd-basis u*)
using *Rep-pd-basis* [*of u*, *unfolded pd-basis-def*] **by** *simp*

lemma *Rep-pd-basis-nonempty* [*simp*]: *Rep-pd-basis u* ≠ {}
using *Rep-pd-basis* [*of u*, *unfolded pd-basis-def*] **by** *simp*

The powerdomain basis type is countable.

lemma *pd-basis-countable*: ∃*f*::'*a*::*bifinite pd-basis* ⇒ *nat*. *inj f* (**is** *Ex ?P*)
proof −
obtain *g* :: '*a compact-basis* ⇒ *nat* **where** *inj g*
using *compact-basis.countable* **..**

hence *image-g-eq*: g ' $A = g$ ' $B \longleftrightarrow A = B$ **for** A B
  by (*rule inj-image-eq-iff*)
have *inj* (λt. *set-encode* (g ' *Rep-pd-basis* t))
  by (*simp add*: *inj-on-def set-encode-eq image-g-eq Rep-pd-basis-inject*)
thus *?thesis* **by** (*rule exI* [*of ?P*])
qed

28.2 Unit and plus constructors

definition
  PDUnit :: '*a*::*bifinite compact-basis* ⇒ '*a pd-basis* **where**
  PDUnit = (λx. *Abs-pd-basis* $\{x\}$)

definition
  PDPlus :: '*a*::*bifinite pd-basis* ⇒ '*a pd-basis* ⇒ '*a pd-basis* **where**
  PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* ∪ *Rep-pd-basis u*)

lemma *Rep-PDUnit*:
  Rep-pd-basis (*PDUnit x*) = $\{x\}$
unfolding *PDUnit-def* **by** (*rule Abs-pd-basis-inverse*) (*simp add*: *pd-basis-def*)

lemma *Rep-PDPlus*:
  Rep-pd-basis (*PDPlus u v*) = *Rep-pd-basis u* ∪ *Rep-pd-basis v*
unfolding *PDPlus-def* **by** (*rule Abs-pd-basis-inverse*) (*simp add*: *pd-basis-def*)

lemma *PDUnit-inject* [*simp*]: (*PDUnit a* = *PDUnit b*) = ($a = b$)
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDUnit* **by** *simp*

lemma *PDPlus-assoc*: *PDPlus* (*PDPlus t u*) *v* = *PDPlus t* (*PDPlus u v*)
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-assoc*)

lemma *PDPlus-commute*: *PDPlus t u* = *PDPlus u t*
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-commute*)

lemma *PDPlus-absorb*: *PDPlus t t* = *t*
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-absorb*)

lemma *pd-basis-induct1* [*case-names PDUnit PDPlus*]:
  assumes *PDUnit*: $\bigwedge a$. P (*PDUnit a*)
  assumes *PDPlus*: $\bigwedge a\ t$. $P\ t \Longrightarrow P$ (*PDPlus* (*PDUnit a*) *t*)
  shows $P\ x$
proof (*induct x*)
  case (*Abs-pd-basis y*)
  then have *finite y* **and** $y \neq \{\}$ **by** (*simp-all add*: *pd-basis-def*)
  then show *?case*
  proof (*induct rule*: *finite-ne-induct*)
    case (*singleton x*)
    show *?case* **by** (*rule PDUnit* [*unfolded PDUnit-def*])
  next

```
  case (insert x F)
  from insert(4) have P (PDPlus (PDUnit x) (Abs-pd-basis F)) by (rule PDPlus)
  with insert(1,2) show ?case
   by (simp add: PDUnit-def PDPlus-def Abs-pd-basis-inverse [unfolded pd-basis-def])
 qed
qed

lemma pd-basis-induct [case-names PDUnit PDPlus]:
  assumes PDUnit: ⋀a. P (PDUnit a)
  assumes PDPlus: ⋀t u. ⟦P t; P u⟧ ⟹ P (PDPlus t u)
  shows P x
  by (induct x rule: pd-basis-induct1) (fact PDUnit, fact PDPlus [OF PDUnit])
```

28.3 Fold operator

```
definition
  fold-pd ::
    ('a::bifinite compact-basis ⇒ 'b::type) ⇒ ('b ⇒ 'b ⇒ 'b) ⇒ 'a pd-basis ⇒ 'b
  where fold-pd g f t = semilattice-set.F f (g ` Rep-pd-basis t)

lemma fold-pd-PDUnit:
  assumes semilattice f
  shows fold-pd g f (PDUnit x) = g x
proof −
  from assms interpret semilattice-set f by (rule semilattice-set.intro)
  show ?thesis by (simp add: fold-pd-def Rep-PDUnit)
qed

lemma fold-pd-PDPlus:
  assumes semilattice f
  shows fold-pd g f (PDPlus t u) = f (fold-pd g f t) (fold-pd g f u)
proof −
  from assms interpret semilattice-set f by (rule semilattice-set.intro)
  show ?thesis by (simp add: image-Un fold-pd-def Rep-PDPlus union)
qed

end
```

29 Upper powerdomain

```
theory UpperPD
imports Compact-Basis
begin
```

29.1 Basis preorder

```
definition
  upper-le :: 'a::bifinite pd-basis ⇒ 'a pd-basis ⇒ bool (infix ‹≤♯› 50) where
  upper-le = (λu v. ∀ y∈Rep-pd-basis v. ∃ x∈Rep-pd-basis u. x ⊑ y)
```

lemma *upper-le-refl* [*simp*]: $t \leq\sharp\ t$
unfolding *upper-le-def* **by** *fast*

lemma *upper-le-trans*: $[\![t \leq\sharp\ u;\ u \leq\sharp\ v]\!] \Longrightarrow t \leq\sharp\ v$
unfolding *upper-le-def*
apply (*rule ballI*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*erule rev-bexI*)
apply (*erule* (*1*) *below-trans*)
done

interpretation *upper-le*: *preorder upper-le*
by (*rule preorder.intro*, *rule upper-le-refl*, *rule upper-le-trans*)

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\sharp\ t$
unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-upper-mono*: $x \sqsubseteq y \Longrightarrow$ *PDUnit* $x \leq\sharp$ *PDUnit* y
unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDPlus-upper-mono*: $[\![s \leq\sharp\ t;\ u \leq\sharp\ v]\!] \Longrightarrow$ *PDPlus* $s\ u \leq\sharp$ *PDPlus* $t\ v$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-upper-le*: *PDPlus* $t\ u \leq\sharp\ t$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
  (*PDUnit* $a \leq\sharp$ *PDUnit* b) = ($a \sqsubseteq b$)
unfolding *upper-le-def Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-PDUnit-iff*:
  (*PDPlus* $t\ u \leq\sharp$ *PDUnit* a) = ($t \leq\sharp$ *PDUnit* $a \lor u \leq\sharp$ *PDUnit* a)
unfolding *upper-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-iff*: ($t \leq\sharp$ *PDPlus* $u\ v$) = ($t \leq\sharp\ u \land t \leq\sharp\ v$)
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-induct* [*induct set*: *upper-le*]:
  assumes *le*: $t \leq\sharp\ u$
  assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P$ (*PDUnit* a) (*PDUnit* b)
  assumes *2*: $\bigwedge t\ u\ a.\ P\ t$ (*PDUnit* a) $\Longrightarrow P$ (*PDPlus* $t\ u$) (*PDUnit* a)
  assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ t\ v]\!] \Longrightarrow P\ t$ (*PDPlus* $u\ v$)
  shows $P\ t\ u$
  using *le*
proof (*induct u arbitrary*: *t rule*: *pd-basis-induct*)
  case (*PDUnit a*)
  then show *?case*

```
  proof (induct t rule: pd-basis-induct)
    case PDUnit
    then show ?case by (simp add: 1)
  next
    case (PDPlus t u)
    from PDPlus(3) consider (t) t ≤♯ PDUnit a | (u) u ≤♯ PDUnit a
      by (auto simp: upper-le-PDPlus-PDUnit-iff)
    then show ?case
    proof cases
      case t
      then have P t (PDUnit a) by (rule PDPlus(1))
      then show ?thesis by (rule 2)
    next
      case u
      then have P u (PDUnit a) by (rule PDPlus(2))
      then have P (PDPlus u t) (PDUnit a) by (rule 2)
      then show ?thesis by (simp only: PDPlus-commute)
    qed
  qed
next
  case (PDPlus t t' u)
  then show ?case by (simp add: upper-le-PDPlus-iff 3)
qed
```

29.2 Type definition

```
typedef 'a::bifinite upper-pd (‹(‹notation=‹postfix upper-pd››'(-')♯)›) =
  {S::'a pd-basis set. upper-le.ideal S}
by (rule upper-le.ex-ideal)

instantiation upper-pd :: (bifinite) below
begin

definition
  x ⊑ y ⟷ Rep-upper-pd x ⊆ Rep-upper-pd y

instance ..
end

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

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

definition
  upper-principal :: 'a::bifinite pd-basis ⇒ 'a upper-pd where
```

upper-principal t = *Abs-upper-pd* {*u*. *u* ≤♯ *t*}

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

Upper powerdomain is pointed

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) ⊑ *ys*
by (*induct ys rule*: *upper-pd.principal-induct*, *simp*, *simp*)

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

lemma *inst-upper-pd-pcpo*: ⊥ = *upper-principal* (*PDUnit compact-bot*)
by (*rule upper-pd-minimal* [*THEN bottomI*, *symmetric*])

29.3 Monadic unit and plus

definition
upper-unit :: *'a*::*bifinite* → *'a upper-pd* **where**
upper-unit = *compact-basis.extension* (λ*a*. *upper-principal* (*PDUnit a*))

definition
upper-plus :: *'a*::*bifinite upper-pd* → *'a upper-pd* → *'a upper-pd* **where**
upper-plus = *upper-pd.extension* (λ*t*. *upper-pd.extension* (λ*u*.
upper-principal (*PDPlus t u*)))

abbreviation
upper-add :: *'a*::*bifinite upper-pd* ⇒ *'a upper-pd* ⇒ *'a upper-pd*
(**infixl** ‹∪♯› *65*) **where**
xs ∪♯ *ys* == *upper-plus*·*xs*·*ys*

syntax
-upper-pd :: *args* ⇒ *logic* (‹(‹indent=1 notation=‹mixfix upper-pd enumeration››{-}♯)›)
translations
{*x*,*xs*}♯ == {*x*}♯ ∪♯ {*xs*}♯
{*x*}♯ == *CONST upper-unit*·*x*

lemma *upper-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♯ = *upper-principal* (*PDUnit a*)
unfolding *upper-unit-def*
by (*simp add*: *compact-basis.extension-principal PDUnit-upper-mono*)

lemma *upper-plus-principal* [*simp*]:
upper-principal t ∪♯ *upper-principal u* = *upper-principal* (*PDPlus t u*)
unfolding *upper-plus-def*

```
by (simp add: upper-pd.extension-principal
    upper-pd.extension-mono PDPlus-upper-mono)

interpretation upper-add: semilattice upper-add proof
  fix xs ys zs :: 'a upper-pd
  show (xs ∪♯ ys) ∪♯ zs = xs ∪♯ (ys ∪♯ zs)
    apply (induct xs rule: upper-pd.principal-induct, simp)
    apply (induct ys rule: upper-pd.principal-induct, simp)
    apply (induct zs rule: upper-pd.principal-induct, simp)
    apply (simp add: PDPlus-assoc)
    done
  show xs ∪♯ ys = ys ∪♯ xs
    apply (induct xs rule: upper-pd.principal-induct, simp)
    apply (induct ys rule: upper-pd.principal-induct, simp)
    apply (simp add: PDPlus-commute)
    done
  show xs ∪♯ xs = xs
    apply (induct xs rule: upper-pd.principal-induct, simp)
    apply (simp add: PDPlus-absorb)
    done
qed

lemmas upper-plus-assoc = upper-add.assoc
lemmas upper-plus-commute = upper-add.commute
lemmas upper-plus-absorb = upper-add.idem
lemmas upper-plus-left-commute = upper-add.left-commute
lemmas upper-plus-left-absorb = upper-add.left-idem
```

Useful for *simp add*: *upper-plus-ac*

```
lemmas upper-plus-ac =
  upper-plus-assoc upper-plus-commute upper-plus-left-commute
```

Useful for *simp only*: *upper-plus-aci*

```
lemmas upper-plus-aci =
  upper-plus-ac upper-plus-absorb upper-plus-left-absorb

lemma upper-plus-below1: xs ∪♯ ys ⊑ xs
apply (induct xs rule: upper-pd.principal-induct, simp)
apply (induct ys rule: upper-pd.principal-induct, simp)
apply (simp add: PDPlus-upper-le)
done

lemma upper-plus-below2: xs ∪♯ ys ⊑ ys
by (subst upper-plus-commute, rule upper-plus-below1)

lemma upper-plus-greatest: [[xs ⊑ ys; xs ⊑ zs]] ⟹ xs ⊑ ys ∪♯ zs
apply (subst upper-plus-absorb [of xs, symmetric])
apply (erule (1) monofun-cfun [OF monofun-cfun-arg])
done
```

lemma *upper-below-plus-iff* [*simp*]:
  $xs \sqsubseteq ys \cup\sharp\ zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
apply *safe*
apply (*erule below-trans* [*OF - upper-plus-below1*])
apply (*erule below-trans* [*OF - upper-plus-below2*])
apply (*erule* (*1*) *upper-plus-greatest*)
done

lemma *upper-plus-below-unit-iff* [*simp*]:
  $xs \cup\sharp\ ys \sqsubseteq \{z\}\sharp \longleftrightarrow xs \sqsubseteq \{z\}\sharp \vee ys \sqsubseteq \{z\}\sharp$
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct z rule*: *compact-basis.principal-induct*, *simp*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)
done

lemma *upper-unit-below-iff* [*simp*]: $\{x\}\sharp \sqsubseteq \{y\}\sharp \longleftrightarrow x \sqsubseteq y$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct y rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
done

lemmas *upper-pd-below-simps* =
  upper-unit-below-iff
  upper-below-plus-iff
  upper-plus-below-unit-iff

lemma *upper-unit-eq-iff* [*simp*]: $\{x\}\sharp = \{y\}\sharp \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *upper-unit-strict* [*simp*]: $\{\bot\}\sharp = \bot$
using *upper-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add*: *inst-upper-pd-pcpo*)

lemma *upper-plus-strict1* [*simp*]: $\bot \cup\sharp\ ys = \bot$
by (*rule bottomI*, *rule upper-plus-below1*)

lemma *upper-plus-strict2* [*simp*]: $xs \cup\sharp\ \bot = \bot$
by (*rule bottomI*, *rule upper-plus-below2*)

lemma *upper-unit-bottom-iff* [*simp*]: $\{x\}\sharp = \bot \longleftrightarrow x = \bot$
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-unit-eq-iff*)

lemma *upper-plus-bottom-iff* [*simp*]:
  $xs \cup\sharp\ ys = \bot \longleftrightarrow xs = \bot \vee ys = \bot$
apply (*induct xs rule*: *upper-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *upper-pd.principal-induct*, *simp*)
apply (*simp add*: *inst-upper-pd-pcpo upper-pd.principal-eq-iff*

upper-le-PDPlus-PDUnit-iff)
done

lemma *compact-upper-unit*: *compact x* ⟹ *compact* {*x*}♯
by (*auto dest*!: *compact-basis.compact-imp-principal*)

lemma *compact-upper-unit-iff* [*simp*]: *compact* {*x*}♯ ⟷ *compact x*
apply (*safe elim*!: *compact-upper-unit*)
apply (*simp only*: *compact-def upper-unit-below-iff* [*symmetric*])
apply (*erule adm-subst* [*OF cont-Rep-cfun2*])
done

lemma *compact-upper-plus* [*simp*]:
⟦*compact xs*; *compact ys*⟧ ⟹ *compact* (*xs* ∪♯ *ys*)
by (*auto dest*!: *upper-pd.compact-imp-principal*)

29.4 Induction rules

lemma *upper-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: ⋀*x*. *P* {*x*}♯
assumes *insert*: ⋀*x ys*. ⟦*P* {*x*}♯; *P ys*⟧ ⟹ *P* ({*x*}♯ ∪♯ *ys*)
shows *P* (*xs*::′*a*::*bifinite upper-pd*)
proof (*induct xs rule*: *upper-pd.principal-induct*)
have ∗: *P* {*Rep-compact-basis a*}♯ **for** *a*
by (*rule unit*)
show *P* (*upper-principal a*) **for** *a*
proof (*induct a rule*: *pd-basis-induct1*)
case (*PDUnit a*)
with ∗ **show** *?case*
by (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*])
next
case (*PDPlus a t*)
with ∗ **have** *P* ({*Rep-compact-basis a*}♯ ∪♯ *upper-principal t*)
by (*rule insert*)
then show *?case*
by (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*]
upper-plus-principal [*symmetric*])
qed
qed (*rule P*)

lemma *upper-pd-induct* [*case-names adm upper-unit upper-plus*, *induct type*: *upper-pd*]:
assumes *P*: *adm P*
assumes *unit*: ⋀*x*. *P* {*x*}♯
assumes *plus*: ⋀*xs ys*. ⟦*P xs*; *P ys*⟧ ⟹ *P* (*xs* ∪♯ *ys*)
shows *P* (*xs*::′*a*::*bifinite upper-pd*)
proof (*induct xs rule*: *upper-pd.principal-induct*)
show *P* (*upper-principal a*) **for** *a*

proof (*induct a rule*: *pd-basis-induct*)
  case *PDUnit*
  then show *?case*
    by (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*] *unit*)
next
  case *PDPlus*
  then show *?case*
    by (*simp only*: *upper-plus-principal* [*symmetric*] *plus*)
qed
qed (*rule P*)

29.5 Monadic bind

definition
  upper-bind-basis ::
  $'a$::*bifinite pd-basis* $\Rightarrow$ ($'a \rightarrow 'b$ *upper-pd*) $\rightarrow$ $'b$::*bifinite upper-pd* **where**
  upper-bind-basis = *fold-pd*
    (λa. Λ *f*. *f*·(*Rep-compact-basis a*))
    (λx *y*. Λ *f*. *x*·*f* $\cup\sharp$ *y*·*f*)

lemma *ACI-upper-bind*:
  semilattice (λx *y*. Λ *f*. *x*·*f* $\cup\sharp$ *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* $\cup\sharp$ *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*:
  $t \leq\sharp u \Longrightarrow$ *upper-bind-basis t* $\sqsubseteq$ *upper-bind-basis u*
unfolding *cfun-below-iff*
apply (*erule upper-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *below-trans* [*OF upper-plus-below1*])
apply *simp*
done

definition

upper-bind :: *'a*::*bifinite upper-pd* → (*'a* → *'b upper-pd*) → *'b*::*bifinite upper-pd*
where
upper-bind = *upper-pd.extension upper-bind-basis*

syntax
-upper-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
(‹(‹*indent=3 notation=*‹*binder upper-bind*››⋃♯-∈-./ -)› [*0*, *0*, *10*] *10*)

translations
⋃♯*x*∈*xs*. *e* == *CONST upper-bind*·*xs*·(Λ *x*. *e*)

lemma *upper-bind-principal* [*simp*]:
upper-bind·(*upper-principal t*) = *upper-bind-basis t*
unfolding *upper-bind-def*
apply (*rule upper-pd.extension-principal*)
apply (*erule upper-bind-basis-mono*)
done

lemma *upper-bind-unit* [*simp*]:
upper-bind·{*x*}♯·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *upper-bind-plus* [*simp*]:
upper-bind·(*xs* ∪♯ *ys*)·*f* = *upper-bind*·*xs*·*f* ∪♯ *upper-bind*·*ys*·*f*
by (*induct xs rule*: *upper-pd.principal-induct*, *simp*,
induct ys rule: *upper-pd.principal-induct*, *simp*, *simp*)

lemma *upper-bind-strict* [*simp*]: *upper-bind*·⊥·*f* = *f*·⊥
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-bind-unit*)

lemma *upper-bind-bind*:
upper-bind·(*upper-bind*·*xs*·*f*)·*g* = *upper-bind*·*xs*·(Λ *x*. *upper-bind*·(*f*·*x*)·*g*)
by (*induct xs*, *simp-all*)

29.6 Map

definition
upper-map :: (*'a*::*bifinite* → *'b*::*bifinite*) → *'a upper-pd* → *'b upper-pd* **where**
upper-map = (Λ *f xs*. *upper-bind*·*xs*·(Λ *x*. {*f*·*x*}♯))

lemma *upper-map-unit* [*simp*]:
upper-map·*f*·{*x*}♯ = {*f*·*x*}♯
unfolding *upper-map-def* **by** *simp*

lemma *upper-map-plus* [*simp*]:
upper-map·*f*·(*xs* ∪♯ *ys*) = *upper-map*·*f*·*xs* ∪♯ *upper-map*·*f*·*ys*
unfolding *upper-map-def* **by** *simp*

lemma *upper-map-bottom* [*simp*]: *upper-map*·*f*·⊥ = {*f*·⊥}♯

```
unfolding upper-map-def by simp

lemma upper-map-ident: upper-map·(Λ x. x)·xs = xs
by (induct xs rule: upper-pd-induct, simp-all)

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

lemma upper-map-map:
  upper-map·f·(upper-map·g·xs) = upper-map·(Λ x. f·(g·x))·xs
by (induct xs rule: upper-pd-induct, simp-all)

lemma upper-bind-map:
  upper-bind·(upper-map·f·xs)·g = upper-bind·xs·(Λ x. g·(f·x))
by (simp add: upper-map-def upper-bind-bind)

lemma upper-map-bind:
  upper-map·f·(upper-bind·xs·g) = upper-bind·xs·(Λ x. upper-map·f·(g·x))
by (simp add: upper-map-def upper-bind-bind)

lemma ep-pair-upper-map: ep-pair e p ⟹ ep-pair (upper-map·e) (upper-map·p)
apply standard
apply (induct-tac x rule: upper-pd-induct, simp-all add: ep-pair.e-inverse)
apply (induct-tac y rule: upper-pd-induct)
apply (simp-all add: ep-pair.e-p-below monofun-cfun del: upper-below-plus-iff)
done

lemma deflation-upper-map: deflation d ⟹ deflation (upper-map·d)
apply standard
apply (induct-tac x rule: upper-pd-induct, simp-all add: deflation.idem)
apply (induct-tac x rule: upper-pd-induct)
apply (simp-all add: deflation.below monofun-cfun del: upper-below-plus-iff)
done


lemma finite-deflation-upper-map:
  assumes finite-deflation d shows finite-deflation (upper-map·d)
proof (rule finite-deflation-intro)
  interpret d: finite-deflation d by fact
  from d.deflation-axioms show deflation (upper-map·d)
    by (rule deflation-upper-map)
  have finite (range (λx. d·x)) by (rule d.finite-range)
  hence finite (Rep-compact-basis −‘ range (λx. d·x))
    by (rule finite-vimageI, simp add: inj-on-def Rep-compact-basis-inject)
  hence finite (Pow (Rep-compact-basis −‘ range (λx. d·x))) by simp
  hence finite (Rep-pd-basis −‘ (Pow (Rep-compact-basis −‘ range (λx. d·x))))
    by (rule finite-vimageI, simp add: inj-on-def Rep-pd-basis-inject)
  hence *: finite (upper-principal ‘ Rep-pd-basis −‘ (Pow (Rep-compact-basis −‘
range (λx. d·x)))) by simp
```

```
  hence finite (range (λxs. upper-map·d·xs))
    apply (rule rev-finite-subset)
    apply clarsimp
    apply (induct-tac xs rule: upper-pd.principal-induct)
    apply (simp add: adm-mem-finite ∗)
    apply (rename-tac t, induct-tac t rule: pd-basis-induct)
    apply (simp only: upper-unit-Rep-compact-basis [symmetric] upper-map-unit)
    apply simp
    apply (subgoal-tac ∃ b. d·(Rep-compact-basis a) = Rep-compact-basis b)
    apply clarsimp
    apply (rule imageI)
    apply (rule vimageI2)
    apply (simp add: Rep-PDUnit)
    apply (rule range-eqI)
    apply (erule sym)
    apply (rule exI)
    apply (rule Abs-compact-basis-inverse [symmetric])
    apply (simp add: d.compact)
    apply (simp only: upper-plus-principal [symmetric] upper-map-plus)
    apply clarsimp
    apply (rule imageI)
    apply (rule vimageI2)
    apply (simp add: Rep-PDPlus)
    done
  thus finite {xs. upper-map·d·xs = xs}
    by (rule finite-range-imp-finite-fixes)
qed
```

29.7 Upper powerdomain is bifinite

```
lemma approx-chain-upper-map:
  assumes approx-chain a
  shows approx-chain (λi. upper-map·(a i))
  using assms unfolding approx-chain-def
  by (simp add: lub-APP upper-map-ID finite-deflation-upper-map)

instance upper-pd :: (bifinite) bifinite
proof
  show ∃ (a::nat ⇒ 'a upper-pd → 'a upper-pd). approx-chain a
    using bifinite [where 'a='a]
    by (fast intro!: approx-chain-upper-map)
qed
```

29.8 Join

```
definition
  upper-join :: 'a::bifinite upper-pd upper-pd → 'a upper-pd where
  upper-join = (Λ xss. upper-bind·xss·(Λ xs. xs))

lemma upper-join-unit [simp]:
```

```
  upper-join·{xs}♯ = xs
unfolding upper-join-def by simp

lemma upper-join-plus [simp]:
  upper-join·(xss ∪♯ yss) = upper-join·xss ∪♯ upper-join·yss
unfolding upper-join-def by simp

lemma upper-join-bottom [simp]: upper-join·⊥ = ⊥
unfolding upper-join-def by simp

lemma upper-join-map-unit:
  upper-join·(upper-map·upper-unit·xs) = xs
by (induct xs rule: upper-pd-induct, simp-all)

lemma upper-join-map-join:
  upper-join·(upper-map·upper-join·xsss) = upper-join·(upper-join·xsss)
by (induct xsss rule: upper-pd-induct, simp-all)

lemma upper-join-map-map:
  upper-join·(upper-map·(upper-map·f)·xss) =
   upper-map·f·(upper-join·xss)
by (induct xss rule: upper-pd-induct, simp-all)

end
```

30 Lower powerdomain

```
theory LowerPD
imports Compact-Basis
begin
```

30.1 Basis preorder

```
definition
  lower-le :: 'a::bifinite pd-basis ⇒ 'a pd-basis ⇒ bool (infix ‹≤♭› 50) where
  lower-le = (λu 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) below-trans)
done
```

interpretation *lower-le*: *preorder lower-le*
by (*rule preorder.intro*, *rule lower-le-refl*, *rule lower-le-trans*)

lemma *lower-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\flat$ *t*
unfolding *lower-le-def Rep-PDUnit*
by (*simp*, *rule Rep-pd-basis-nonempty* [*folded ex-in-conv*])

lemma *PDUnit-lower-mono*: $x \sqsubseteq y \Longrightarrow$ *PDUnit x* $\leq\flat$ *PDUnit y*
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *PDPlus-lower-mono*: $[\![s \leq\flat t;\ u \leq\flat v]\!] \Longrightarrow$ *PDPlus s u* $\leq\flat$ *PDPlus t v*
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-lower-le*: *t* $\leq\flat$ *PDPlus t u*
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-PDUnit-PDUnit-iff* [*simp*]:
  (*PDUnit a* $\leq\flat$ *PDUnit b*) = ($a \sqsubseteq b$)
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *lower-le-PDUnit-PDPlus-iff*:
  (*PDUnit a* $\leq\flat$ *PDPlus t u*) = (*PDUnit a* $\leq\flat$ *t* $\vee$ *PDUnit a* $\leq\flat$ *u*)
unfolding *lower-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *lower-le-PDPlus-iff*: (*PDPlus t u* $\leq\flat$ *v*) = (*t* $\leq\flat$ *v* $\wedge$ *u* $\leq\flat$ *v*)
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-induct* [*induct set*: *lower-le*]:
  assumes *le*: *t* $\leq\flat$ *u*
  assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow$ *P* (*PDUnit a*) (*PDUnit b*)
  assumes *2*: $\bigwedge t\ u\ a.$ *P* (*PDUnit a*) *t* $\Longrightarrow$ *P* (*PDUnit a*) (*PDPlus t u*)
  assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ v;\ P\ u\ v]\!] \Longrightarrow$ *P* (*PDPlus t u*) *v*
  shows *P t u*
  using *le*
proof (*induct t arbitrary*: *u rule*: *pd-basis-induct*)
  case (*PDUnit a*)
  then show *?case*
  proof (*induct u rule*: *pd-basis-induct*)
    case *PDUnit*
    then show *?case* **by** (*simp add*: *1*)
  next
    case (*PDPlus t u*)
    from *PDPlus*(*3*) **consider** (*t*) *PDUnit a* $\leq\flat$ *t* | (*u*) *PDUnit a* $\leq\flat$ *u*
      by (*auto simp*: *lower-le-PDUnit-PDPlus-iff*)
    then show *?case*
    proof *cases*
      case *t*
      then have *P* (*PDUnit a*) *t* **by** (*rule PDPlus*(*1*))
      then show *?thesis* **by** (*rule 2*)

```
    next
      case u
      then have P (PDUnit a) u by (rule PDPlus(2))
      then have P (PDUnit a) (PDPlus u t) by (rule 2)
      then show ?thesis by (simp only: PDPlus-commute)
    qed
  qed
next
  case (PDPlus t t')
  then show ?case by (simp add: lower-le-PDPlus-iff 3)
qed
```

30.2 Type definition

```
typedef 'a::bifinite lower-pd (‹(‹notation=‹postfix lower-pd››'(-')♭)›) =
  {S::'a pd-basis set. lower-le.ideal S}
by (rule lower-le.ex-ideal)

instantiation lower-pd :: (bifinite) below
begin

definition
  x ⊑ y ⟷ Rep-lower-pd x ⊆ Rep-lower-pd y

instance ..
end

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

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

definition
  lower-principal :: 'a::bifinite pd-basis ⇒ 'a lower-pd where
  lower-principal t = Abs-lower-pd {u. u ≤♭ t}

interpretation lower-pd:
  ideal-completion lower-le lower-principal Rep-lower-pd
using type-definition-lower-pd below-lower-pd-def
using lower-principal-def pd-basis-countable
by (rule lower-le.typedef-ideal-completion)
```

Lower powerdomain is pointed

```
lemma lower-pd-minimal: lower-principal (PDUnit compact-bot) ⊑ ys
by (induct ys rule: lower-pd.principal-induct, simp, simp)
```

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

lemma *inst-lower-pd-pcpo*: $\bot$ = *lower-principal* (*PDUnit compact-bot*)
by (*rule lower-pd-minimal* [*THEN bottomI*, *symmetric*])

30.3 Monadic unit and plus

definition
lower-unit :: *'a*::*bifinite* $\rightarrow$ *'a lower-pd* **where**
lower-unit = *compact-basis.extension* (λ*a*. *lower-principal* (*PDUnit a*))

definition
lower-plus :: *'a*::*bifinite lower-pd* $\rightarrow$ *'a lower-pd* $\rightarrow$ *'a lower-pd* **where**
lower-plus = *lower-pd.extension* (λ*t*. *lower-pd.extension* (λ*u*.
lower-principal (*PDPlus t u*)))

abbreviation
lower-add :: *'a*::*bifinite lower-pd* $\Rightarrow$ *'a lower-pd* $\Rightarrow$ *'a lower-pd*
(**infixl** ‹∪♭› *65*) **where**
xs ∪♭ *ys* == *lower-plus*·*xs*·*ys*

syntax
-lower-pd :: *args* $\Rightarrow$ *logic* (‹(‹*indent=1 notation=*‹*mixfix lower-pd enumeration*››{-}♭)›)
translations
{*x*,*xs*}♭ == {*x*}♭ ∪♭ {*xs*}♭
{*x*}♭ == *CONST lower-unit*·*x*

lemma *lower-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♭ = *lower-principal* (*PDUnit a*)
unfolding *lower-unit-def*
by (*simp add*: *compact-basis.extension-principal PDUnit-lower-mono*)

lemma *lower-plus-principal* [*simp*]:
lower-principal t ∪♭ *lower-principal u* = *lower-principal* (*PDPlus t u*)
unfolding *lower-plus-def*
by (*simp add*: *lower-pd.extension-principal*
lower-pd.extension-mono PDPlus-lower-mono)

interpretation *lower-add*: *semilattice lower-add* **proof**
fix *xs ys zs* :: *'a*::*bifinite lower-pd*
show (*xs* ∪♭ *ys*) ∪♭ *zs* = *xs* ∪♭ (*ys* ∪♭ *zs*)
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct zs rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done
show *xs* ∪♭ *ys* = *ys* ∪♭ *xs*

apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done
show $xs \cup\flat\ xs = xs$
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done
qed

lemmas *lower-plus-assoc* = *lower-add.assoc*
lemmas *lower-plus-commute* = *lower-add.commute*
lemmas *lower-plus-absorb* = *lower-add.idem*
lemmas *lower-plus-left-commute* = *lower-add.left-commute*
lemmas *lower-plus-left-absorb* = *lower-add.left-idem*

Useful for *simp add*: *lower-plus-ac*

lemmas *lower-plus-ac* =
lower-plus-assoc lower-plus-commute lower-plus-left-commute

Useful for *simp only*: *lower-plus-aci*

lemmas *lower-plus-aci* =
lower-plus-ac lower-plus-absorb lower-plus-left-absorb

lemma *lower-plus-below1*: $xs \sqsubseteq xs \cup\flat\ ys$
apply (*induct xs rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *PDPlus-lower-le*)
done

lemma *lower-plus-below2*: $ys \sqsubseteq xs \cup\flat\ ys$
by (*subst lower-plus-commute*, *rule lower-plus-below1*)

lemma *lower-plus-least*: $[\![xs \sqsubseteq zs;\ ys \sqsubseteq zs]\!] \Longrightarrow xs \cup\flat\ ys \sqsubseteq zs$
apply (*subst lower-plus-absorb* [*of zs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *lower-plus-below-iff* [*simp*]:
$xs \cup\flat\ ys \sqsubseteq zs \longleftrightarrow xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
apply *safe*
apply (*erule below-trans* [*OF lower-plus-below1*])
apply (*erule below-trans* [*OF lower-plus-below2*])
apply (*erule* (*1*) *lower-plus-least*)
done

lemma *lower-unit-below-plus-iff* [*simp*]:
$\{x\}\flat \sqsubseteq ys \cup\flat\ zs \longleftrightarrow \{x\}\flat \sqsubseteq ys \vee \{x\}\flat \sqsubseteq zs$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)

apply (*induct ys rule*: *lower-pd.principal-induct*, *simp*)
apply (*induct zs rule*: *lower-pd.principal-induct*, *simp*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
done

lemma *lower-unit-below-iff* [*simp*]: $\{x\}\flat \sqsubseteq \{y\}\flat \longleftrightarrow x \sqsubseteq y$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct y rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
done

lemmas *lower-pd-below-simps* =
lower-unit-below-iff
lower-plus-below-iff
lower-unit-below-plus-iff

lemma *lower-unit-eq-iff* [*simp*]: $\{x\}\flat = \{y\}\flat \longleftrightarrow x = y$
by (*simp add*: *po-eq-conv*)

lemma *lower-unit-strict* [*simp*]: $\{\bot\}\flat = \bot$
using *lower-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add*: *inst-lower-pd-pcpo*)

lemma *lower-unit-bottom-iff* [*simp*]: $\{x\}\flat = \bot \longleftrightarrow x = \bot$
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-unit-eq-iff*)

lemma *lower-plus-bottom-iff* [*simp*]:
$xs \cup\flat\ ys = \bot \longleftrightarrow xs = \bot \wedge ys = \bot$
apply *safe*
apply (*rule bottomI*, *erule subst*, *rule lower-plus-below1*)
apply (*rule bottomI*, *erule subst*, *rule lower-plus-below2*)
apply (*rule lower-plus-absorb*)
done

lemma *lower-plus-strict1* [*simp*]: $\bot \cup\flat\ ys = ys$
apply (*rule below-antisym* [*OF - lower-plus-below2*])
apply (*simp add*: *lower-plus-least*)
done

lemma *lower-plus-strict2* [*simp*]: $xs \cup\flat \bot = xs$
apply (*rule below-antisym* [*OF - lower-plus-below1*])
apply (*simp add*: *lower-plus-least*)
done

lemma *compact-lower-unit*: *compact x* $\Longrightarrow$ *compact* $\{x\}\flat$
by (*auto dest*!: *compact-basis.compact-imp-principal*)

lemma *compact-lower-unit-iff* [*simp*]: *compact* $\{x\}\flat \longleftrightarrow$ *compact x*
apply (*safe elim*!: *compact-lower-unit*)

apply (*simp only*: *compact-def lower-unit-below-iff* [*symmetric*])
apply (*erule adm-subst* [*OF cont-Rep-cfun2*])
done

lemma *compact-lower-plus* [*simp*]:
  ⟦*compact xs*; *compact ys*⟧ ⟹ *compact* (*xs* ∪♭ *ys*)
by (*auto dest*!: *lower-pd.compact-imp-principal*)

30.4 Induction rules

lemma *lower-pd-induct1*:
  assumes *P*: *adm P*
  assumes *unit*: ⋀*x*. *P* {*x*}♭
  assumes *insert*: ⋀*x ys*. ⟦*P* {*x*}♭; *P ys*⟧ ⟹ *P* ({*x*}♭ ∪♭ *ys*)
  shows *P* (*xs*::'*a*::*bifinite lower-pd*)
proof (*induct xs rule*: *lower-pd.principal-induct*)
  have ∗: *P* {*Rep-compact-basis a*}♭ **for** *a*
    by (*rule unit*)
  show *P* (*lower-principal a*) **for** *a*
  proof (*induct a rule*: *pd-basis-induct1*)
    case *PDUnit*
    from ∗ **show** *?case*
      by (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*])
  next
    case (*PDPlus a t*)
    with ∗ **have** *P* ({*Rep-compact-basis a*}♭ ∪♭ *lower-principal t*)
      by (*rule insert*)
    then show *?case*
      by (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *lower-plus-principal* [*symmetric*])
  qed
qed (*rule P*)

lemma *lower-pd-induct* [*case-names adm lower-unit lower-plus*, *induct type*: *lower-pd*]:
  assumes *P*: *adm P*
  assumes *unit*: ⋀*x*. *P* {*x*}♭
  assumes *plus*: ⋀*xs ys*. ⟦*P xs*; *P ys*⟧ ⟹ *P* (*xs* ∪♭ *ys*)
  shows *P* (*xs*::'*a*::*bifinite lower-pd*)
proof (*induct xs rule*: *lower-pd.principal-induct*)
  show *P* (*lower-principal a*) **for** *a*
  proof (*induct a rule*: *pd-basis-induct*)
    case *PDUnit*
    then show *?case*
      by (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *unit*)
  next
    case *PDPlus*
    then show *?case*
      by (*simp only*: *lower-plus-principal* [*symmetric*] *plus*)
  qed

qed (*rule P*)

30.5 Monadic bind

definition
  lower-bind-basis ::
  'a::bifinite pd-basis ⇒ (*'a* → *'b lower-pd*) → *'b::bifinite lower-pd* **where**
  lower-bind-basis = *fold-pd*
    (λ*a*. Λ *f*. *f*·(*Rep-compact-basis a*))
    (λ*x y*. Λ *f*. *x*·*f* ∪♭ *y*·*f*)

lemma *ACI-lower-bind*:
  semilattice (λ*x y*. Λ *f*. *x*·*f* ∪♭ *y*·*f*)
apply *unfold-locales*
apply (*simp add*: *lower-plus-assoc*)
apply (*simp add*: *lower-plus-commute*)
apply (*simp add*: *eta-cfun*)
done

lemma *lower-bind-basis-simps* [*simp*]:
  lower-bind-basis (*PDUnit a*) =
    (Λ *f*. *f*·(*Rep-compact-basis a*))
  lower-bind-basis (*PDPlus t u*) =
    (Λ *f*. *lower-bind-basis t*·*f* ∪♭ *lower-bind-basis u*·*f*)
unfolding *lower-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-lower-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-lower-bind*])
done

lemma *lower-bind-basis-mono*:
  t ≤♭ *u* ⟹ *lower-bind-basis t* ⊑ *lower-bind-basis u*
unfolding *cfun-below-iff*
apply (*erule lower-le-induct*, *safe*)
apply (*simp add*: *monofun-cfun*)
apply (*simp add*: *rev-below-trans* [*OF lower-plus-below1*])
apply *simp*
done

definition
  lower-bind :: *'a::bifinite lower-pd* → (*'a* → *'b lower-pd*) → *'b::bifinite lower-pd*
where
  lower-bind = *lower-pd.extension lower-bind-basis*

syntax
  -lower-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
    (‹(‹*indent=3 notation*=‹*binder lower-bind*››⋃♭-∈-./ -)› [*0*, *0*, *10*] *10*)

translations

$\bigcup\flat x{\in}xs.\ e$ == *CONST lower-bind*·*xs*·(Λ *x*. *e*)

lemma *lower-bind-principal* [*simp*]:
lower-bind·(*lower-principal t*) = *lower-bind-basis t*
unfolding *lower-bind-def*
apply (*rule lower-pd.extension-principal*)
apply (*erule lower-bind-basis-mono*)
done

lemma *lower-bind-unit* [*simp*]:
lower-bind·{*x*}♭·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *lower-bind-plus* [*simp*]:
lower-bind·(*xs* ∪♭ *ys*)·*f* = *lower-bind*·*xs*·*f* ∪♭ *lower-bind*·*ys*·*f*
by (*induct xs rule*: *lower-pd.principal-induct*, *simp*,
induct ys rule: *lower-pd.principal-induct*, *simp*, *simp*)

lemma *lower-bind-strict* [*simp*]: *lower-bind*·⊥·*f* = *f*·⊥
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-bind-unit*)

lemma *lower-bind-bind*:
lower-bind·(*lower-bind*·*xs*·*f*)·*g* = *lower-bind*·*xs*·(Λ *x*. *lower-bind*·(*f*·*x*)·*g*)
by (*induct xs*, *simp-all*)

30.6 Map

definition
lower-map :: (*'a*::*bifinite* → *'b*::*bifinite*) → *'a lower-pd* → *'b lower-pd* **where**
lower-map = (Λ *f xs*. *lower-bind*·*xs*·(Λ *x*. {*f*·*x*}♭))

lemma *lower-map-unit* [*simp*]:
lower-map·*f*·{*x*}♭ = {*f*·*x*}♭
unfolding *lower-map-def* **by** *simp*

lemma *lower-map-plus* [*simp*]:
lower-map·*f*·(*xs* ∪♭ *ys*) = *lower-map*·*f*·*xs* ∪♭ *lower-map*·*f*·*ys*
unfolding *lower-map-def* **by** *simp*

lemma *lower-map-bottom* [*simp*]: *lower-map*·*f*·⊥ = {*f*·⊥}♭
unfolding *lower-map-def* **by** *simp*

lemma *lower-map-ident*: *lower-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-map-ID*: *lower-map*·*ID* = *ID*
by (*simp add*: *cfun-eq-iff ID-def lower-map-ident*)

lemma *lower-map-map*:

lower-map·f·(lower-map·g·xs) = lower-map·(Λ x. f·(g·x))·xs
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-bind-map*:
lower-bind·(lower-map·f·xs)·g = lower-bind·xs·(Λ x. g·(f·x))
by (*simp add*: *lower-map-def lower-bind-bind*)

lemma *lower-map-bind*:
lower-map·f·(lower-bind·xs·g) = lower-bind·xs·(Λ x. lower-map·f·(g·x))
by (*simp add*: *lower-map-def lower-bind-bind*)

lemma *ep-pair-lower-map*: *ep-pair e p* ⟹ *ep-pair* (*lower-map·e*) (*lower-map·p*)
apply *standard*
apply (*induct-tac x rule*: *lower-pd-induct*, *simp-all add*: *ep-pair.e-inverse*)
apply (*induct-tac y rule*: *lower-pd-induct*)
apply (*simp-all add*: *ep-pair.e-p-below monofun-cfun del*: *lower-plus-below-iff*)
done

lemma *deflation-lower-map*: *deflation d* ⟹ *deflation* (*lower-map·d*)
apply *standard*
apply (*induct-tac x rule*: *lower-pd-induct*, *simp-all add*: *deflation.idem*)
apply (*induct-tac x rule*: *lower-pd-induct*)
apply (*simp-all add*: *deflation.below monofun-cfun del*: *lower-plus-below-iff*)
done

lemma *finite-deflation-lower-map*:
assumes *finite-deflation d* **shows** *finite-deflation* (*lower-map·d*)
proof (*rule finite-deflation-intro*)
interpret *d*: *finite-deflation d* **by** *fact*
from *d.deflation-axioms* **show** *deflation* (*lower-map·d*)
by (*rule deflation-lower-map*)
have *finite* (*range* (*λx. d·x*)) **by** (*rule d.finite-range*)
hence *finite* (*Rep-compact-basis −‘ range* (*λx. d·x*))
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-compact-basis-inject*)
hence *finite* (*Pow* (*Rep-compact-basis −‘ range* (*λx. d·x*))) **by** *simp*
hence *finite* (*Rep-pd-basis −‘* (*Pow* (*Rep-compact-basis −‘ range* (*λx. d·x*))))
by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-pd-basis-inject*)
hence ∗: *finite* (*lower-principal ‘ Rep-pd-basis −‘* (*Pow* (*Rep-compact-basis −‘ range* (*λx. d·x*)))) **by** *simp*
hence *finite* (*range* (*λxs. lower-map·d·xs*))
apply (*rule rev-finite-subset*)
apply *clarsimp*
apply (*induct-tac xs rule*: *lower-pd.principal-induct*)
apply (*simp add*: *adm-mem-finite* ∗)
apply (*rename-tac t*, *induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *lower-map-unit*)
apply *simp*
apply (*subgoal-tac* ∃ *b. d·*(*Rep-compact-basis a*) = *Rep-compact-basis b*)

apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDUnit*)
apply (*rule range-eqI*)
apply (*erule sym*)
apply (*rule exI*)
apply (*rule Abs-compact-basis-inverse* [*symmetric*])
apply (*simp add*: *d.compact*)
apply (*simp only*: *lower-plus-principal* [*symmetric*] *lower-map-plus*)
apply *clarsimp*
apply (*rule imageI*)
apply (*rule vimageI2*)
apply (*simp add*: *Rep-PDPlus*)
done
thus *finite* {*xs. lower-map·d·xs* = *xs*}
by (*rule finite-range-imp-finite-fixes*)
qed

30.7 Lower powerdomain is bifinite

lemma *approx-chain-lower-map*:
assumes *approx-chain a*
shows *approx-chain* (*λi. lower-map·*(*a i*))
using *assms* **unfolding** *approx-chain-def*
by (*simp add*: *lub-APP lower-map-ID finite-deflation-lower-map*)

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

30.8 Join

definition
lower-join :: *'a*::*bifinite lower-pd lower-pd* → *'a lower-pd* **where**
lower-join = (Λ *xss. lower-bind·xss·*(Λ *xs. xs*))

lemma *lower-join-unit* [*simp*]:
lower-join·{*xs*}♭ = *xs*
unfolding *lower-join-def* **by** *simp*

lemma *lower-join-plus* [*simp*]:
lower-join·(*xss* ∪♭ *yss*) = *lower-join·xss* ∪♭ *lower-join·yss*
unfolding *lower-join-def* **by** *simp*

lemma *lower-join-bottom* [*simp*]: *lower-join·*⊥ = ⊥
unfolding *lower-join-def* **by** *simp*

lemma *lower-join-map-unit*:
lower-join·(lower-map·lower-unit·xs) = xs
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-join*:
lower-join·(lower-map·lower-join·xsss) = lower-join·(lower-join·xsss)
by (*induct xsss rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-map*:
lower-join·(lower-map·(lower-map·f)·xss) =
lower-map·f·(lower-join·xss)
by (*induct xss rule*: *lower-pd-induct*, *simp-all*)

end

31 Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

31.1 Basis preorder

definition
convex-le :: *'a::bifinite pd-basis ⇒ 'a pd-basis ⇒ bool* (**infix** ‹≤♮› *50*) **where**
convex-le = (λu v. u ≤♯ v ∧ u ≤♭ v)

lemma *convex-le-refl* [*simp*]: *t ≤♮ t*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-refl lower-le-refl*)

lemma *convex-le-trans*: ⟦*t ≤♮ u*; *u ≤♮ v*⟧ ⟹ *t ≤♮ v*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-trans lower-le-trans*)

interpretation *convex-le*: *preorder convex-le*
by (*rule preorder.intro*, *rule convex-le-refl*, *rule convex-le-trans*)

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot ≤♮ t*
unfolding *convex-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-convex-mono*: *x ⊑ y* ⟹ *PDUnit x ≤♮ PDUnit y*
unfolding *convex-le-def* **by** (*fast intro*: *PDUnit-upper-mono PDUnit-lower-mono*)

lemma *PDPlus-convex-mono*: ⟦*s ≤♮ t*; *u ≤♮ v*⟧ ⟹ *PDPlus s u ≤♮ PDPlus t v*
unfolding *convex-le-def* **by** (*fast intro*: *PDPlus-upper-mono PDPlus-lower-mono*)

lemma *convex-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a ≤♮ PDUnit b*) = (*a ⊑ b*)
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit* **by** *fast*

```
lemma convex-le-PDUnit-lemma1:
  (PDUnit a ≤♮ t) = (∀ b∈Rep-pd-basis t. a ⊑ b)
unfolding convex-le-def upper-le-def lower-le-def Rep-PDUnit
using Rep-pd-basis-nonempty [of t, folded ex-in-conv] by fast

lemma convex-le-PDUnit-PDPlus-iff [simp]:
  (PDUnit a ≤♮ PDPlus t u) = (PDUnit a ≤♮ t ∧ PDUnit a ≤♮ u)
unfolding convex-le-PDUnit-lemma1 Rep-PDPlus by fast

lemma convex-le-PDUnit-lemma2:
  (t ≤♮ PDUnit b) = (∀ a∈Rep-pd-basis t. a ⊑ b)
unfolding convex-le-def upper-le-def lower-le-def Rep-PDUnit
using Rep-pd-basis-nonempty [of t, folded ex-in-conv] by fast

lemma convex-le-PDPlus-PDUnit-iff [simp]:
  (PDPlus t u ≤♮ PDUnit a) = (t ≤♮ PDUnit a ∧ u ≤♮ PDUnit a)
unfolding convex-le-PDUnit-lemma2 Rep-PDPlus by fast

lemma convex-le-PDPlus-lemma:
  assumes z: PDPlus t u ≤♮ z
  shows ∃ v w. z = PDPlus v w ∧ t ≤♮ v ∧ u ≤♮ w
proof (intro exI conjI)
  let ?A = {b∈Rep-pd-basis z. ∃ a∈Rep-pd-basis t. a ⊑ b}
  let ?B = {b∈Rep-pd-basis z. ∃ a∈Rep-pd-basis u. a ⊑ b}
  let ?v = Abs-pd-basis ?A
  let ?w = Abs-pd-basis ?B
  have Rep-v: Rep-pd-basis ?v = ?A
    apply (rule Abs-pd-basis-inverse)
    apply (rule Rep-pd-basis-nonempty [of t, folded ex-in-conv, THEN exE])
    apply (cut-tac z, simp only: convex-le-def lower-le-def, clarify)
    apply (drule-tac x=x in bspec, simp add: Rep-PDPlus, erule bexE)
    apply (simp add: pd-basis-def)
    apply fast
    done
  have Rep-w: Rep-pd-basis ?w = ?B
    apply (rule Abs-pd-basis-inverse)
    apply (rule Rep-pd-basis-nonempty [of u, folded ex-in-conv, THEN exE])
    apply (cut-tac z, simp only: convex-le-def lower-le-def, clarify)
    apply (drule-tac x=x in bspec, simp add: Rep-PDPlus, erule bexE)
    apply (simp add: pd-basis-def)
    apply fast
    done
  show z = PDPlus ?v ?w
    apply (insert z)
    apply (simp add: convex-le-def, erule conjE)
    apply (simp add: Rep-pd-basis-inject [symmetric] Rep-PDPlus)
    apply (simp add: Rep-v Rep-w)
    apply (rule equalityI)
```

```
      apply (rule subsetI)
      apply (simp only: upper-le-def)
      apply (drule (1) bspec, erule bexE)
      apply (simp add: Rep-PDPlus)
      apply fast
     apply fast
    done
  show t ≤♮ ?v u ≤♮ ?w
     using z by (simp-all add: convex-le-def upper-le-def lower-le-def Rep-PDPlus
Rep-v Rep-w) fast+
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
proof (induct t arbitrary: u rule: pd-basis-induct)
  case (PDUnit a)
  then show ?case
  proof (induct u rule: pd-basis-induct1)
    case (PDUnit b)
    then show ?case by (simp add: 3)
  next
    case (PDPlus b t)
    have P (PDPlus (PDUnit a) (PDUnit a)) (PDPlus (PDUnit b) t)
      by (rule 4 [OF 3]) (use PDPlus in simp-all)
    then show ?case by (simp add: PDPlus-absorb)
  qed
next
  case PDPlus
  from PDPlus(1,2) show ?case
    using convex-le-PDPlus-lemma [OF PDPlus(3)] by (auto simp add: 4)
qed
```

31.2 Type definition

```
typedef 'a::bifinite convex-pd (‹(‹notation=‹postfix convex-pd››'(-')♮)›) =
  {S::'a pd-basis set. convex-le.ideal S}
by (rule convex-le.ex-ideal)

instantiation convex-pd :: (bifinite) below
begin

definition
  x ⊑ y ⟷ Rep-convex-pd x ⊆ Rep-convex-pd y
```

instance ..
end

instance *convex-pd* :: (*bifinite*) *po*
using *type-definition-convex-pd below-convex-pd-def*
by (*rule convex-le.typedef-ideal-po*)

instance *convex-pd* :: (*bifinite*) *cpo*
using *type-definition-convex-pd below-convex-pd-def*
by (*rule convex-le.typedef-ideal-cpo*)

definition
convex-principal :: *'a::bifinite pd-basis* ⇒ *'a convex-pd* **where**
convex-principal t = *Abs-convex-pd* {*u. u* ≤♮ *t*}

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

Convex powerdomain is pointed

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) ⊑ *ys*
by (*induct ys rule*: *convex-pd.principal-induct*, *simp*, *simp*)

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

lemma *inst-convex-pd-pcpo*: ⊥ = *convex-principal* (*PDUnit compact-bot*)
by (*rule convex-pd-minimal* [*THEN bottomI*, *symmetric*])

31.3 Monadic unit and plus

definition
convex-unit :: *'a::bifinite* → *'a convex-pd* **where**
convex-unit = *compact-basis.extension* (λ*a. convex-principal* (*PDUnit a*))

definition
convex-plus :: *'a::bifinite convex-pd* → *'a convex-pd* → *'a convex-pd* **where**
convex-plus = *convex-pd.extension* (λ*t. convex-pd.extension* (λ*u.*
convex-principal (*PDPlus t u*)))

abbreviation
convex-add :: *'a::bifinite convex-pd* ⇒ *'a convex-pd* ⇒ *'a convex-pd*
(**infixl** ‹∪♮› *65*) **where**
xs ∪♮ *ys* == *convex-plus·xs·ys*

syntax
-convex-pd :: *args* ⇒ *logic* (‹(‹*indent=1 notation=*‹*mixfix convex-pd enumera-*

tion››{-}♮)›)
translations
  {*x*,*xs*}♮ == {*x*}♮ ∪♮ {*xs*}♮
  {*x*}♮ == *CONST convex-unit·x*

lemma *convex-unit-Rep-compact-basis* [*simp*]:
  {*Rep-compact-basis a*}♮ = *convex-principal* (*PDUnit a*)
unfolding *convex-unit-def*
by (*simp add*: *compact-basis.extension-principal PDUnit-convex-mono*)

lemma *convex-plus-principal* [*simp*]:
  convex-principal t ∪♮ *convex-principal u* = *convex-principal* (*PDPlus t u*)
unfolding *convex-plus-def*
by (*simp add*: *convex-pd.extension-principal*
    convex-pd.extension-mono PDPlus-convex-mono)

interpretation *convex-add*: *semilattice convex-add* **proof**
  fix *xs ys zs* :: *'a convex-pd*
  show (*xs* ∪♮ *ys*) ∪♮ *zs* = *xs* ∪♮ (*ys* ∪♮ *zs*)
    apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
    apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
    apply (*induct zs rule*: *convex-pd.principal-induct*, *simp*)
    apply (*simp add*: *PDPlus-assoc*)
    done
  show *xs* ∪♮ *ys* = *ys* ∪♮ *xs*
    apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
    apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
    apply (*simp add*: *PDPlus-commute*)
    done
  show *xs* ∪♮ *xs* = *xs*
    apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
    apply (*simp add*: *PDPlus-absorb*)
    done
qed

lemmas *convex-plus-assoc* = *convex-add.assoc*
lemmas *convex-plus-commute* = *convex-add.commute*
lemmas *convex-plus-absorb* = *convex-add.idem*
lemmas *convex-plus-left-commute* = *convex-add.left-commute*
lemmas *convex-plus-left-absorb* = *convex-add.left-idem*

Useful for *simp add*: *convex-plus-ac*

lemmas *convex-plus-ac* =
  convex-plus-assoc convex-plus-commute convex-plus-left-commute

Useful for *simp only*: *convex-plus-aci*

lemmas *convex-plus-aci* =
  convex-plus-ac convex-plus-absorb convex-plus-left-absorb

lemma *convex-unit-below-plus-iff* [*simp*]:
$\{x\}\natural \sqsubseteq ys \cup\natural\ zs \longleftrightarrow \{x\}\natural \sqsubseteq ys \wedge \{x\}\natural \sqsubseteq zs$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct zs rule*: *convex-pd.principal-induct*, *simp*)
apply *simp*
done

lemma *convex-plus-below-unit-iff* [*simp*]:
$xs \cup\natural\ ys \sqsubseteq \{z\}\natural \longleftrightarrow xs \sqsubseteq \{z\}\natural \wedge ys \sqsubseteq \{z\}\natural$
apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct z rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
done

lemma *convex-unit-below-iff* [*simp*]: $\{x\}\natural \sqsubseteq \{y\}\natural \longleftrightarrow x \sqsubseteq y$
apply (*induct x rule*: *compact-basis.principal-induct*, *simp*)
apply (*induct y rule*: *compact-basis.principal-induct*, *simp*)
apply *simp*
done

lemma *convex-unit-eq-iff* [*simp*]: $\{x\}\natural = \{y\}\natural \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *convex-unit-strict* [*simp*]: $\{\bot\}\natural = \bot$
using *convex-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add*: *inst-convex-pd-pcpo*)

lemma *convex-unit-bottom-iff* [*simp*]: $\{x\}\natural = \bot \longleftrightarrow x = \bot$
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-unit-eq-iff*)

lemma *compact-convex-unit*: *compact* $x \Longrightarrow$ *compact* $\{x\}\natural$
by (*auto dest*!: *compact-basis.compact-imp-principal*)

lemma *compact-convex-unit-iff* [*simp*]: *compact* $\{x\}\natural \longleftrightarrow$ *compact* x
apply (*safe elim*!: *compact-convex-unit*)
apply (*simp only*: *compact-def convex-unit-below-iff* [*symmetric*])
apply (*erule adm-subst* [*OF cont-Rep-cfun2*])
done

lemma *compact-convex-plus* [*simp*]:
$[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* $(xs \cup\natural\ ys)$
by (*auto dest*!: *convex-pd.compact-imp-principal*)

31.4 Induction rules

lemma *convex-pd-induct1*:
assumes *P*: *adm P*

assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\natural;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\natural \cup\natural\ ys)$
shows $P\ (xs::'a::bifinite\ convex\text{-}pd)$
proof (*induct xs rule*: *convex-pd.principal-induct*)
show *P* (*convex-principal a*) **for** *a*
proof (*induct a rule*: *pd-basis-induct1*)
case *PDUnit*
show *?case* **by** (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*]) (*rule unit*)
next
case *PDPlus*
show *?case*
by (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*] *convex-plus-principal* [*symmetric*])
(*rule insert* [*OF unit PDPlus*])
qed
qed (*rule P*)

lemma *convex-pd-induct* [*case-names adm convex-unit convex-plus*, *induct type*: *convex-pd*]:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs \cup\natural\ ys)$
shows $P\ (xs::'a::bifinite\ convex\text{-}pd)$
proof (*induct xs rule*: *convex-pd.principal-induct*)
show *P* (*convex-principal a*) **for** *a*
proof (*induct a rule*: *pd-basis-induct*)
case *PDUnit*
then show *?case* **by** (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*] *unit*)
next
case *PDPlus*
then show *?case* **by** (*simp only*: *convex-plus-principal* [*symmetric*] *plus*)
qed
qed (*rule P*)

31.5 Monadic bind

definition
convex-bind-basis ::
$'a::bifinite\ pd\text{-}basis \Rightarrow ('a \rightarrow 'b\ convex\text{-}pd) \rightarrow 'b::bifinite\ convex\text{-}pd$ **where**
convex-bind-basis = *fold-pd*
$(\lambda a.\ \Lambda\ f.\ f\cdot(Rep\text{-}compact\text{-}basis\ a))$
$(\lambda x\ y.\ \Lambda\ f.\ x\cdot f \cup\natural\ y\cdot f)$

lemma *ACI-convex-bind*:
semilattice $(\lambda x\ y.\ \Lambda\ f.\ x\cdot f \cup\natural\ y\cdot f)$
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*) =
(Λ *f*. *f*·(*Rep-compact-basis a*))
convex-bind-basis (*PDPlus t u*) =
(Λ *f*. *convex-bind-basis t*·*f* ∪♮ *convex-bind-basis u*·*f*)
unfolding *convex-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-convex-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-convex-bind*])
done

lemma *convex-bind-basis-mono*:
t ≤♮ *u* ⟹ *convex-bind-basis t* ⊑ *convex-bind-basis u*
apply (*erule convex-le-induct*)
apply (*erule* (*1*) *below-trans*)
apply (*simp add*: *monofun-LAM monofun-cfun*)
apply (*simp add*: *monofun-LAM monofun-cfun*)
done

definition
convex-bind :: ′*a*::*bifinite convex-pd* → (′*a* → ′*b convex-pd*) → ′*b*::*bifinite convex-pd*
where
convex-bind = *convex-pd.extension convex-bind-basis*

syntax
-convex-bind :: [*logic*, *logic*, *logic*] ⇒ *logic*
(‹(‹*indent=3 notation=*‹*binder convex-bind*››⋃♮-∈-./ -)› [*0*, *0*, *10*] *10*)

translations
⋃♮*x*∈*xs*. *e* == *CONST convex-bind*·*xs*·(Λ *x*. *e*)

lemma *convex-bind-principal* [*simp*]:
convex-bind·(*convex-principal t*) = *convex-bind-basis t*
unfolding *convex-bind-def*
apply (*rule convex-pd.extension-principal*)
apply (*erule convex-bind-basis-mono*)
done

lemma *convex-bind-unit* [*simp*]:
convex-bind·{*x*}♮·*f* = *f*·*x*
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-bind-plus* [*simp*]:
convex-bind·(*xs* ∪♮ *ys*)·*f* = *convex-bind*·*xs*·*f* ∪♮ *convex-bind*·*ys*·*f*
by (*induct xs rule*: *convex-pd.principal-induct*, *simp*,

induct ys rule: *convex-pd.principal-induct*, *simp*, *simp*)

lemma *convex-bind-strict* [*simp*]: *convex-bind*$\cdot\bot\cdot$*f* = *f*$\cdot\bot$
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-bind-unit*)

lemma *convex-bind-bind*:
convex-bind$\cdot$(*convex-bind*$\cdot$*xs*$\cdot$*f*)$\cdot$*g* =
convex-bind$\cdot$*xs*$\cdot$(Λ *x*. *convex-bind*$\cdot$(*f*$\cdot$*x*)$\cdot$*g*)
by (*induct xs*, *simp-all*)

31.6 Map

definition
convex-map :: ('*a*::*bifinite* $\rightarrow$ '*b*) $\rightarrow$ '*a convex-pd* $\rightarrow$ '*b*::*bifinite convex-pd* **where**
convex-map = (Λ *f xs*. *convex-bind*$\cdot$*xs*$\cdot$(Λ *x*. {*f*$\cdot$*x*}$\natural$))

lemma *convex-map-unit* [*simp*]:
convex-map$\cdot$*f*$\cdot${*x*}$\natural$ = {*f*$\cdot$*x*}$\natural$
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-plus* [*simp*]:
convex-map$\cdot$*f*$\cdot$(*xs* $\cup\natural$ *ys*) = *convex-map*$\cdot$*f*$\cdot$*xs* $\cup\natural$ *convex-map*$\cdot$*f*$\cdot$*ys*
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-bottom* [*simp*]: *convex-map*$\cdot$*f*$\cdot\bot$ = {*f*$\cdot\bot$}$\natural$
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-ident*: *convex-map*$\cdot$(Λ *x*. *x*)$\cdot$*xs* = *xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-map-ID*: *convex-map*$\cdot$*ID* = *ID*
by (*simp add*: *cfun-eq-iff ID-def convex-map-ident*)

lemma *convex-map-map*:
convex-map$\cdot$*f*$\cdot$(*convex-map*$\cdot$*g*$\cdot$*xs*) = *convex-map*$\cdot$(Λ *x*. *f*$\cdot$(*g*$\cdot$*x*))$\cdot$*xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-bind-map*:
convex-bind$\cdot$(*convex-map*$\cdot$*f*$\cdot$*xs*)$\cdot$*g* = *convex-bind*$\cdot$*xs*$\cdot$(Λ *x*. *g*$\cdot$(*f*$\cdot$*x*))
by (*simp add*: *convex-map-def convex-bind-bind*)

lemma *convex-map-bind*:
convex-map$\cdot$*f*$\cdot$(*convex-bind*$\cdot$*xs*$\cdot$*g*) = *convex-bind*$\cdot$*xs*$\cdot$(Λ *x*. *convex-map*$\cdot$*f*$\cdot$(*g*$\cdot$*x*))
by (*simp add*: *convex-map-def convex-bind-bind*)

lemma *ep-pair-convex-map*: *ep-pair e p* $\Longrightarrow$ *ep-pair* (*convex-map*$\cdot$*e*) (*convex-map*$\cdot$*p*)
apply *standard*
apply (*induct-tac x rule*: *convex-pd-induct*, *simp-all add*: *ep-pair.e-inverse*)
apply (*induct-tac y rule*: *convex-pd-induct*)

apply (*simp-all add*: *ep-pair.e-p-below monofun-cfun*)
done

lemma *deflation-convex-map*: *deflation d* $\Longrightarrow$ *deflation* (*convex-map*·*d*)
apply *standard*
apply (*induct-tac x rule*: *convex-pd-induct*, *simp-all add*: *deflation.idem*)
apply (*induct-tac x rule*: *convex-pd-induct*)
apply (*simp-all add*: *deflation.below monofun-cfun*)
done

lemma *finite-deflation-convex-map*:
  assumes *finite-deflation d* **shows** *finite-deflation* (*convex-map*·*d*)
proof (*rule finite-deflation-intro*)
  interpret *d*: *finite-deflation d* **by** *fact*
  from *d.deflation-axioms* **show** *deflation* (*convex-map*·*d*)
    by (*rule deflation-convex-map*)
  have *finite* (*range* ($\lambda x.$ *d*·*x*)) **by** (*rule d.finite-range*)
  hence *finite* (*Rep-compact-basis* −' *range* ($\lambda x.$ *d*·*x*))
    by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-compact-basis-inject*)
  hence *finite* (*Pow* (*Rep-compact-basis* −' *range* ($\lambda x.$ *d*·*x*))) **by** *simp*
  hence *finite* (*Rep-pd-basis* −' (*Pow* (*Rep-compact-basis* −' *range* ($\lambda x.$ *d*·*x*))))
    by (*rule finite-vimageI*, *simp add*: *inj-on-def Rep-pd-basis-inject*)
  hence ∗: *finite* (*convex-principal* ' *Rep-pd-basis* −' (*Pow* (*Rep-compact-basis* −'
range ($\lambda x.$ *d*·*x*)))) **by** *simp*
  hence *finite* (*range* ($\lambda xs.$ *convex-map*·*d*·*xs*))
    apply (*rule rev-finite-subset*)
    apply *clarsimp*
    apply (*induct-tac xs rule*: *convex-pd.principal-induct*)
    apply (*simp add*: *adm-mem-finite* ∗)
    apply (*rename-tac t*, *induct-tac t rule*: *pd-basis-induct*)
    apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*] *convex-map-unit*)
    apply *simp*
    apply (*subgoal-tac* ∃ *b*. *d*·(*Rep-compact-basis a*) = *Rep-compact-basis b*)
    apply *clarsimp*
    apply (*rule imageI*)
    apply (*rule vimageI2*)
    apply (*simp add*: *Rep-PDUnit*)
    apply (*rule range-eqI*)
    apply (*erule sym*)
    apply (*rule exI*)
    apply (*rule Abs-compact-basis-inverse* [*symmetric*])
    apply (*simp add*: *d.compact*)
    apply (*simp only*: *convex-plus-principal* [*symmetric*] *convex-map-plus*)
    apply *clarsimp*
    apply (*rule imageI*)
    apply (*rule vimageI2*)
    apply (*simp add*: *Rep-PDPlus*)
    done

thus *finite* {*xs*. *convex-map·d·xs* = *xs*}
  by (*rule finite-range-imp-finite-fixes*)
qed

31.7 Convex powerdomain is bifinite

lemma *approx-chain-convex-map*:
  assumes *approx-chain a*
  shows *approx-chain* (λ*i*. *convex-map*·(*a i*))
  using *assms* **unfolding** *approx-chain-def*
  by (*simp add*: *lub-APP convex-map-ID finite-deflation-convex-map*)

instance *convex-pd* :: (*bifinite*) *bifinite*
proof
  show ∃(*a*::*nat* ⇒ ′*a convex-pd* → ′*a convex-pd*). *approx-chain a*
    using *bifinite* [**where** ′*a*=′*a*]
    by (*fast intro*!: *approx-chain-convex-map*)
qed

31.8 Join

definition
  convex-join :: ′*a*::*bifinite convex-pd convex-pd* → ′*a convex-pd* **where**
  convex-join = (Λ *xss*. *convex-bind·xss*·(Λ *xs*. *xs*))

lemma *convex-join-unit* [*simp*]:
  convex-join·{*xs*}♮ = *xs*
unfolding *convex-join-def* **by** *simp*

lemma *convex-join-plus* [*simp*]:
  convex-join·(*xss* ∪♮ *yss*) = *convex-join·xss* ∪♮ *convex-join·yss*
unfolding *convex-join-def* **by** *simp*

lemma *convex-join-bottom* [*simp*]: *convex-join*·⊥ = ⊥
unfolding *convex-join-def* **by** *simp*

lemma *convex-join-map-unit*:
  convex-join·(*convex-map·convex-unit·xs*) = *xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-join*:
  convex-join·(*convex-map·convex-join·xsss*) = *convex-join*·(*convex-join·xsss*)
by (*induct xsss rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-map*:
  convex-join·(*convex-map*·(*convex-map·f*)·*xss*) =
   convex-map·f·(*convex-join·xss*)
by (*induct xss rule*: *convex-pd-induct*, *simp-all*)

31.9 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: $t \leq\natural\ u \Longrightarrow t \leq\sharp\ u$
unfolding *convex-le-def* **by** *simp*

definition
  convex-to-upper :: *'a::bifinite convex-pd* → *'a upper-pd* **where**
  convex-to-upper = *convex-pd.extension upper-principal*

lemma *convex-to-upper-principal* [*simp*]:
  convex-to-upper·(*convex-principal t*) = *upper-principal t*
unfolding *convex-to-upper-def*
apply (*rule convex-pd.extension-principal*)
apply (*rule upper-pd.principal-mono*)
apply (*erule convex-le-imp-upper-le*)
done

lemma *convex-to-upper-unit* [*simp*]:
  convex-to-upper·{*x*}♮ = {*x*}♯
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-to-upper-plus* [*simp*]:
  convex-to-upper·(*xs* ∪♮ *ys*) = *convex-to-upper·xs* ∪♯ *convex-to-upper·ys*
by (*induct xs rule*: *convex-pd.principal-induct*, *simp*,
    induct ys rule: *convex-pd.principal-induct*, *simp*, *simp*)

lemma *convex-to-upper-bind* [*simp*]:
  convex-to-upper·(*convex-bind·xs·f*) =
    upper-bind·(*convex-to-upper·xs*)·(*convex-to-upper oo f*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-upper-map* [*simp*]:
  convex-to-upper·(*convex-map·f·xs*) = *upper-map·f·*(*convex-to-upper·xs*)
by (*simp add*: *convex-map-def upper-map-def cfcomp-LAM*)

lemma *convex-to-upper-join* [*simp*]:
  convex-to-upper·(*convex-join·xss*) =
    upper-bind·(*convex-to-upper·xss*)*·convex-to-upper*
by (*simp add*: *convex-join-def upper-join-def cfcomp-LAM eta-cfun*)

Convex to lower

lemma *convex-le-imp-lower-le*: $t \leq\natural\ u \Longrightarrow t \leq\flat\ u$
unfolding *convex-le-def* **by** *simp*

definition
  convex-to-lower :: *'a::bifinite convex-pd* → *'a lower-pd* **where**
  convex-to-lower = *convex-pd.extension lower-principal*

lemma *convex-to-lower-principal* [*simp*]:
  convex-to-lower·(*convex-principal t*) = *lower-principal t*
unfolding *convex-to-lower-def*
apply (*rule convex-pd.extension-principal*)
apply (*rule lower-pd.principal-mono*)
apply (*erule convex-le-imp-lower-le*)
done

lemma *convex-to-lower-unit* [*simp*]:
  convex-to-lower·{*x*}♮ = {*x*}♭
by (*induct x rule*: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-to-lower-plus* [*simp*]:
  convex-to-lower·(*xs* ∪♮ *ys*) = *convex-to-lower*·*xs* ∪♭ *convex-to-lower*·*ys*
by (*induct xs rule*: *convex-pd.principal-induct*, *simp*,
    induct ys rule: *convex-pd.principal-induct*, *simp*, *simp*)

lemma *convex-to-lower-bind* [*simp*]:
  convex-to-lower·(*convex-bind*·*xs*·*f*) =
    lower-bind·(*convex-to-lower*·*xs*)·(*convex-to-lower oo f*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

lemma *convex-to-lower-map* [*simp*]:
  convex-to-lower·(*convex-map*·*f*·*xs*) = *lower-map*·*f*·(*convex-to-lower*·*xs*)
by (*simp add*: *convex-map-def lower-map-def cfcomp-LAM*)

lemma *convex-to-lower-join* [*simp*]:
  convex-to-lower·(*convex-join*·*xss*) =
    lower-bind·(*convex-to-lower*·*xss*)·*convex-to-lower*
by (*simp add*: *convex-join-def lower-join-def cfcomp-LAM eta-cfun*)

Ordering property

lemma *convex-pd-below-iff*:
  (*xs* ⊑ *ys*) =
    (*convex-to-upper*·*xs* ⊑ *convex-to-upper*·*ys* ∧
     convex-to-lower·*xs* ⊑ *convex-to-lower*·*ys*)
apply (*induct xs rule*: *convex-pd.principal-induct*, *simp*)
apply (*induct ys rule*: *convex-pd.principal-induct*, *simp*)
apply (*simp add*: *convex-le-def*)
done

lemmas *convex-plus-below-plus-iff* =
  convex-pd-below-iff [**where** *xs*=*xs* ∪♮ *ys* **and** *ys*=*zs* ∪♮ *ws*]
  for *xs ys zs ws*

lemmas *convex-pd-below-simps* =
  convex-unit-below-plus-iff
  convex-plus-below-unit-iff
  convex-plus-below-plus-iff

convex-unit-below-iff
convex-to-upper-unit
convex-to-upper-plus
convex-to-lower-unit
convex-to-lower-plus
upper-pd-below-simps
lower-pd-below-simps

end

32 Powerdomains

theory *Powerdomains*
imports *ConvexPD Domain*
begin

32.1 Universal domain embeddings

definition *upper-emb* = *udom-emb* ($\lambda i.$ *upper-map*·(*udom-approx i*))
definition *upper-prj* = *udom-prj* ($\lambda i.$ *upper-map*·(*udom-approx i*))

definition *lower-emb* = *udom-emb* ($\lambda i.$ *lower-map*·(*udom-approx i*))
definition *lower-prj* = *udom-prj* ($\lambda i.$ *lower-map*·(*udom-approx i*))

definition *convex-emb* = *udom-emb* ($\lambda i.$ *convex-map*·(*udom-approx i*))
definition *convex-prj* = *udom-prj* ($\lambda i.$ *convex-map*·(*udom-approx i*))

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

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

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

32.2 Deflation combinators

definition *upper-defl* :: *udom defl* $\to$ *udom defl*
where *upper-defl* = *defl-fun1 upper-emb upper-prj upper-map*

definition *lower-defl* :: *udom defl* $\to$ *udom defl*
where *lower-defl* = *defl-fun1 lower-emb lower-prj lower-map*

definition *convex-defl* :: *udom defl* $\to$ *udom defl*
where *convex-defl* = *defl-fun1 convex-emb convex-prj convex-map*

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

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

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

32.3 Domain class instances

instantiation *upper-pd* :: (*domain*) *domain*
begin

definition
emb = upper-emb oo upper-map·emb

definition
prj = upper-map·prj oo upper-prj

definition
defl (t::'a upper-pd itself) = upper-defl·DEFL('a)

definition
(*liftemb* :: *'a upper-pd u → udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u → 'a upper-pd u*) = *u-map·prj*

definition
liftdefl (t::'a upper-pd itself) = liftdefl-of·DEFL('a upper-pd)

instance proof
show *ep-pair emb (prj :: udom → 'a upper-pd)*
unfolding *emb-upper-pd-def prj-upper-pd-def*
by (*intro ep-pair-comp ep-pair-upper ep-pair-upper-map ep-pair-emb-prj*)
next
show *cast·DEFL('a upper-pd) = emb oo (prj :: udom → 'a upper-pd)*
unfolding *emb-upper-pd-def prj-upper-pd-def defl-upper-pd-def cast-upper-defl*
by (*simp add: cast-DEFL oo-def cfun-eq-iff upper-map-map*)
qed (*fact liftemb-upper-pd-def liftprj-upper-pd-def liftdefl-upper-pd-def*)+

end

instantiation *lower-pd* :: (*domain*) *domain*
begin

definition
emb = *lower-emb oo lower-map·emb*

definition
prj = *lower-map·prj oo lower-prj*

definition
defl (*t*::′*a lower-pd itself*) = *lower-defl·DEFL*(′*a*)

definition
(*liftemb* :: ′*a lower-pd u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → ′*a lower-pd u*) = *u-map·prj*

definition
liftdefl (*t*::′*a lower-pd itself*) = *liftdefl-of·DEFL*(′*a lower-pd*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → ′*a lower-pd*)
unfolding *emb-lower-pd-def prj-lower-pd-def*
by (*intro ep-pair-comp ep-pair-lower ep-pair-lower-map ep-pair-emb-prj*)
next
show *cast·DEFL*(′*a lower-pd*) = *emb oo* (*prj* :: *udom* → ′*a lower-pd*)
unfolding *emb-lower-pd-def prj-lower-pd-def defl-lower-pd-def cast-lower-defl*
by (*simp add*: *cast-DEFL oo-def cfun-eq-iff lower-map-map*)
qed (*fact liftemb-lower-pd-def liftprj-lower-pd-def liftdefl-lower-pd-def*)+

end

instantiation *convex-pd* :: (*domain*) *domain*
begin

definition
emb = *convex-emb oo convex-map·emb*

definition
prj = *convex-map·prj oo convex-prj*

definition
defl (*t*::′*a convex-pd itself*) = *convex-defl·DEFL*(′*a*)

definition
(*liftemb* :: ′*a convex-pd u* → *udom u*) = *u-map·emb*

definition
(*liftprj* :: *udom u* → *'a convex-pd u*) = *u-map·prj*

definition
liftdefl (*t*::*'a convex-pd itself*) = *liftdefl-of·DEFL*(*'a convex-pd*)

instance proof
show *ep-pair emb* (*prj* :: *udom* → *'a convex-pd*)
unfolding *emb-convex-pd-def prj-convex-pd-def*
by (*intro ep-pair-comp ep-pair-convex ep-pair-convex-map ep-pair-emb-prj*)
next
show *cast·DEFL*(*'a convex-pd*) = *emb oo* (*prj* :: *udom* → *'a convex-pd*)
unfolding *emb-convex-pd-def prj-convex-pd-def defl-convex-pd-def cast-convex-defl*
by (*simp add*: *cast-DEFL oo-def cfun-eq-iff convex-map-map*)
qed (*fact liftemb-convex-pd-def liftprj-convex-pd-def liftdefl-convex-pd-def*)+

end

lemma *DEFL-upper*: *DEFL*(*'a*::*domain upper-pd*) = *upper-defl·DEFL*(*'a*)
by (*rule defl-upper-pd-def*)

lemma *DEFL-lower*: *DEFL*(*'a*::*domain lower-pd*) = *lower-defl·DEFL*(*'a*)
by (*rule defl-lower-pd-def*)

lemma *DEFL-convex*: *DEFL*(*'a*::*domain convex-pd*) = *convex-defl·DEFL*(*'a*)
by (*rule defl-convex-pd-def*)

32.4 Isomorphic deflations

lemma *isodefl-upper*:
isodefl d t ⟹ *isodefl* (*upper-map·d*) (*upper-defl·t*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-upper-defl cast-isodefl*)
apply (*simp add*: *emb-upper-pd-def prj-upper-pd-def*)
apply (*simp add*: *upper-map-map*)
done

lemma *isodefl-lower*:
isodefl d t ⟹ *isodefl* (*lower-map·d*) (*lower-defl·t*)
apply (*rule isodeflI*)
apply (*simp add*: *cast-lower-defl cast-isodefl*)
apply (*simp add*: *emb-lower-pd-def prj-lower-pd-def*)
apply (*simp add*: *lower-map-map*)
done

lemma *isodefl-convex*:
isodefl d t ⟹ *isodefl* (*convex-map·d*) (*convex-defl·t*)
apply (*rule isodeflI*)

apply (*simp add*: *cast-convex-defl cast-isodefl*)
apply (*simp add*: *emb-convex-pd-def prj-convex-pd-def*)
apply (*simp add*: *convex-map-map*)
done

32.5 Domain package setup for powerdomains

lemmas [*domain-defl-simps*] = *DEFL-upper DEFL-lower DEFL-convex*
lemmas [*domain-map-ID*] = *upper-map-ID lower-map-ID convex-map-ID*
lemmas [*domain-isodefl*] = *isodefl-upper isodefl-lower isodefl-convex*

lemmas [*domain-deflation*] =
deflation-upper-map deflation-lower-map deflation-convex-map

setup ‹
fold Domain-Take-Proofs.add-rec-type
[(***type-name*** ‹*upper-pd*›, [*true*]),
(***type-name*** ‹*lower-pd*›, [*true*]),
(***type-name*** ‹*convex-pd*›, [*true*])]
›

end

theory *HOLCF*
imports
Main
Domain
Powerdomains
begin

default-sort *domain*

end